École Centrale Paris

1re année d'études

Algorithmique

Jean-Jacques Dhénin

2000–2001

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

Programmation structurée
Itération, récurrence et récursivité

1. Algorithme

Algorithme[1]

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.

1. *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.

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

(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).

$$\begin{array}{ccc} & 4 & \\ 1 & \times & 1 \\ & 7 & \end{array}$$

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.

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

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.

– 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° 3, Association pour le développement de la culture scientifique.

2. 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 `15` à `30`.

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 vérifie 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) ;
}
```

Fig.2.1 – Afficher les nombres limites

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.

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).

1. Page 11 la propriété de l'affectation
2. Page 81 Table de caractères

La norme IEEE 754

La norme IEEE[3] *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 Fig.2.2 et 3) et quatre modes d'arrondis (vers ∞, vers $-\infty$, vers 0, au plus près) sont définis, ainsi que des formats dits *étendus*.

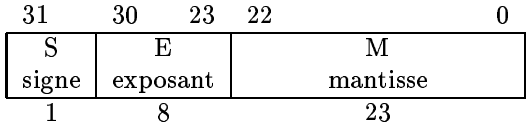


Fig.2.2 – Format flottant simple précision

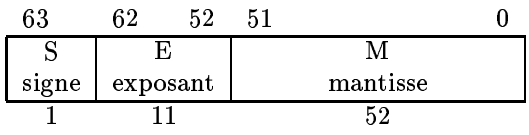


Fig.2.3 – Format flottant double précision

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[4] ; 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

Fig.2.4 – La norme IEEE 754

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[5] 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.

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

4. 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.

5. Page 11

3. L'affectation

Déroulement linéaire

La forme la plus simple de l'algorithme[1] 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 dont la plus simple est l'*affectation.*

Propriété de l'affectation

Nous noterons `/* ...*/` une description d'état, c'est à dire une relation entre les variables du programme et des constantes.

Lorsque l'on fait une affectation, 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.

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

Dans l'exemple ci-dessus, 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++`.

Autre exemple :

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

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

Supposons que, dans la relation (P), x vaut x_0, dans la relation (Q),
x vaut $\frac{1}{x_0} + y \Rightarrow \frac{1}{x_0} + y > y + 1$
$0 < x_0 < 1 \Rightarrow \frac{1}{x_0} > 1 \Rightarrow \frac{1}{x_0} + y > y + 1$

Fig.3.1 – Propriété de l'affectation

1. Page 7 la définition.

Propriété de l'enchaînement

Le déroulement linéaire, ou enchaînement, est une succession d'instructions pour 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 x et y, avec	Nous savons que
(P) `/*` $0 < x < 1$ `*/`	`/*` $0 < x < 1$ `*/`
	$x = \frac{1}{x} + y$;
Pour obtenir	`/*` $x > y + 1$ `*/`
(Q) `/*` $x > y + 2$ `*/`	
	et que
on exécute	`/*` $x > n$ `*/`
$x = \frac{1}{x} + y$;	$x = x + 1$;
$x = x + 1$;	`/*` $x > n + 1$ `*/`

Fig.3.2 – Propriété de l'enchaînement

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) ;

Fig.3.3 - Programme (faux)
pour permutter deux nombres

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. Soient donc a et b les valeurs initiales des variables x et y

`/*` $x == a$ `et` $y == b$ `*/`

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[2] 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

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

De même, retrancher b de a ne change pas a

/* $x == a$ **et** $y == a$ */ $x = x - y$;
/* $x == 0$ **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.

2. Page 9, l'ensemble des réels est discontinu

4. Choix conditionnel

Les sélections : choisir c'est exclure

L'algorithme linéaire correspond à un schéma très simple : les actions s'enchaînent dans un ordre figé. La réalité est plus souvent construite suivant un schéma conditionnel. La mise en œuvre d'algorithmes conditionnels permet de supprimer le déterminisme lié aux algorithmes linéaires. En programmant la prise de décision, nous donnerons à l'ordinateur la capacité de *raisonner*, c'est-à-dire de suivre une démarche logique (exemple : jouer aux échecs) donnant au profane l'impression que l'ordinateur est capable de *penser*.

La puissance de calcul d'un ordinateur est mise en œuvre lorsqu'il évalue les expressions contenues dans les lignes de programmes. Le pouvoir de décision est utilisé pour déterminer l'ordre d'exécution des lignes.

Pour bien saisir le concept de prise de décision d'un ordinateur, il faut savoir ce qu'est le compteur programme. Le compteur programme est la partie du système interne de l'ordinateur capable d'indiquer à l'ordinateur la prochaine ligne à exécuter. À moins d'une indication contraire, le compteur programme s'incrémente à la fin de chaque ligne afin d'indiquer la prochaine ligne du programme.

Prenons comme exemple un laboratoire de chimie. Un ordinateur n'y sera pas d'une grande utilité si sa fonction se limite à ouvrir une valve lorsqu'un technicien appuie sur le bouton START . Dans ce cas, le technicien ferait aussi bien de l'ouvrir lui-même. Toutefois, l'ordinateur exécutera une tâche beaucoup plus utile s'il ouvre la valve lorsqu'on appuie sur START et la ferme lorsqu'on atteint la valeur donnée du pH. Il peut surpasser le technicien par exemple dans l'utilisation de valves télécommandées et des dispositifs de mesure électroniques du pH. Dans cet exemple, le côté utile de l'ordinateur demeure sa capacité de décider de la fermeture de la valve. En fait, c'est le programmeur qui spécifie les critères de décision. Ces critères sont ensuite communiqués à l'ordinateur grâce aux structures d'exécution conditionnelle du programme. En conséquence, L'ordinateur est capable d'interpréter la décision du programmeur à une vitesse et à une précision plus grandes que celles d'un être humain.

Il existe diverses applications d'instructions d'exécution conditionnelle

1. Choix conditionnel d'un segment parmi deux,
2. Exécution conditionnelle d'un segment (Schéma conditionnel simplifié).
3. Choix conditionnel d'un segment parmi plusieurs.

L'alternative

Grâce à l'instruction **if** ... **else**, vous pouvez exécuter ou sauter un segment de programme. Cette instruction contient une expression qui peut être vraie ou fausse. Si elle est vraie (différente de zéro), le segment conditionnel est exécuté. Si elle est fausse (égale à zéro), le segment conditionnel est ignoré.

Le segment conditionnel peut être soit une seule instruction, soit une partie de programme comprenant un nombre quelconque d'instructions.

```
if      (condition)
        { action(s) 1 }
else
        { action(s) 2 }
```

La condition ou *prédicat*[1] est *une fonction propositionnelle* dont le résultat est booléen c'est-à-dire a 2 valeurs : vrai, faux[2].

Il existe un schéma conditionnel simplifié qui omet le deuxième terme de l'alternative.

Les actions internes 1 et 2 du schéma conditionnel peuvent être elles-mêmes conditionnelles. On obtient alors des structures conditionnelles emboîtées[3].

Propriété de l'alternative

La succession d'instructions poursuit un objectif : se rapprocher d'un résultat prévu. Avec la structure de contrôle **if** ... **else**, il y a 2 chemins possibles pour se rapprocher du même objectif, il y a 2 possibilités de modifier les relations entre les variables.

Supposons

- les relations P et C entre les variables, et qu'une action A conduise à une nouvelle relation Q,

1. cf. *logique* en troisième partie
2. il existe en langage C une abréviation pour ce schéma ; par exemple le calcul du maximum de 2 nombres peut s'écrire : `max = a > b? a: b ;`
3. Page 15

- les relations P et $\overline{C}$ entre les variables, et qu'une action B conduise à la même relation Q.

La propriété de la structure de contrôle **if** ... **else**, s'écrit :

```
/* P */ if (C) A ; else B ; /* Q */
```

Exemple :

pour expliciter

```
 /* x < y */
if (x < −2)
{      y = x² ;
       x = −x ;
}
else   y = x + y + 3 ;
  /* y > x + 1 */
```

il suffit, d'après les propriétés de l'alternative, de montrer séparément deux choses :

1. /* $x < y$ et $x < -2$ */
   $y = x^2$; $x = -x$;
   /* $y > x + 1$ */
2. /* $x < y$ et $x \geq -2$ */
   $y = x + y + 3$;
   /* $y > x + 1$ */

Pour montrer la première partie, il faut appliquer les règles de l'affectation et de l'enchaînement. Raisonnant de *droite à gauche*, nous sommes ramenés à montrer [4] que :

/* $x < y$ et $x < -2$ */
$y = x^2$;
/* $y > -x + 1$ */

ce qui est vrai [5] puisque $x < -2 \Rightarrow x^2 > -x + 1$

Pour montrer la deuxième partie, il suffit de montrer que [6] :

/* $x < y$ et $x \geq -2$ */ $\Rightarrow$ /* $x + y + 3 > x + 1$ */

ce qui est vrai puisque :

/* $x < y$ et $x \geq -2$ */ $\Rightarrow$ /* $y > -2$ */

4. application des règles de l'affectation à $x = -x$ et $P =$ /* $y > x + 1$ */)
5. Pour éviter le calcul du discriminant voici une démonstration simple : $x < -2 \quad \rightarrow \quad 0 > x + 2 \quad x^2 > -2x$ par addition :
$x^2 > -2x + x + 2$ et donc $x^2 > -x + 2 > -x + 1$
6. substitution de $x + y + 3$ pour y dans /* $y > x + 1$ */)

Schémas conditionnels emboîtés et ventilation

Le schéma conditionnel permet de régler l'alternative ou choix d'un ensemble d'actions *parmi* 2 ensembles possibles.

Lorsque l'on doit réaliser le choix d'un ensemble d'actions *parmi* n ($n > 2$) on peut :

- soit utiliser les structures conditionnelles imbriquées mais la lisibilité de la définition algorithmique devient rapidement malaisée dès que n devient important.

```
if  (condition1)
{
action(s) 1
}
else
{
if (condition2)
     {
     action(s) 2 ;
     }
else
     {
     action(s) 3 ;
     }
}
```

- soit utiliser la ventilation qui généralise de manière beaucoup plus satisfaisante le choix de : 1 *parmi* n.

```
switch (expression)
{
    case constante1 : action(s) 1 ;
    case constante2 : action(s) 2 ;
        . . .
    default    : action(s) n ;
}
```

Exemple Soit à écrire un programme qui fournit le nombre de jours écoulés depuis le premier janvier jusqu'au début du mois. Cet algorithme sera utilisé dans le calendrier perpétuel page 26.

```
switch (mois)
    {
    case  1 : ecoules =   0 ; break ;
    case  2 : ecoules =  31 ; break ;
    case  3 : ecoules =  59 ; break ;
    case  4 : ecoules =  90 ; break ;
    case  5 : ecoules = 120 ; break ;
    case  6 : ecoules = 151 ; break ;
    case  7 : ecoules = 181 ; break ;
    case  8 : ecoules = 212 ; break ;
    case  9 : ecoules = 243 ; break ;
    case 10 : ecoules = 273 ; break ;
    case 11 : ecoules = 304 ; break ;
    case 12 : ecoules = 334 ; break ;
    }
```

Raymond Queneau a proposé, sous le titre *Un conte à votre façon*, le texte suivant (extrait de : *L'Oulipo*, Coll. Idées, Gallimard 1972).

UN CONTE À VOTRE FAÇON

Ce texte soumis à la 83e réunion de travail de l'Ouvroir de Littérature Potentielle, s'inspire de la présentation des instructions destinées aux ordinateurs ou bien encore de l'enseignement programmé. C'est une structure analogue à la littérature "en arbre" proposée par F. Lionnais à la 79e réunion.

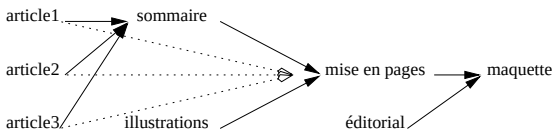
1. Désirez-vous connaître l'histoire des trois alertes petits pois ?
   si oui, passez à 4,
   si non, passez à 2.
2. Préférez-vous celle des trois minces grands échalas ?
   si oui, passez à 16,
   si non, passez à 3.
3. Préférez-vous celles des trois moyens médiocres arbustes ?
   si oui, passez à 17,
   si non, passez à 21.
4. Il y avait une fois trois petits pois vêtus de vert qui dormaient gentiment dans leur cosse. Leur visage bien rond respirait par les trous de leurs narines et l'on entendait leur ronflement doux et harmonieux.
   si vous préférez une autre description, passez en 9
   si celle-ci vous convient, passez à 5.
5. Ils ne rêvaient pas. Ces petits êtres en effet ne rêvent jamais.
   si vous préférez qu'ils rêvent, passez à 6,
   sinon, passez à 7.
6. Ils rêvaient. Ces êtres en effet rêvent toujours et leurs nuits sécrètent des songes charmants.
   si vous désirez connaître ces songes, passez à 11.
7. Leurs pieds mignons trempaient dans de chaudes chaussettes et ils portaient au lit des gants de velours noirs.
   si vous préférez des gants d'une autre couleur, passez en 8,
   si cette couleur vous convient, passez en 10
8. Ils portaient au lit des gants de velours bleu.
   si vous préférez des gants d'une autre couleur, passez en 7,
   si cette couleur vous convient, passez en 10.
9. Il y avait une fois trois petits pois qui roulaient leur bosse sur les grands chemins. Le soir venu, fatigués et las, ils s'endormirent très rapidement.
   si vous désirez connaître la suite, passez à 5
   sinon, passez à 21.
10. Tous les trois faisaient le même rêve, ils s'aimaient en effet tendrement et, en bons fiers trumeaux, songeaient toujours semblablement.
    si vous désirez connaître leur rêve, passez à 11,
    si non, passez à 12.
11. Ils rêvaient qu'ils allaient chercher leur soupe à la cantine populaire et qu'en ouvrant leur gamelle ils découvraient que c'était de la soupe d'ers. D'horreur, il s'éveillent.
    si vous voulez savoir pourquoi il s'éveillent d'horreur, consultez le Larousse au mot "ers" et n'en parlons plus.
    si vous jugez inutile d'approfondir la question, passez à 12.
12. Opopoï! s'écrient-ils en ouvrant les yeux. Opopoï! Quel songe avons-nous enfanté là ! Mauvais présage, dit le premier. Oui-da, dit le second, c'est bien vrai, me voilà triste. Ne vous troublez pas ainsi, dit le troisième qui était le plus futé, il ne s'agit pas de s'émouvoir, mais de comprendre, bref, je m'en vais vous analyser ça.
    si vous désirez connaître tout de suite l'interprétation de ce songe, passez à 15,
    si vous souhaitez au contraire connaître les réactions des deux autres, passez à 13.
13. Tu nous la bailles belle, dit le premier. Depuis quand sais-tu analyser les songes ? Oui, depuis quand ? Ajouta le second.
    si vous désirez aussi savoir depuis quand, passez à 14,
    si non, passez à 14 tout de même, car vous ne le saurez pas plus.
14. Depuis quand ? s'écria le troisième. Est-ce que je sais moi ! Le fait est que je pratique la chose. Vous allez voir !
    si vous voulez aussi voir, passez à 15,
    si non, passez également à 15, car vous ne verrez rien.
15. Eh bien ! Voyons, dirent ses frères. Votre ironie ne me plaît pas, répliqua l'autre, et vous ne saurez rien. D'ailleurs, au cours de cette conversation d'un ton assez vif, votre sentiment d'horreur ne s'est-il pas estompé ? effacé même ? Alors à quoi bon remuer le bourbier de votre inconscient de papilionacées ? Allons plutôt nous laver à la fontaine et saluer ce gai matin dans l'hygiène et la sainte euphorie ! Aussitôt dit, aussitôt fait : les voilà qui se glissent hors de leur cosse, se laissent doucement rouler sur le sol et puis au petit trot gagnent joyeusement le théâtre de leurs ablutions.
    si vous désirez savoir ce qui se passe sur le théâtre de leurs ablutions, passez à 16,
    si vous ne le désirez pas, vous passez à 21.
16. Trois grands échalas les regardaient faire.
    si les trois grands échalas vous déplaisent passez à 21,
    s'ils vous conviennent passez à 18.
17. Trois moyens médiocres arbustes les regardaient faire.
    si les trois moyens médiocres arbustes vous déplaisent passez à 21,
    s'ils vous conviennent passez à 18.
18. Se voyant ainsi zyeutés, les trois alertes petits pois qui étaient fort pudiques s'ensauvèrent.
    si vous désirez savoir ce qu'ils firent ensuite, passez à 19,
    si vous ne le désirez pas passez à 21.
19. Ils coururent bien fort pour regagner leur cosse et, refermant celle-ci derrière eux, s'y endormirent de nouveau.
    si vous désirez connaître la suite, passez à 20,
    si vous ne le désirez pas passez à 21.
20. Il n'y a pas de suite, le conte est terminé.
21. Dans ce cas, le conte est également terminé.

Raymond Queneau.

5. Dépendances et présuppositions

Makefile

La réalisation d'un projet présuppose la réalisation d'étapes préalables. Par exemple la confection d'un journal nécessite la rédaction des acticles qui le constituent. Il est possible de représenter cette dépendance par un graphe :



Chaque nouveau *numéro* suivra la même organisation.

On peut écrire un fichier de description (Makefile) de la confection du journal :

```
$ART= article1 article2 article3

maquette : mise_en_page editorial
    assembler pages editorial

editorial :
    rediger editorial

mise_en_page : sommaire illustrations $ART
    ajuster espaces sommaire illustrations $ART

sommaire : $ART
    lister titres ($ART)
```

Au cours de la réalisation d'une maquette, il peut arriver qu'un article soit réécrit. Il n'est pas pour autant nécessaire de refaire les autres articles ; par contre il est probable que le sommaire sera revu (si le titre de l'article est modifié par exemple) ainsi que la mise page ; par conséquent il faudra refaire la maquette.

Exemple de pensée par les présupposés

Au moyen de deux récipients dont les capacités respectives sont neuf litres et quatre litres, nous souhaitons disposer d'une quantité d'eau de six litres.

Représentons-nous clairement nos instruments de travail, c'est-à-dire, les deux récipients[1]. Imaginons qu'ils soient cylindriques, de bases égales et de hauteurs neuf et quatre (cf. Fig.5.1).

S'il y avait, sur la surface latérale de chacun d'eux, une graduation aux lignes horizontales également espacées, ce qui donnerait à tout moment la hauteur du niveau de l'eau, notre problème serait facile. Mais cette graduation n'existant pas, nous sommes encore loin de la solution.

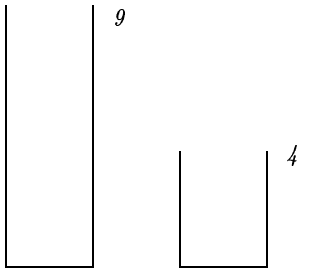


Fig.5.1

Nous ne savons pas encore comment mesurer exactement ; mais pourrions-nous mesurer une autre quantité ? Faisons des essais, tâtonnons un peu. Nous pouvons remplir complètement le plus grand ; si, avec son contenu, nous remplissons alors le petit, il nous reste cinq litres dans le grand. Pouvons-nous également en obtenir six? Vidons à nouveau les deux récipients. Nous pourrions aussi...

Nous agissons ainsi comme la plupart des gens à qui l'on pose ce problème. Partant de deux récipients vides, nous faisons un essai, puis un autre, les vidant et les remplissant à tour de rôle, et, après chaque échec, nous recommençons et cherchons autre chose. En somme, nous progressons, en partant de la situation donnée au début, vers la situation finale désirée, c'est à dire en allant du connu vers l'inconnu. Il se peut qu'après maintes tentatives nous finissions par réussir mais ce sera par hasard.

Que nous demande-t-on ? Représentons-nous le plus distinctement possible la situation finale que nous cherchons à atteindre. Imaginons que nous avons là, devant nous, le grand récipient contenant exactement six litres et le petit vide, comme à la figure 5.2. *(Partons de ce qui est demandé et admettons que ce que l'on cherche est déjà trouvé).*

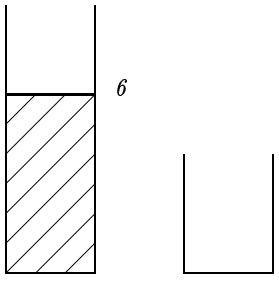


Fig.5.2

1. Proverbe numéro 1 *un petit dessin vaut mieux qu'un grand discours.*

À partir de quelle situation précédant immédiatement celle-ci pourrions-nous obtenir la situation finale désirée, comme à la figure 5.2? *(Cherchons à partir de quel antécédent le résultat final pourrait être obtenu).* Nous pourrions remplir complètement le grand récipient, donc, y verser neuf litres ; mais il faudra en retirer trois litres exactement. Pour cela . . . , il faudrait avoir déjà un litre dans le petit. Voilà l'idée ! (Cf. Fig. 5.3);

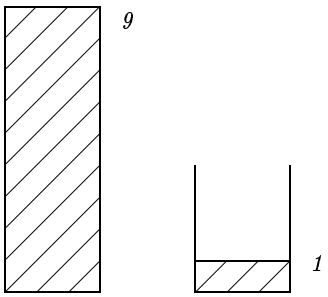


Fig.5.3

Mais comment atteindre la situation ainsi trouvée qu'illustre la figure 5.2.3? *(Cherchons à nouveau quel pourrait être l'antécédent de cet antécédent).* Étant donné qu'il est toujours possible de re-transvaser un récipient dans le récipient d'origine, la situation de la figure 5.3 est équivalente aux situations des figures 5.4 et 5.5.

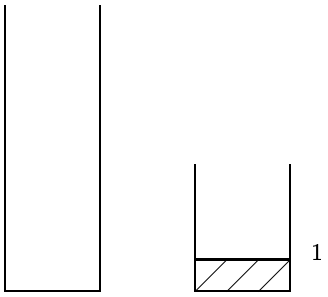


Fig.5.4

Il est facile de reconnaître que, si l'on obtient l'une quelconque des situations des figures 5.2, 5.3 et 5.4, on obtiendra aussi bien les deux autres ; mais il n'est pas si facile de tomber juste sur la situation de la figure 5.4, à moins de l'*avoir déjà rencontrée*, de l'avoir vue accidentellement, au cours d'une de nos précédentes tentatives. En multipliant les expériences avec les deux récipients, nous pouvons avoir réalisé quelque chose d'analogue et nous rappeler, au bon moment, que la situation de la figure 5.4 peut se présenter comme elle est suggérée à la figure suivante : en remplissant le grand

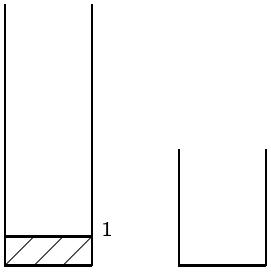


Fig.5.5

récipient, puis en vidant deux fois de suite quatre litres dans le petit et de là dans le récipient d'origine, nous rencontrons finalement quelque chose de déjà connu ; ainsi, par la méthode de l'analyse, par *raisonnement régressif* nous avons découvert la succession d'opérations appropriée.

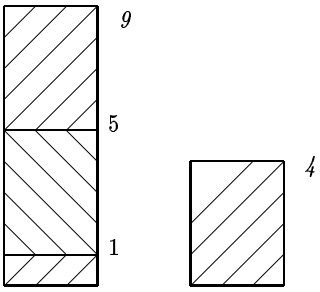


Fig.5.6

Il est vrai que cela s'est fait à rebours, mais nous n'avons plus qu'à *renverser le processus* en *partant du dernier point atteint dans notre analyse.* Nous faisons les opérations suggérée par la figure 5.5 et obtenons la figure 5.4, puis nous passons à la figure 5.3, de là à la figure 5.2 et finalement à la figure 5.1. *En revenant sur nos pas, nous arrivons finalement à trouver ce qui nous était demandé.* [2]

Il y a certainement dans cette méthode quelque chose d'assez profond. L'obligation de sinuer, de s'éloigner du but, en revenant en arrière, de ne pas prendre la route qui mène directement au résultat désiré entraîne certaines difficultés, dans le domaine de l'esprit. Pour découvrir la succession d'opérations appropriées, notre intellect doit suivre un ordre exactement à l'inverse de l'ordre réel. Il n'est nul besoin de génie pour résoudre un problème en revenant en arrière. Il suffit de se concentrer sur le but désiré, de se représenter la situation finale que l'on veut obtenir. À partir de quelle situation précédente pourrions-nous y parvenir? Il est essentiel de se poser cette question et, ce faisant, l'on revient en arrière.

2. C'est à Platon que la tradition grecque attribuait la découverte de la méthode d'analyse.

6.1 Schémas d'analyse

Fonctions

Un système peut être étudié de deux façons :

Aspect fonctionnel : Il s'agit de répondre à la question «à quoi ça sert ? ». On parle alors de fonction d'usage. (Le citoyen qui utilise un téléviseur le voit comme un objet permettant de véhiculer des informations)

Aspect structurel : Il s'agit de répondre à la question «comment ça marche ? ». On parle alors de fonction globale. (Le technicien voit le téléviseur comme un ensemble d'éléments transformant des ondes).

La méthode SADT (SADT : Structured - Analysis - Désign - Technique. - ©SADT est une marque déposée de SofTech (USA) et d'IGL Technologie (France).) développés aux USA par Doug Ross en 1977 et introduits en Europe à partir de 1982 par Michel Galiner.

Formalisme du modèle

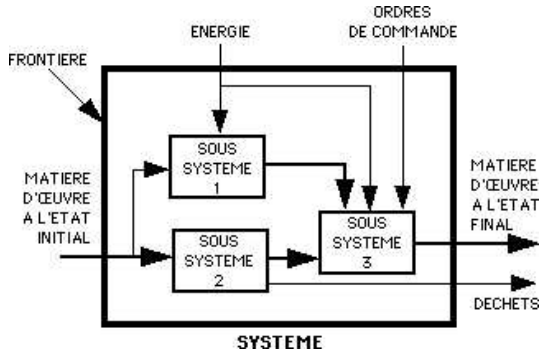


Fig.6.1.1 - Bloc fonctionnel SADT

On modélise graphiquement un système par un bloc fonctionnel (ou boîte) représenté par un rectangle à l'intérieur duquel est mentionnée la fonction globale (ou d'usage suivant les cas).

Les entrées sont de deux types :

- Les entrées de matière d'œuvre qui sont transformées par la fonction. Elles sont notées par des flêches entrantes à gauche.
- Les données de contrôle qui provoquent ou modifient la mise en œuvre de la fonction. Elles sont notées sur le dessus.

Les sorties représentent ce qui est produit par le système :

- La sortie de matière d'œuvre dotée de valeur ajoutée.
- Les sorties secondaires qui représentent généralement des flux d'informations associées au processus et des sous-produits ou déchets.

Rares sont les systèmes qui ne produisent pas de déchets (on emploiera le terme «pollution» lorsqu'ils ne sont pas traitables); et ne pas les prendre en compte risque de provoquer l'incapacité du système à assurer sa fonction.

Les supports de la fonction qui représentent les éléments matériels sont éventuellement notés sous le rectangle.

Description de la méthode

La première phase est la modélisation du système décrit précédemment (construction du modèle) qui en montre les fonctions. Le contexte est identifié par les flèches qui entrent ou sortent de cette boîte-mère.

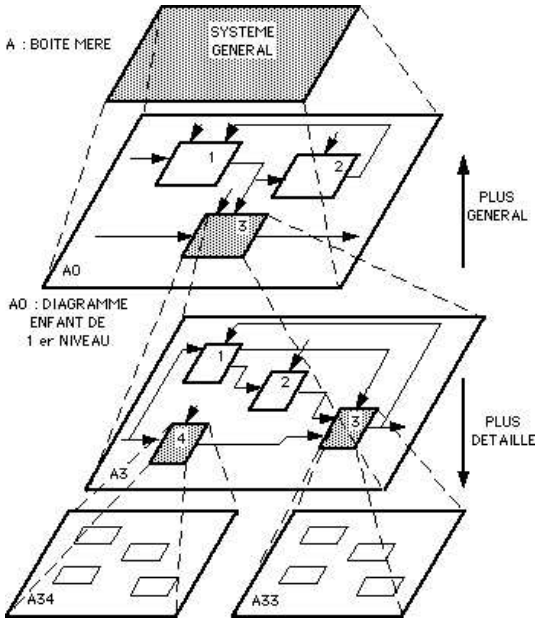


Fig.6.1.2 - Actigrammes SADT

La décomposition en éléments, ou sous-fonctions de cette boîte-mère permet d'affiner la perception du système et sa structure. Cette décomposition doit faire apparaître de trois à six éléments maximum. Ces éléments ou boîtes sont des activités. Les flèches qui les relient représentent les contraintes qui existent entre elles, mais ne représentent en au cun cas un flux de commande et n'ont pas de signification séquentielle (n'impliquent pas de notions d'ordre d'exécution dans le temps).

Les diagrammes ainsi construits sont des actigrammes ou encore diagrammes d'activité.

Si le niveau de décomposition ne permet pas une totale compréhension du système, on procède à une nouvelle construction d'actigrammes correspondant aux boîtes à analyser plus en détail.

On définit ainsi successivement :

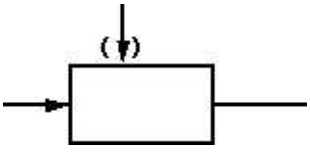
- La boîte-mère A.
- Le diagramme enfant de premier niveau A0.
- Les diagrammes enfants de chaque boîte du diagramme précédent (qui devient diagramme-mère) soit : A1, A2, A23, ...

Les principales règles régissant la construction des diagrammes sont :

- Chaque flêche entrant ou sortant de sa boîte-mère doit se retrouver sur le diagramme enfant.
- Les flêches sont affectées d'un label indiquant leur nature. Celui-ci peut être remplacé par un code dont la signification est donnée en marge.
- Les supports peuvent ne pas être mentionnés si cela n'éclaire pas la compréhension.
- On ne mentionne que les éléments nécessaires à ce que l'on veut montrer.
- Lorsque la relation est à double-sens (entrée réciproque ou contrôle réciproque), on utilisera une double flèche avec un point à droite ou sous la pointe des flèches concernées.

- Les flèches parenthésées, également appelées «flèches tunnel», indiquent qu'un flux de données est présent dans une partie du modèle bien qu'il ne soit pas dessiné. On trouve deux types de flèches tunnel :

  - La flèche tunnel dont les parenthèses entourent l'extrémité de la flèche qui est connectée à une boîte, qui signifie que cette flèche existe implicitement dans toutes les boîtes résultant de la décomposition de celle-ci.



  - La flèche tunnel dont les parenthèses se trouvent à l'autre extrémité, donc près des frontières du diagramme, qui signifie que cette flèche existe implicitement dans toutes les boîtes qui sont hiérarchiquement au dessus de la boîte concernée; c'est à dire sa boîte mère, grand-mère,... jusqu'à "A0" compris.

- Lorsque l'entrée est aussi une donnée de contrôle, on peut n'indiquer que la donnée de contrôle.

Exercice

Un *distributeur* (de sandwiches par exemple) accepte des pièces données (*1, 2, 5, 10*, pour fixer les idées), délivre des produits de différents prix et rend éventuellement la monnaie.

On se propose d'analyser le fonctionnement de cet appareil.

Décrire des *objets* nécessaires à la modélisation du distributeur.

...

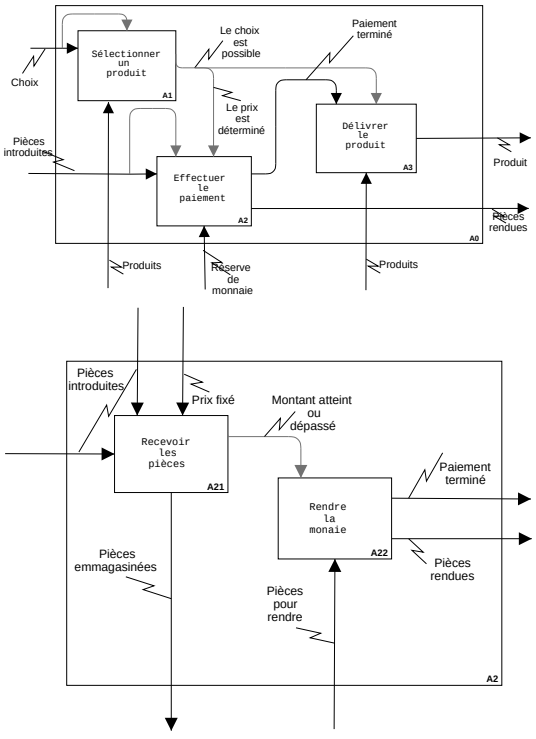


Fig.6.1.3 - Analyse SADT du distributeur de sandwiches

6.2 Grille d'analyse

La grille d'analyse intervient lors de l'étape de l'écriture de l'algorithme. Cette grille permet d'organiser l'expression de l'algorithme.

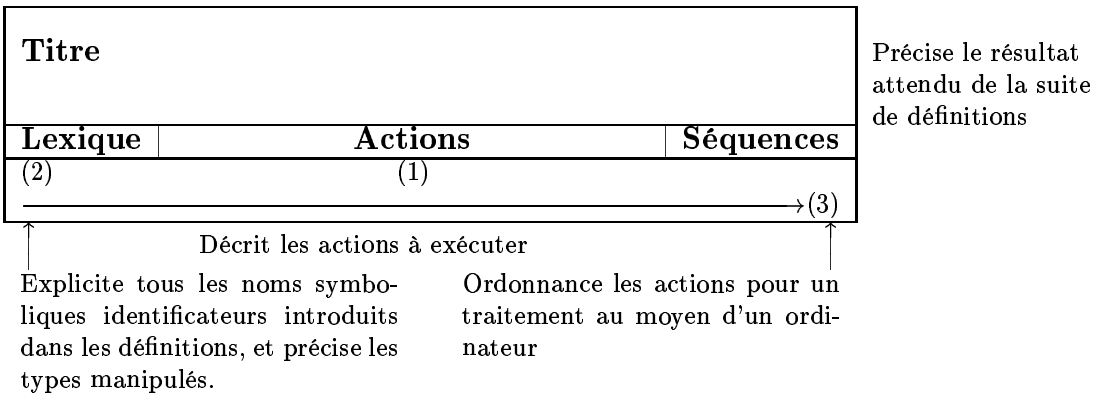


Fig. 6.2.1 – La grille d'analyse

La grille d'analyse est un tableau de 3 colonnes (fig. 6.2.1) réalisant le schéma dans lequel la conception de l'algorithme s'organise et se développe.

Dans la colonne centrale, **on commence par la dernière action**, ce qui fait apparaître une variable au moins, que l'on cherche à **expliciter**. Cette variable en **présuppose** d'autres que l'on explicite, et ainsi de suite jusqu'à ce que toutes les variables soient explicitées. Dans la colonne *Lexique*, on précise pour chaque variable son domaine de définition. On termine en fixant dans la colonne de droite l'ordre d'exécution pour le programme.

Ordre (1) $\longleftrightarrow$ (2) puis à la fin (3)

Exemple : Calcul du poids idéal d'une personne.

Le `POIDS` en Kg dépend de la `TAILLE` en cm et d'un coefficient :
`POIDS = ECART` $\times$ `COEF`
où `ECART` représente `TAILLE - 100`,

`coef` est fonction du sexe

`COEF =`	`1.1`	`SI SEXE = "masculin"`
	`1`	`SI SEXE = "féminin"`

Poids-idéal		
Lexique	**Définitions**	**Séquence**
	(1) Résultat = écrire POIDS	6
(2) POIDS (réel) : Kg	(3) POIDS = ECART $\times$ COEF	5
(4) ECART (entier)	(5) ECART = TAILLE - 100	4
(6) TAILLE (entier) : cm (4)	(7) TAILLE = donnée ('taille en cm $>$ 150')	2
(8) COEF (réel) : pondération fonction du sexe de la personne (10)	(9) **if** (SEXE == "masculin") COEF = 1.1 **else** COEF = 1	3
(10) SEXE (chaîne) : féminin ou masculin	(11) SEXE = donnée ("sexe? répondre par masculin ou féminin")	1

Les numéros entre parenthèses sont indiqués optionnellement pour montrer l'ordre de remplissage.

Fig. 6.2.2 – Calcul du poids idéal

Problème : Facture de pièces identiques

pièces_id		
Lexique	Définitions	Séquence
PHT (réel) prix hors taxe des pièces	résultat = écrire "prix hors taxe=" PHT à la ligne "prix toute taxe =" PTT	7
PTT (réel) prix toute taxe des pièces	PHT = PU × N	4
PU (réel) prix unitaire ht	PTT = PHT + TAXE	6
N (entier) nombre de pièces	PU = donnée ("prix unitaire")	3
TAXE (réel) TVA	N = donnée ("nombre de pièces")	2
TAUX (réel) taux de TVA	TAXE = TAUX × PHT	5
	TAUX = 0.186	1

Fig. 6.2.3 – Calcul d'une facture de pièces identiques

Contrôle de l'algorithme à l'aide du lexique
À chaque définition algorithmique de l'identificateur d'une donnée ou d'un résultat intermédiaire, on *coche* celui-ci dans le lexique (fig. ci-dessous). Ce lexique montre donc à *chaque instant* ce qui reste à définir.

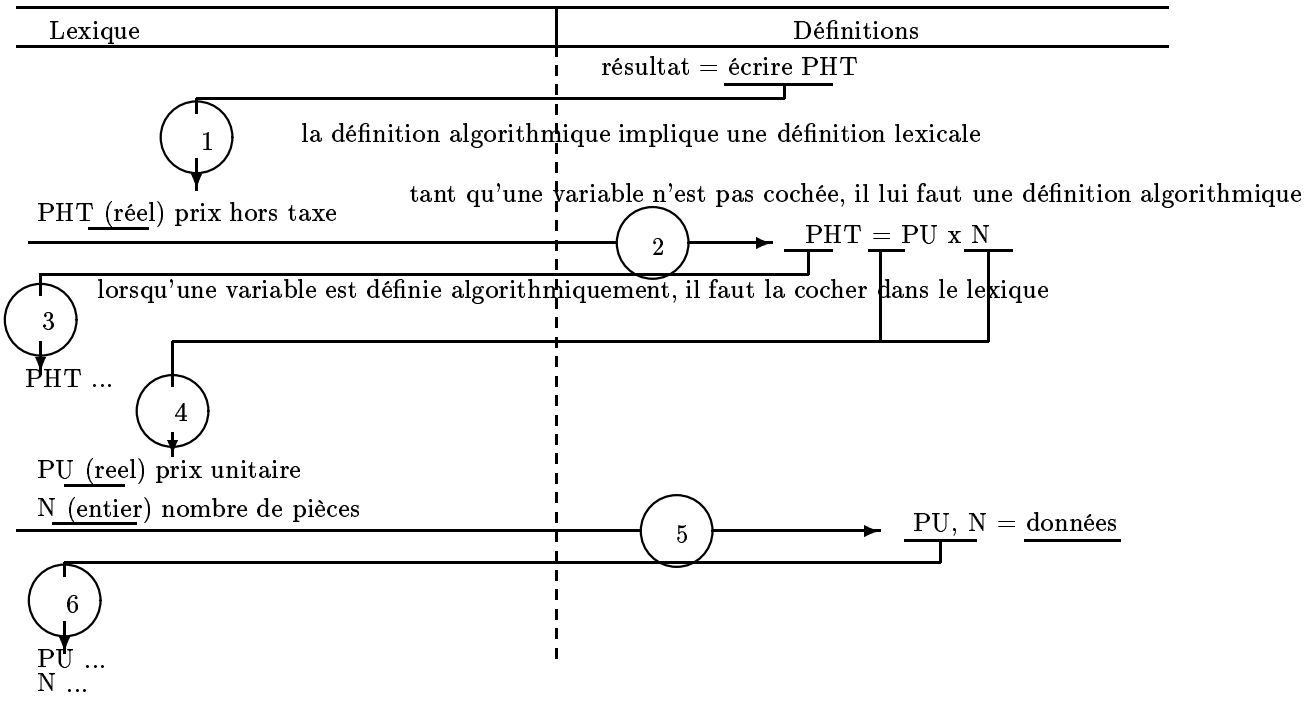


Fig. 6.2.4 – Contrôle de l'algorithme à l'aide du lexique

7. Itérations : les boucles

Propriétés de la répétition

À la «sortie» d'une boucle `tant que`, la condition (C) de boucle est toujours *fausse* ($\overline{C}$). Ceci correspond à l'utilisation naturelle de la boucle **while** : on veut obtenir, en un certain point du programme, la validité d'une affirmation $\overline{C}$. On connaît une action A (qui peut évidemment avoir une certaine complexité) dont on espère qu'elle *rapproche* l'état initial d'un état où $\overline{C}$ est vraie. On `répéte` A `tant que` C `est vraie`.

Si l'action A ne modifie pas la relation P, P est **invariant** pour la boucle `while (C) { A ; }`.

```
/* P et C */
while (C)
   {
    A ;
   }
/* P et C̄ */
```

Cette propriété est extrêmement importante. La notion d'invariant de boucle joue un rôle décisif dans la construction de programmes par des méthodes systématiques. En fait, on peut considérer qu'une boucle `tant que` est entièrement définie par sa condition d'arrêt **et** un invariant. Aussi souvent que possible, nous indiquerons en même temps qu'une boucle un invariant significatif associé.

Un invariant est souvent de la forme : «telle variable a telle valeur». À titre d'exemple, soit le programme ci-dessous, qui localise dans une liste l'élément x :

`/*` la variable "`trouve`" vaut `vrai` si et seulement si $L[p] = x$
et pour tout i compris entre `Premier(L)` et `Acceder(p-1, L)` $L[p] \neq x$ `*/`

```
Localiser(x, L)
{
    vrai = 1 ; NonTrouve = -1 ; trouve = faux ;
    p = Premier(L) ;
    /* TANT QUE p n'a pas parcouru toute la liste */
    while (p != Fin(L))
      {
      if (Identique ((Acceder(p, L), x)))
        trouve = vrai ; break ;
      else p = Suivant(p, L) ;
      }
    /* Si trouve == vrai alors renvoie p sinon renvoie NonTrouve */
    return  (trouve == vrai)? p:  NonTrouve ;
}
```

Fig.7.2 – Fonction de localisation d'un élément dans une liste

Cet invariant étant vrai initialement (puisque `trouve = faux`), il reste vrai au sortir de la boucle. Joint à la négation de la condition de bouclage, c'est-à-dire `/* p = Fin(L) ou trouve */`, il donne comme assertion finale :

```
/* trouve == faux et aucun élément de L ne vaut x,
ou bien trouve == vrai et x = L(p) */
```

ce qui est bien le but recherché.

Corriger un programme faux

Dans le numéro de janvier-février 1979 de l'*ordinateur individuel*, est paru ce programme de calcul de `x` puissance `n`, que nous étudierons réécrit en langage C :

```
01 void main()
02    {
03     int x, n, a, i = 0, t ;
04     scanf("%d", &x) ; scanf ("%d, &n) ;
05     a = x ;
06
07      do{
08        a = a * x ;
09        i = i + 1 ;
10        t = n - 1 ;
11        } while (i < t) ;
12    printf("%d\n", a) ;
13    }
```

La ligne (8) opère par multiplications répétées par x. Il est facile de voir qu'il est faux pour des valeurs simples :

$x = 2$ et $n = 1$

(5) a prend la valeur de x, donc $a = 2$.
(3) $i = 0$, (8) a est multiplié par x, $a = 2 \times 2 = 4$
(9) i est augmenté de 1 $i = 1$
(11) $i = 1$ n'est pas inférieur à t ($t = 0$), on passe donc en (12) et on affiche le résultat **4**, évidemment faux.

Fig.7.3 - Un programme faux

Examinons la boucle (6-11) : on arrive la $1^{ère}$ fois en (6)
en venant de (5) avec `/*` $a = x$ `et` $i = 0$ `*/`

Essayons l'assertion $a = x^{i+1}$ $\left\{ \begin{array}{llll} \texttt{(9)} & \texttt{/*}\ a = x^{i+1}\ \texttt{*/} & a = a \times x & \texttt{/*}\ a = x^{i+2}\ \texttt{*/} \\ \texttt{(10)} & \texttt{/*}\ a = x^{i+2}\ \texttt{*/} & i = i + 1 & \texttt{/*}\ a = x^{i+1}\ \texttt{*/} \end{array} \right.$

Remarquons que l'assertion $a = x^{x+1}$ est rétablie, avec une valeur de i plus grande. Mais la donnée introduite est n . Il faut donc situer i par rapport à n.

`(11) /*` $a = x^{i+1}$ `*/` **si** $i < t$ **alors** $a = x^{i+1}$ `et` $i < n - 1$ **boucler**
`(12) /*` $a = x^{i+1}$ `et` $i >= n - 1$ `*/`

Si, en (11), i devient égal à $n - 1$ alors $a = x^n$ et le programme est correct. Or par l'initialisation $a = x$ et $i = 0$.

Il faut donc avoir $0 < n - 1$ ou $n > 1$. Nous constatons que le programme calcule $a = x^n$ **si** et **si seulement** $n > 1$.

Il n'est pas très facile de corriger le programme. Il paraît probable que l'auteur a mal choisi son test d'arrêt, et qu'il a cherché à le corriger en introduisant la variable t calculée ligne (10) dont la valeur est constante. Son calcul dans la boucle est inatendu.

Il est plus important de considérer les situations que les actions. Pour **rédiger** le programme, il faut d'abord proposer une situation générale.

Supposons que l'on ait fait une partie du travail et calculé $a = x^i$ pour $i \leq n$.

`(7bis) /* on a calculé` $a = x^i$ `et` $i \leq n$ `*/`

Si $i = n$, c'est fini ; si non, il faut se rapprocher de la solution en faisant croître i `(8-9)` $a = a \times x$; $i = i + 1$; **boucler en(6)**

Il reste à voir le démarrage, c'est à dire trouver des valeurs de a et i telles que l'assertion soit vérifiée pour tout n positif ou nul. Il faut prendre $i = 0$ et donc $a = x^0 = 1$. D'où le nouveau programme correct, cette fois.

```
01 void main()
02  {
03    int x, n, a, i = 0, t ;
04    scanf("%d", &x); scanf ("%d, &n);
05    a = 1 ;
06    while (i < n)
07    {
07bis    /* a == x^i et i < n */
08       a = a * x ;
09       i = i + 1 ;
10
11     }
11bis  /* a == x^i et i == n */
12     printf("%d\n", a) ;
13   }
```

8.1 Fonctions et procédures

Dans la méthode de construction des algorithmes, nous partons de l'objectif final, le plus souvent une valeur à calculer ; ainsi, nous faisons apparaître une variable pour laquelle il faut expliciter le mode de calcul. Ce calcul **présuppose** lui-même le calcul d'autres variables. On comprend intuitivement que la valeur finale est calculée en *fonction* des variables dont elle dépend. Chaque calcul peut faire l'objet d'un algorithme et on peut utiliser, dans une définition d'un algorithme, une valeur résultant de l'exécution d'un autre algorithme.

Soit par exemple à fabriquer une série d'appareils dont chaque exemplaire sera équipé d'un quartz et de deux diviseurs de fréquence correspondant à la demande de chaque client. À la commande, le client précisera les 2 fréquences dont il a besoin. Nous allons écrire le programme qui détermine la valeur du quartz et les valeurs des deux diviseurs de fréquence, sachant que toutes ces valeurs sont des nombres entiers.

FREQ : /* Calcul d'une fréquence et de deux diviseurs */		
Lexique	Actions	ordre
Fq : Fréquence du quartz	AFFICHER(Fq, $div1$, $div2$) ;	5
F_1 et F_2 : Fréquences client	Fq = PPCM(F_1, F_2) ;	2
$div1$: diviseur pour F_1	$div1 = \frac{Fq}{F_1}$;	3
	$div2 = \frac{Fq}{F_2}$;	4
$div2$: diviseur pour F_2	SAISIR(F_1, F_2) ;	1

La valeur Fq sera renvoyée par le calcul du PPCM de F_1etF_2.

PPCM(a, b) /* Calcul du Plus Petit Commun Multiple */		
	PPCM = $\frac{a \times b}{\text{PGCD}(a,b)}$;	

La valeur du PPCM sera renvoyée par le calcul du PGCD[1] de F_1etF_2.

PGCD(x, y) /* Calcul du Plus Grand Commun Diviseur */		

m : variable temporaire	**while** (x != 0)
	{
	  if (x <y) y -= x ;
	  else x -= y ;
	}
	return y ;	

Fig.8.1.1 Calcul de la fréquence d'un quartz et de 2 diviseurs

On voit donc que l'on est conduit à écrire 2 nouvelles fonctions (qui ne sont pas natives) le PPCM et le PGCD pour réaliser ce programme.

Relation de précédente On définit une relation de précédente sur les variables d'un algorithme. Nous dirons que la variable x **précède** la variable y **si** x a au moins **une occurrence dans la définition** de y. À cause de cela, la valeur de y dépend de la valeur de x, et le calcul de y n'est possible que si x a déjà été calculé. La relation «x précède y» peut donc être lue comme «le calcul de x doit précéder le calcul de y».

Cette relation est un ordre partiel sur les variables de l'algorithme.

Un calendrier perpétuel

Définition du problème : On se propose de déterminer le jour de la semaine d'une date donnée. Sachant que le 1^{er} janvier 1900 était un lundi ; on appellera `Jour` la position d'un jour dans la semaine tel que `Jour` $\in$ [0-6], $0 \equiv$ dimanche.

Pour cela, on va chercher le **`décalage`** entre le 1^{er} janvier de l'année 1900 et la date considérée. On se donne une table du nombre de jours écoulés depuis le début de l'année jusqu'au début du mois `mois`.

`Calper(quantième, mois, an)`		
	`printf ("%d\n", Jour);`	5
`Jour` $\in$ `[0-6]`	`Jour = décalage` **`modulo`** `7 ;`	4
`décalage` $\in \mathbb{N}$	`décalage = quantième + DecalAA + DecalMM ;`	3
`DecalAA` $\in \mathbb{N}$	`DecalAA = EcoulesAn (année) ;`	1
`DecalMM` $\in \mathbb{N}$	`DecalMM = EcoulesMM (année, mois) ;`	2
`EcoulesAn(a)`		
	`/* 1 jour de décalage par an + 1 par année bissextile */`	
	`return` `DecAn ;`	4
`DecAn` $\in \mathbb{N}$	`DecAn = AnsApres1900 + AnsBissextiles ;`	3
`AnsApres1900` $\in \mathbb{N}$	`AnsApres1900 = a` **`modulo`** `100 ;`	1
	`/* On ne compte pas l'année courante parmi bissextiles */`	
`AnsBissextiles` $\in \mathbb{N}$	`AnsBissextiles = (a - 1) / 4 ;`	2
`EcoulesMM(a, m)`		
	`/* Décalage de jours entre 1.1.a et 1.m.a */`	
	`return` `DecMois ;`	5
`DecMois` $\in \mathbb{N}$	`DecMois = SommeMois[m] + Biss ;`	4
`Biss` $\in [0, 1]$	`Biss = 0 ;`	2
	`if (a%4)`	3
	`if (m>2)`	
	`Biss = 1 ;`	
`SommeMois` $\in \mathbb{N}$	`SommeMois[] = {0, 0, 31, 59, 90, 120, 151,`	
	`181, 212, 243, 273, 304, 334} ;`	1

Fig.8.1.2 Calendrier perpétuel

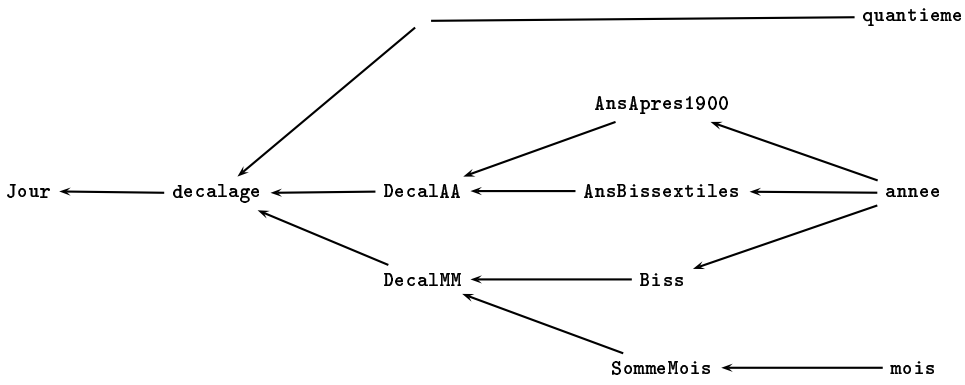


Fig.8.1.3 – Graphe de dépendance des variables

On vérifie sur le graphe de *dépendance des variables* de la figure ci-dessus que l'objectif (`Jour`) présuppose le calcul des variables dont il dépend, et que chacune d'elles est explicitée, soit par un calcul simple, soit par l'**appel d'une fonction**, jusqu'à remonter aux valeurs fournies à l'algorithme.

8.2 Récursivité

La récursivité

Reprenons le calcul de x^n. On suppose que l'on a fait une partie du travail, disons «on a calculé x^i». C'est fini si $i = n$. Sinon, en multipliant le résultat déjà calculé par x, on obtient x^{i+1}. Par changement de i en $i+1$, on se ramène à l'hypothèse de départ. Pour démarrer, on peut prendre $i = 1$ et $x^i = x$.

La récurrence joue donc de la façon suivante :

- si j'ai pu calculer x^i, je peux calculer x^{i+1}
- or je peux calculer x^1.

Sans rien changer à cette forme de raisonnement, nous pouvons construire un algorithme «récursif».

```
expr(x, n)
  {
  if (n == 1) return x ;
  else return x * expr(x, n-1) ;
  }
```

Pour écrire la définition récursive de $\texttt{exp}(x, n)$, nous avons utilisé la propriété bien connue :

$$x^n = x \times x^{n-1}$$

Construction

Reprenons le calcul de x^n :

$$\texttt{exp}(x, n) = \underbrace{x \times x \times x \times ... \times x}_{(nx \text{ dans cette formule})}$$

Encadrons les $n-1$ x de droite :

$$\texttt{exp}(x, n) = x \times \boxed{x \times x \times ... \times x}$$

($n-1$ x dans la boîte)

On retrouve la relation connue

$$\texttt{exp}(x, n) = x \times \texttt{exp}(x, n-1)$$

Comme cas singulier, on peut prendre $\texttt{exp}(x, 1) = x$ ou $\texttt{exp}(x, 0) = 1$ (qui a l'avantage d'étendre la définition au cas $n = 0$)

```
exp(x, n)
    {
    if (n == 0)
        return 1 ;
    return x * exp(x, n-1) ;
    }
```

Fig.8.2.1 $x^n = x * x^{n-1}$

On peut grouper autrement les x dans $\texttt{exp}(x, n)$. Supposons n pair. On peut partager les x en deux paquets égaux :

$$\texttt{exp}(x, n) = \boxed{x \times x \times \ldots \times x} \times \boxed{x \times x \times \ldots \times x}$$

avec $\frac{n}{2}$ x dans chaque paquet.

Dans ce cas

$$\texttt{exp}(x, n) = \texttt{exp}(x, \tfrac{n}{2})^2$$

Si maintenant n est impair, nous pouvons faire de même, en mettant à part le premier x

$$\texttt{exp}(x, n) = x \times \boxed{x \times x \times \ldots \times x} \times \boxed{x \times \ldots \times x}$$

On obtient ainsi une deuxième définition :

```
exp2(x, n)
    {
    if (n == 0)
        return 1 ;

    y = expr2(x, n/2) ;
    if (pair(n))
        return y * y ;
    return y * y * x ;
    }
```

Fig.8.2.2 $x^n \simeq x^{n/2} * x^{n/2}$

Nous pouvons faire, quand n est pair, les groupements d'une autre façon, en enfermant les paires de x dans des boîtes.

$$\texttt{exp}(x, n) = \boxed{x \times x} \times \ldots \times \boxed{x \times x}$$

Il y a $\frac{n}{2}$ boîtes ayant chacune deux x

```
exp3(x, n)
    {
    if (!n) return 0 ;
    if (pair(n))
        return exp3(x*x, n/2) ;
    return x * exp3(x*x, n/2) ;
    }
```

Fig.8.2.3 $x^n \simeq x^2 * x^2 \ldots$

Le pgcd d'Euclide

Euclide (300 av. J.-C.) a donné un algorithme pour trouver le PGCD[1] de 2 **nombres entiers positifs**:

```
PGCD (a, b)
    if (b == 0) return (a) ;
    else        return (PGCD(b, a % b)) ;
```

Fig.8.2.4 Calcul récursif du pgcd

ou son équivalent sous forme d'une boucle

```
1 PGCD (a, b)
2     while (b != 0)
      {
3         t = a ;
4         a = b ;
5         b = t % b ;
      }
6 return a ;
```

Fig.8.2.5 Calcul itératif du pgcd

Preuve mathématique

1. Si $d = \text{PGCD}(a, b)$ alors d divise a et b
2. $(a \ \% \ b) = a - q.b$ (q = reste de la division de a par b)
   $\longrightarrow$ $(a \ \% \ b)$ est une combinaison linéaire de a et b
   comme d divise b et a, et que $(a \ \% \ b)$ est une combinaison linéaire de a et b,
   alors d divise $(a \ \% \ b)$, cela permet de dire que d divise $\text{PGCD}(b, a \ \% \ b)$.

   Cela implique que:

   d qui est $\text{PGCD}(a, b)$ divise $\text{PGCD}(b, a \ \% \ b)$
   donc, $\text{PGCD}(a, b)$ divise $\text{PGCD}(b, a \ \% \ b)$

   idem pour $\text{PGCD}(b, a \ \% \ b)$ divise $\text{PGCD}(a, b)$
   donc: $\text{PGCD}(a, b) = \text{PGCD}(b, a \ \% \ b)$

Preuve de l'algorithme

Lignes	a	b	t
1	30	21	–
3	30	21	30
4-5	21	9	30
3	21	9	21
4-5	9	3	21
3	9	3	9
4-5	**3**	0	9
	PGCD ↑	↑	
condition d'arrêt			

Trace de l'algorithme pour 30 et 21 b ne peut jamais devenir négatif et il est toujours diminué de la valeur $(a \ \% \ b)$ qui est $< b$.

Modulo est une fonction qui

- ramène toujours une valeur inférieure d'au moins 1 par rapport à l'opérande droit.
  Ex: $3 \ \% \ 2 = 1$, $1 < 2$,
- possède aussi comme caractéristique de donner 0 si b vaut 1, car $\forall x, x/1 = x$,
  donc $(x \ \% \ 1) = 0$,
- renvoie a, si $a < b$.

Donc, b diminue de au moins 1 à chaque appel, et à la fin il doit valoir 0, donc la condition de sortie de la fonction arrivera **toujours**, quelles que soient les valeurs a et b données en entrées entières positives. Cette première validation est extrêmement importante, il est fondamental qu'un programme récursif, une boucle, ait une condition d'arrêt.

1. Que nous avons utilisé page 25.

9. Temps et espace d'exécution

Le temps d'exécution et la place mémoire requise caractérisent la complexité[1] d'un programme. Sous UNIX, on mesure le temps d'exécution d'un programme au moyen de la commande `time`.

```
:; time wc session.log
31 136 1044 session.log

real    0m0.08s
user    0m0.01s
sys     0m0.02s
```

Il tombe sous le sens qu'un programme mettra d'autant plus de temps à s'achever qu'il aura de données à traiter. Pour un même résultat, deux algorithmes équivalents ne verront sans doute pas leur temps d'exécution croître dans les mêmes proportions en fonction des données à traiter. Exemple : Calculs du plus grand diviseur de n.

```
i = n - 1 ;
while (0 != (n % i))
  /* pgd ≤ i const. de boucle */
  i = i - 1 ;
pgd = i ;
```

1ère méthode

```
i = 2 ;
while (i < √n et 0 != (n % i))
   /* ppd > i constante de boucle */
   i = i + 1 ;
/* pgd = SI (0 == n modulo i) ALORS n/i SINON 1
*/
pgd = (0 == (n % i))? n/i:  1 ;
```

2ème méthode

Fig.9.1 et 2 Deux méthodes de calcul du pgd

- **Complexité en temps**
  La complexité d'un algorithme se mesure essentiellement en calculant le nombre d'opérations élémentaires pour traiter une donnée de taille n. Les opérations élémentaires considérées sont
  - le nombre de comparaisons (algorithmes de recherche)
  - le nombre d'affectations (algorithmes de tri)
  - Le nombre d'opérations ($+$, $*$) réalisées par l'algorithme (calculs sur les matrices ou les polynômes).

  Le coût d'un algorithme A pour une donnée D est le nombre d'opérations élémentaires nécessaires au traitement de la donnée D et est noté $0(D)$
- **Complexité dans le pire des cas**, exemple : recherche d'un nombre dans un tableau, alors qu'il n'y est pas :

$$\mathrm{Max}_{(n)} = \max\{O(d_1), d_i \in D_i\}$$

- **Complexité dans le meilleur des cas**, ex : rechercher d'un nombre dans un tableau, alors qu'il est en première position :

$$\mathrm{Min}_{(n)} = \min\{O(d_1), d_i \in D_i\}$$

- **Complexité en moyenne**

$$\mathrm{Moy}(n) = \sum_{d \subset D_n} p(d)O(d)$$

où $p(d)$ est la probabilité d'avoir en entrée la donnée d parmi toutes les données de taille n.
Si toutes les données sont équiprobables, alors on a,

$$\mathrm{Moy} = \frac{1}{|D_i|} \sum_{d \in D_n} O(d)$$

- **Espace mémoire**
  Il est quelquefois nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire (tableau auxiliaire de même taille que le tableau donné en entrée par exemple).

1. La *simplicité* d'un algorithme n'est donc pas le contraire de la complexité.

Calcul de complexité d'un algorithme : apparition d'un nombre dans un tableau.

Soit T un tableau de taille N contenant des nombres entiers de 1 à k. Soit a un entier entre 1 et k.

La fonction suivante renvoie 1 lorsque l'un des éléments du tableau est égal à a, et 0 sinon.

```
int Trouve (int T[], int n, int a)
{
  int i = 0 ;
  while (i < n)
    if (T[i] == a )
      return i ;
  /* i == n ; le tableau est parcouru */
  return 0 ;
}
```

Fig.9.3 – Fonction de recherche d'un élément dans un tableau

Cas le pire : N (le tableau ne contient pas a)
Cas le meilleur : 1 (le 1^{er} élém. du tableau est a)
Complexité moyenne : Si les nombres entiers de 1 à k apparaissent de manière équiprobable, on peut montrer que le coût moyen de l'algorithme est $N = k(1-(1-1/k))$.

Les cas où l'on peut explicitement calculer la complexité en moyenne sont rares.

Analyse asymptotique

- Analyse du temps d'un calcul d'un programme
  - Valeur approchée. Le temps de calcul d'un programme dépend trop de la vitesse de l'ordinateur et du compilateur utilisés. On peut donc calculer les performances d'un algorithme à facteur multiplicatif constant. Des programmes de complexités n, $2n$ ou $3n$ sont quasiment équivalents.
  - Règle des 80/20 : 80% du temps de calcul d'un programme est réalisé dans 20% du code. Inutile donc d'essayer de perdre trop de temps à optimiser les 80% qui ne prennent que 20% du temps. Autant se consacrer à ce qui est le plus pénalisant.
  - Exemple : comparaison d'un algorithme A de complexité $100n$ et d'un algorithme B de complexité $2n^2$ (cf Aho-Ullman)
- Notations θ et O :
  - Définitions. On dit que $f = \theta(g)$ lorsqu'il existe deux constantes c_1 et c_2 positives (f et g sont également supposées à valeurs positives) telle que, pour n assez grand

$$c_1.g(n) < f(n) < c_2.g(n)$$

  Remarquons que cette relation est réflexive.
  - On dit que $f = O(g)$ lorsqu'il existe une constante c positive telles que, pour n assez grand

$$f(n) < c.g(n)$$

  c'est dire que f est bornée par g à un facteur multiplicatif près.
  Cette relation n'est pas réflexive.
  - Propriétés.
  Un polynôme est de l'ordre de son degré. On distingue les fonctions linéaires (en $O(n)$), les fonctions quadratiques (en $O(n^2)$) et les fonctions cubiques (en $O(n^3)$).
  Les fonctions d'ordre exponentiel sont les fonctions en $O(a^n)$ ou $a > 1$.
  Les fonctions d'ordre logarithmique sont les fonctions en $O(\log(n))$ (Peu importe la base du logarithme).
  - Une classe intéressante d'algorithme est en $n\log(n)$. Comparaison de $n\log(n)$ et de n^2.
  - Comparaison des asymptotiques classiques.
    - Rappelons que $log(n)^i \ll n^k \ll a^n (f \ll g$ lorsque limite$(g/f) = 0)$.
    - Fractions rationnelles
    - Factorielle : formule de Srirling
    - Nombres de Fibonacci
- Calcul de complexité dans les structures de contrôle.
  - Les instructions élémentaires (affectations, comparaisons) sont soit en temps constant, soit en O(1).
  - Tests :
  `O(if A then B else C fi) = O(A)+max(O(B),O(C))`
  - Boucles
  `O(for i from 1 to n do Ai od) = somme(O(Ai))`
  Lorsque O(Ai) est constant à O(A), on a
  `O(for i from 1 to n do A od) = nO(A)`
  - Cas particuliers : boucles imbriquées
    - `O(for i from 1 to n do for j from 1 to n do A od)` $= n^2$ `. O(A)`
    - Le fait que la borne sup de la boucle intérieure soit i plutôt que n ne change rien :
    `O(for i from 1 to n do for j from 1 to i do A od od)` `= (1+2+...+n). O(A).`

Carrés magiques[1]

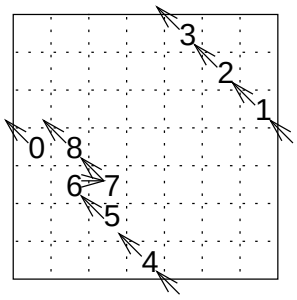


Fig.9.4 – Remplir un carré magique

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 (Fig.9.4).

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

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

Fig.9.5 – Un carré magique de 5×5

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:

```
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-- ;
  }
```

Fig.9.6 – Algorithme du carré magique

1. *in PPC* Dhénin Jean-Jacques

Jouer au «quinze vainc»

Le *quinze-vainc* se joue à deux. Il nécessite les neuf premières cartes (1.. 9) d'un jeu de 52 cartes. Seule la valeur compte, la couleur n'a pas d'importance. On étale les cartes entre les joueurs ; elles constituent le *talon* (Fig. 9.7.).

Fig.9.7 – *Le talon*

La partie se déroule en deux phases.

Première phase : à rour de rôle, chacun des deux joueurs prend une carte dans le talon et s'efforce de faire, le premier, exactement un total de 15 points avec 3 cartes... Attention de ne pas se laisser surprendre par l'adversaire ! (Fig. 9.8 9.9 et 9.10).
La première phase du jeu prend fin au moment où chacun est en possession de 3 cartes.

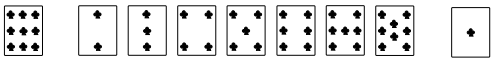


Fig.9.8 – *A prend 9, B prend 1 ;*

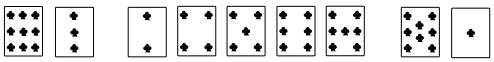


Fig.9.9 – *A prend 3, B prend 8 ;*

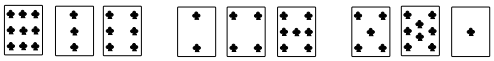


Fig.9.10 – *A prend 6, sinon B gagne ; B prend 5. Il reste au talon 2, 4, 7.*

Vous vérifierez aisément que, sauf erreur grossière de la part de l'un des deux joueurs, aucun d'eux ne sera parvenu à atteindre le total de 15 points. À ce moment commence la deuxième phase du jeu.

Deuxième phase : chacun conserve ses trois cartes ; et il reste également trois cartes au talon. C'est maintenant l'étape de l'échange. À son tour, chaque joueur échange obligatoirement une de ses cartes contre une du talon ; et il doit toujours faire quinze pour gagner. (Fig. 9.11 9.12 et 9.13).

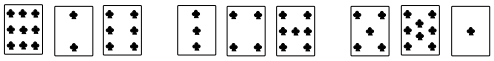


Fig.9.11 – *A échange 3 contre 2 ;*

Fig.9.12 – *B échange 8 contre 4, sinon A peut gagner (2, 9, 4) ;*

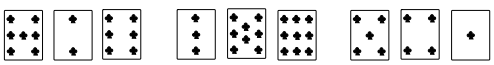


Fig.9.13 – *Mais A échange 9 contre 7 et gagne* $(6 + 7 + 2 = 15)$.

```
: # *-*-perl-*-
   eval 'exec perl -S $0 "$@"'
   if $running_under_some_shell ;

$HOMME = -1 ; $PROG  =  2 ; $LIBRE =  0 ;
@combinaison=([2, 4, 9], [3, 7, 5], [8, 6, 1], [6, 2,
7],
              [9, 1, 5], [4, 8, 3], [2, 8, 5], [4, 6,
5]) ;

sub afficher {
  $humain = $talon = $hp = "" ;
  for ($i = 1 ; $i < 10 ; $i++) {
     if ($jeu[$i] == $LIBRE ) { $talon = sprintf( "%s
%d", $talon, $i) ;  }
     if ($jeu[$i] == $HOMME ) { $humain = sprintf("%s
%d", $humain, $i) ; }
     if ($jeu[$i] == $PROG  ) { $hp = sprintf("%s %d",
$hp, $i) ;         }
  }
  printf ("%s : %s : %s\n", $humain, $talon, $hp) ;
}

sub carte {
  $carte = <> ; chomp $carte ; $carte ;
}

sub attribuer {
  $jeu[$_[0]] = $_[1] ;
}

sub peser {
  $somme = 0 ;
  $pipo[0] = $pipo[1] = 0 ;
  for ($j = 0 ; $j < 3 ; $j++) {
    $carte = $jeu[$combinaison[$_[0]][$j]] ;
    if ($carte == $LIBRE) { $pipo[0] = $combinaison[$_[0]][$j]
; }
    if ($carte == $PROG)  { $pipo[1] = $combinaison[$_[0]][$j]
; }
    $somme += $carte ;
  }
  $somme ;
}

# main
afficher ;
while (!$fini) {
  if (++$coups > 3) {
    $rendue = &carte  ;
    attribuer($rendue, $LIBRE) ;
    afficher ;
  }
  $prise = &carte ;
  attribuer($prise, $HOMME) ;
  $possible = $rendue = $obligatoire = 0 ;
  for ($i = 0 ; $i < 8 ; $i++) {
    $poids = &peser($i) ;
    if ($poids ==  0 && !$rendue)    { $rendue    =
$pipo[1] ; }
    if ($poids ==  1 || $poids == -1)  { $possible  =
$pipo[0] ; }
    if ($poids == -2 && !$fini)    { $obligatoire =
$pipo[0] ; }
    if ($poids == -3)          { $fini    = 1    ; }
    if ($poids ==  2 || $poids ==  3 ) { $rendue    =
$pipo[1] ; }
    if ($poids ==  4 &&  !$fini) {
      $obligatoire = $pipo[0] ;
      $fini = 2 ;
    }
  }

  if ($coups > 3)   { attribuer($rendue,      $LIBRE )
;}
  if ($obligatoire) { attribuer($obligatoire, $PROG  )
;}
  else              { attribuer($possible,    $PROG  )
;}

  afficher ;
}
if ($fini == 1) { printf("Vous avez gagne\n") ; }
if ($fini == 2) { printf("J'ai gagne\n")      ; }
```

10. Programmation linéaire

Position du problème

Un agriculteur peut utiliser 2 types d'engrais X et Y, pour fertiliser ses terres : il sait que celles-ci requièrent, par hectare et par an, au moins 60 kg de potasse, 120 kg de calcium et 90 kg de nitrates.

Les deux types d'engrais coûtent le même prix au poids ; pour 10 kg ils contiennent, en plus d'un produit neutre :

pour X : 1 kg de potasse, 3 kg de calcium et 3 kg de nitrates ;

pour Y : 2 kg de potasse, 2 kg de calcium et 1 kg de nitrates.

Comment l'agriculteur peut-il fertiliser ses cultures au moindre coût ?

Il s'agit d'un problème à 2 inconnues x et y (nombres positifs), représentant les quantités d'engrais de types X et Y à acheter par hectare et par an.

Les contraintes imposées sur la fertilisation d'un hectare peuvent s'écrire

$$\begin{cases} x + 2y & \geq & 60 \\ 3x + 2y & \geq & 120 \\ 3x + y & \geq & 90 \end{cases}$$

Le problème consiste à trouver x et y de façon à minimiser $x + y$ (le coût étant proportionnel à la quantité d'engrais achetée par hectare et par an).

D'une manière générale, un problème de programmation linéaire simple consiste à déterminer les valeurs numériques de m variables $x_1, x_2, ..., x_m$ satisfaisant n contraintes

$a_i^1 x_1 + a_i^2 x_2 + ... + a_i^m x_m \leq c_i \ (1 \leq i \leq n)$

(où les a_i^j sont des constantes, c'est-à-dire que $a_i^1 x_1 + a_i^2 x_2 + ... + a_i^m x_m$ est une fonction *linéaire* des variables) de façon à minimiser (ou à maximiser) une quantité $Z = z_1 x_1 + z_2 x_2 + ... + z_m x_m$ (donc encore une fonction linéaire des variables).

Interprétation géométrique

Dans un espace de dimension m, $a_i^1 x_1 + a_i^2 x_2 + ... + a_i^m x_m = c_i$ est une équation d'un **hyperplan** (espace de dimension $m - 1$: droite si $m = 1$, plan si $m = 2$ ...) séparant l'espace en deux demi-espaces caractérisés par les relations $a_i^1 x_1 + a_i^2 x_2 + ... + a_i^m x_m \geq c_i$ et $a_i^1 x_1 + a_i^2 x_2 + ... + a_i^m x_m < c_i$

L'ensemble des solutions $(x_1, x_2..., x_m)$ vérifiant les contraintes est ainsi l'ensemble des coordonnées des points d'un polyèdre convexe (s'il contient deux points, il contient le segment de droite qui les joint).

Les ensembles de points d'équations $z_1 x_1 + z_2 x_2 + ... + z_m x_m = Z$ sont des hyperplans tous parallèles à une certaine direction ; le problème consiste donc à trouver la valeur extréme de Z pour laquelle l'hyperplan d'équation $z_1 x_1 + z_2 x_2 + ... + z_m x_m = Z$ coupe le polyèdre des contraintes, et les solutions correspondant à cette valeur de Z, c'est-à-dire les points d'intersection de l'hyperplan et du polyèdre.

On peut montrer que, du fait que le polyèdre est convexe, cette intersection est une partie de la surface du polyèdre : soit un sommet, soit (cas dégénéré) une arête ou une face du polyèdre (en dimension $n = 3$; ce qui se généralise facilement à n quelconque).

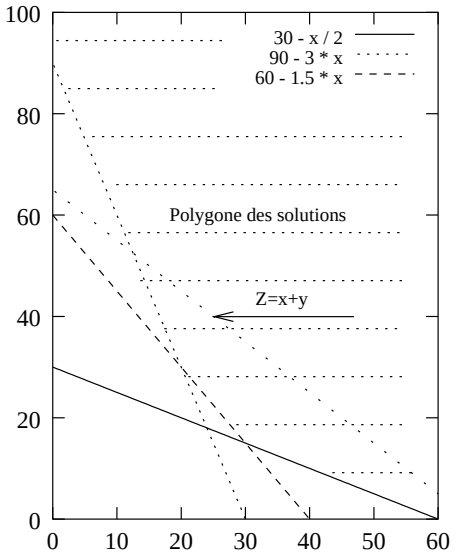


Fig.10.1 – *Interprétation géométrique*

Résolution graphique

Lorsque le problème ne comporte que 2 équations, il se prête à une résolution graphique : il suffit de chercher la position de la droite $Z = z_1 x_1 + z_2 x_2$ pour laquelle Z a une valeur extrêmale, et le sommet (ou l'arête) du polygone des solutions pour laquelle cette valeur est atteinte.

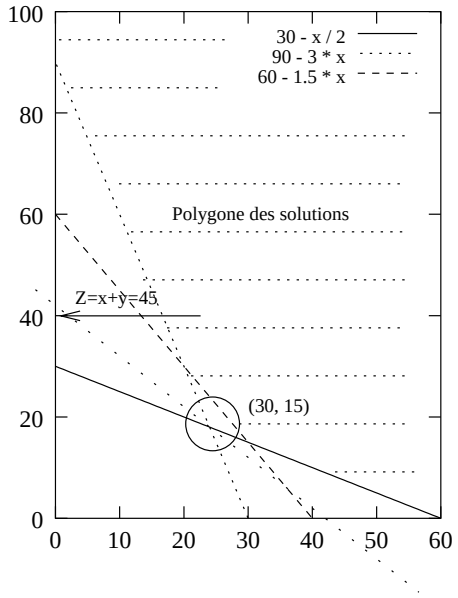


Fig.10.2 – Résolution graphique
d'un problème de programmation linéaire

Algorithme du simplexe

Le principe de l'algorithme du simplexe consiste à améliorer progressivement une solution de base non dégénérée (correspondant à l'un des sommets du polyèdre), en se déplaçcant le long des arêtes.

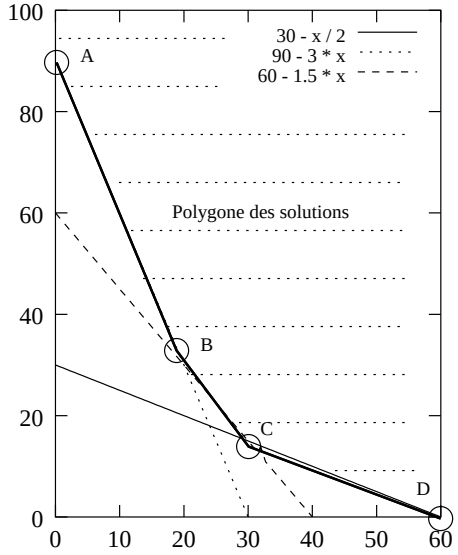


Fig.10.3 – Algorithme du simplexe

Dans l'exemple ci-dessus, on peut partir de la solution correspondant au point A c'est à dire à $x = 0, 3x+y = 90$. En posant $u = 3x+y-90$, on peut exprimer Z sous la forme $Z = u - 2x + 90$, et les contraintes sous la forme

$$\begin{cases} x, u & \geq & 0 \\ 5x - 2u & \leq & 120 \\ 3x - 2u & \leq & 60 \end{cases}$$

On peut faire diminuer Z en gardant u à zéro, et en augmentant x de 20 (quantité maximale respectant les contraintes) : on obtient ainsi la solution B, correspondant à $3x - 2u = 60$.

En posant $v = -3x + 2u + 60$, on peut exprimer Z sous la forme $-\frac{1}{3}u + \frac{2}{3}v + 50$, et les contraintes sous la forme $u, v \geq 0, \frac{4}{3}u - \frac{5}{3}v \leq 20$. On peut faire diminuer Z en gardant v à zéro, et en augmentant u de la valeur maximale respectant les contraintes, c'est-à-dire de 15 : on obtient ainsi la solution C, correspondant à $\frac{4}{3}u - \frac{5}{3}v = 20$.

En posant $w = -\frac{4}{3}u + \frac{5}{3}v + 20$, on peut exprimer Z sous la forme $\frac{1}{4}v + \frac{1}{4}w + 45$, et les contraintes sous la forme $v, w \geq 0$. On ne peut faire diminuer Z ; le point C correspond donc à la solution optimale.

D'une manière générale, soit un problème de programmation linéaire à m variables x_1, x_2, ..., x_m, que l'on suppose contraintes à vérifier $x_1, x_2, ..., x_m \geq 0$ (on peut toujours se ramener à ce cas), avec n inéquations de la forme

$$\sum_{j=1}^{j=m} a_i^j x_j \leq c_i$$

(pour $1 \leq i \leq n$), et une fonction économique à maximiser

$$Z = \sum_{j=1}^{j=m} z_j x_j + c$$

En introduisant n variables d'écart x_{m+1}, x_{m+2}, ..., x_{m+n} on peut remplacer chaque contrainte $\sum_{j=1}^{j=m} a_i^j x_j \leq c_i$ par l'équation $\sum_{j=1}^{j=m} a_i^j x_j + x_{m+i} = c_i$ et la contrainte $x_{m+i} \leq 0$.

Une solution **réalisable** du problème est la donnée de $m+n$ valeurs $x_1, x_2, ..., x_{m+n}$ satisfaisant les contraintes ; la solution est dite **de base** si au moins m de ces valeurs sont nulles (ce qui correspond à la surface du polyèdre) ; une solution de base est dite **non dégénérée** si exactement m des valeurs sont nulles (ce qui correspond aux sommets du polyèdre).

Les variables correspondant aux valeurs non nulles sont dites **variables de base**, les autres **variables hors base**.

L'algorithme du simplexe consiste à améliorer une solution de base non dégénérée en augmentant une et une seule des valeurs associées aux variables hors base (ce qui revient à faire entrer dans la base l'une de ces variables), et en mettant à zéro une variable de base (ce qui revient à la faire sortir de la base). On poursuit cette amélioration jusqu'à obtention de la solution optimale.

Méthode des tableaux

Dans la méthode dite «des tableaux réduits», on présente à la fois les données et contraintes du problème, et la solution en cours d'optimisation, sous forme d'un tableau

Variables de base	x_1	x_2	...	x_{m+n}	Valeurs
x_i	a_i^1	a_i^2	...	a_i^{m+n}	c_i
Z	z_1	z_2	...	z_{m+n}	c

dans lequel les lignes correspondent aux équations
$\sum_{j=1}^{j=m+n} a_i^j x_j + x_j = c_i$ la dernière ligne du tableau correspondant à

$$Z = \sum_{j=1}^{j=m+n} z_j + x_j + c$$

La solution à optimiser correspond aux valeurs $x_j = 0$ pour les variables hors base, $x_i = c_i$ pour les variables de base, et $Z = c$.

> Remarque: dans le cas fréquent où $x_1 = x_2 = ... = x_m = 0$ est une solution de base réalisable, le tableau associé correspond exactement aux données du problème: les variables de base sont les variables d'écart x_{m+i}, et les coefficients a_i^j, z_j, c_i et c sont ceux des équations définissant les x_{m+i} et Z).

Si x_i est une variable de base, alors on doit avoir $a_i^j = 1$, et $a^j i = 0$ pour toute variable de base $x_j \neq x_i$: c'est-à-dire que l'équation associée à x_i permet d'exprimer x_i en fonction des variables hors base : $x_i = -\sum_{\substack{x_j \\ \text{hors base}}} a_i^j x_j + c_i$

De même, si x_j est une variable de base, on doit avoir $z_j = 0$, de façon que la fonction économique s'exprime en fonction des variables hors base :

$$Z = -\sum_{\substack{x_j \\ \text{hors base}}} z_j x_j + c$$

Si tous les coefficients z_j sont négatifs, le maximum est atteint : en effet, du fait des contraintes $x_j \geq 0$, Z ne peut dépasser la valeur c.

Sinon, il est possible d'améliorer la solution en faisant entrer dans la base une variable hors base x_q (on choisira celle qui correspond au plus grand des coefficients z_j positifs).

On obtient une nouvelle solution de base en faisant sortir de la base l'un des x_p : pour que les contraintes soient respectées, on doit choisir p pour que le quotient $\frac{c_p}{a_p^q}$ soit minimal parmi les $\frac{c_i}{a_p^i}$ (tels que $a_p^q \neq 0$) et positif.

L'identité

$$x_p + a_p^q x_q + \sum_{\substack{x_j \\ \text{hors base}, j \neq q}} a_p^i x_j = c_p$$

permet d'exprimer x_q en fonction des nouvelles variables hors base :

$$x_q = -\frac{1}{a_p^q} x_p - \sum_{\substack{x_j \\ \text{hors base}, j \neq q}} \frac{a_p^j}{a_p^q} x_j + \frac{1}{a_p^q} c_p$$

En remplaçant x_q par cette expression dans les équations associées aux lignes du tableau, on obtiendra des équations correspondant à la nouvelle base $\sum_{\substack{x_j \\ \text{hors base}}} a_i^j x_j + x_i = c_i$

$$Z = \sum_{\substack{x_j \\ \text{hors base}}} z_j x_j + c$$

Dans la pratique, le tableau associé à la nouvelle solution s'obtient comme suit à partir du précédent :

- dans la ligne d'indice p, on remplace l'indice p par l'indice q, et on divise tous les coefficients par a_p^q ;
- pour toute ligne d'indice $i \neq p$, on remplace chaque coefficient a_i^j par $a_i^j - \frac{a_j^p}{a_p^q} a_i^q$; de même, on remplace c_i par $c_i - \frac{c_p}{a_p^q} a_i^q$;
- dans la dernière ligne, on remplace chaque z_j par $z_j - \frac{a_p^j}{a_p^q} z$, et la valeur c par $c + \frac{c_p}{a_p^q} z_q$.

Exemple

La résolution de l'exemple du cultivateur par la méthode des tableaux réduits commence par l'introduction de variables d'écart $u, v, w \geq 0$, avec

$$\begin{cases} x & + & 2y & -w & = & 60 \\ 3x & + & 2y & -v & = & 120 \\ 3x & + & y & -u & = & 90 \end{cases}$$

et la fonction économique $Z = -x - y$ à maximiser. La solution de base non dégénérée $x = 0, y = 90$ correspond à la base y, v, w; en remplaçant y par $-3x + u + 90$ on obtient

$$\begin{cases} x & + & 2y & -w & & = & 60 \\ 3x & + & 2y & -v & & = & 120 \\ 3x & + & y & -u & & = & 90 \\ Z & = & 2x & -u & -90 & & \end{cases}$$

ce qui correspond au tableau

	x	y	u	v	w		
w	5		-2		1	120	$\frac{120}{5} = 24$
v	3		-2	1		60	$\frac{60}{3} = 20$
y	3	1	-1			90	$\frac{90}{3} = 30$
Z	2		-1			-90	

On fait entrer x dans la base, et sortir v :

	x	y	u	v	w		
w			$\frac{4}{3}$	$\frac{-5}{3}$	1	120	$\frac{20}{\frac{4}{3}} = 15$
x	1		$\frac{-2}{3}$	$\frac{1}{3}$		20	$\frac{20}{\frac{-2}{3}}$ est négatif
y		1	1	-1		30	$\frac{30}{1} = 30$
Z			$\frac{1}{3}$	$\frac{-2}{3}$		-50	

On fait entrer u dans la base, et sortir w :

	x	y	u	v	w	
u			1	$\frac{(-5}{4}$	$\frac{3}{4}$	15
x	1		$\frac{-1}{2}$	$\frac{1}{2}$		30
y		1		$\frac{1}{4}$	$\frac{-3}{4}$	15
Z			$\frac{-1}{4}$	$\frac{-1}{4}$	-45	

Les coefficients z_j sont tous deux négatifs : le maximum est atteint. On retrouve ainsi la solution $x = 30, y = 15$.

Deuxième partie

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'est à dire 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. 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 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 :

1. Cf. page 29

```
#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.

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 aux types de données.

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'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"
/* Creation du type pointeur sur int */
typedef int  *entier ;
liste(entier) ;

/* fonctions pour le nouveau type */

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

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

2. Le programme complet est en annexe

```
  *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) ;
}
```

Fig.1.3 – Instanciation d'une liste d'entiers

2. Correspondance par tables

Définition

Une table de correspondance ou plus simplement correspondance, est une fonction d'un ensemble d'éléments de type TypeSource 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 d de type TypeCible par la relation $C(s) = d$.

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 :

vider(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',
'simon',   'tranche' );
```

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

```
$val = $caracteristiques{'xenon'};
# $val = 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)
  ```

3. Les structures

Rappels

- Un tableau est un agrégat d'éléments de même type tandis qu'une `structure` est un agrégat d'éléments de type (presque) arbitraire. Par exemple :

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

- Les types de données abstraits, en langage C font un grand usage de structures et notamment de la structure «MAILLON».
- 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 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 adresses mémoire 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 ci-dessous */

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 */
```

4. 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, accéder au $i$ème élément, en connaître le contenu, connaître la longueur de la liste (le nombre d'éléments qu'elle contient), supprimer un élément de la liste, y insérer un nouvel élément et enfin accéder au successeur d'un élément dont on connaît la place, vouloir fabriquer une liste à partir de deux autres.

La pile (LIFO) est une liste dans laquelle on introduit et on supprime des éléments à une et *une seule* extrémité ;

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

Il existe de nombreuses façons de mettre en œuvre les listes. La mise en œuvre d'une liste 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 4.1 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.

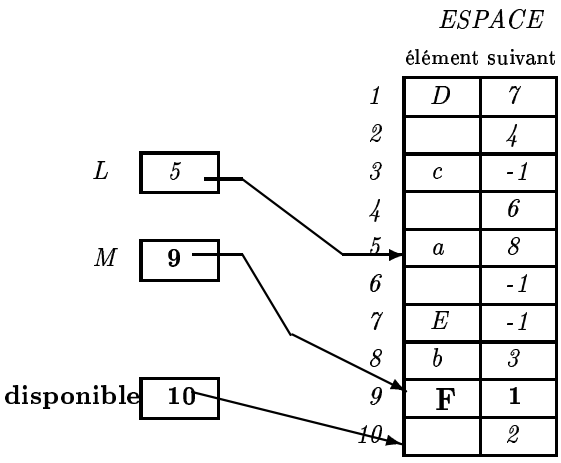


FIG. 4.1 – *Une liste chaînée dans un tableau*

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 55) ; en effet on considére 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 9)
4. Le langage 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 4.1, l'adresse en mémoire, lorsqu'il s'agit de *pointeurs*.
6. Cf. page 51.

Programmation d'une liste générique

Pour ne plus être limité par le tableau, comme dans la page précédente, on utilise une liste chaînée de structure.

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.

```
#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 ;
```

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 4.2. 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.

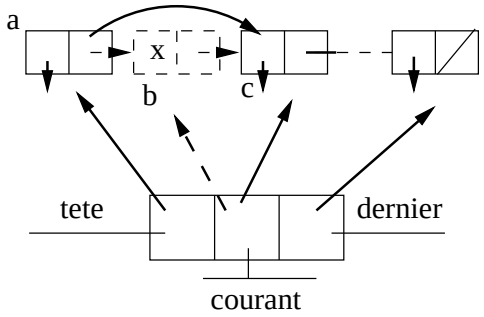


FIG. 4.2 – *Suppression de l'élément* `x`

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 */
      }
    }
  } /* 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 4.2)

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 ;
}
```

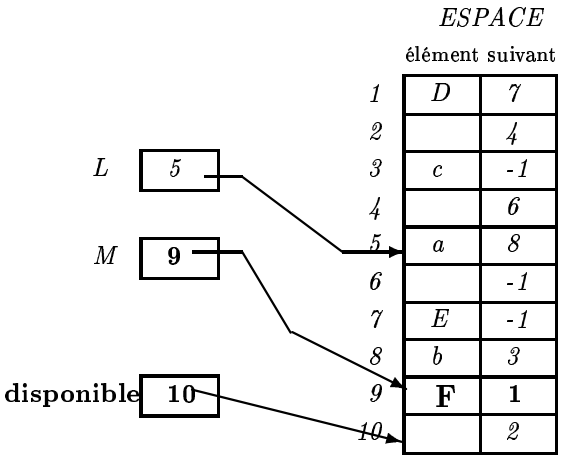


FIG. 4.3 – *Insertion en tête de* $\mathcal{M}$

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) */
}
```

5. Utilisation d'une pile pour le traitement des expressions

Généralités

Intuitivement, une pile (LIFO) 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 (d'où le nom de Last In First Out).

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 postfixée 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() ;
}
```

FIG. 5.1 – Utilisation d'une pile pour le traitement des expressions

6. Les files

Peut-être plus encore que les piles, les files d'attente (ou simplement files ou FIFO) 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 type de données abstrait 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 (d'où le nom de First In First Out).

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 (c'est, entre autres, l'objet du génie informatique).
- 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, ..., 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.

  unshift(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.

  shift(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.

Mise en œuvre d'une file dans un tableau

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 fig.6.1

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 `unshift`.

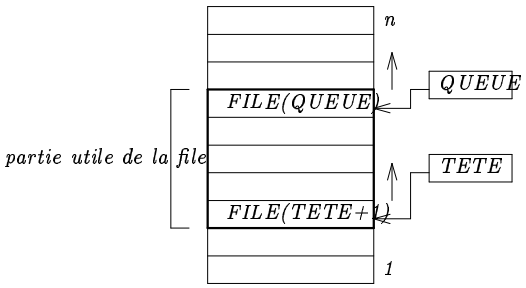


Fig.6.1 – Une file dans un tableau

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`.

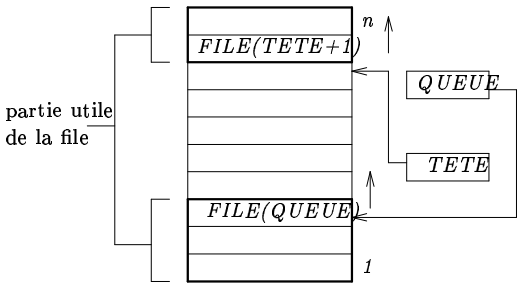


Fig.6.2 – Une file circulaire dans un tableau

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 œuvre, 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 possible. 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 fig.6.2

L'intérêt de cette méthode est qu'elle ne nécessite aucun décalage. On parle d'une représentation par *file circulaire*, on peut représenter la figure précédente 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`$| \geq 1$ 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. On utilise alors une liste chaînée et non plus un tableau. 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)) ;            \
}
```

7. Les 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.

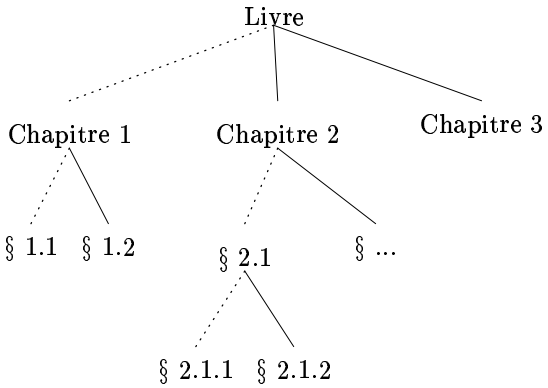


FIG. 7.1 – Une table des matières est un arbre

Étiquettes et expressions

L'arbre *étiqueté* de la figure 7.2 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 e_2 représente $(a) + (b)$, ou simplement $a + b$. Le nœud e_1 représente $(a+b)*(a+c)$, puisque $*$ est l'étiquette associée à e_1 et puisque $a+b$ et $a+c$ sont les expressions représentées par e_2 et e_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 trigonométrique, en en restant toujours le plus près possible.

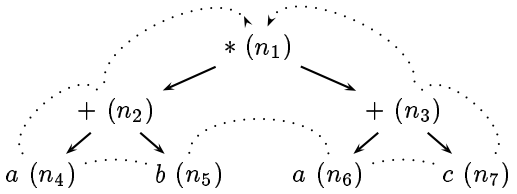


FIG. 7.2 – *L'arbre d'une expression*

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

1.– *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. 7.3 – *Exemple d'arborescence*

Dans l'exemple ci-dessus, 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
```

7.4 *Script-shell Types de fichiers*

La récursivité : parcours d'une arborescence

2.– *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 "Rep : $FICHIER
# ok meme apres un mv de prog2
         $0 $FICHIER
    else echo "Fic : $FICHIER
  fi
done
```

7.5 *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.

8. Les graphes

Exemples de problèmes formalisables par des graphes

Choix d'un itinéraire

Sachant que la gare de Poitiers est inutilisable, et connaissant la durée des trajets suivants :

Bordeaux	→	Nantes	4 h
Bordeaux	→	Marseille	9 h
Bordeaux	→	Lyon	12 h
Nantes	→	Paris-Mtparn.	2 h
Nantes	→	Lyon	7 h
Paris-Mtparn.	→	Paris-Lyon	1 h
Paris-Lyon	→	Grenoble	4 h 30
Marseille	→	Lyon	2 h 30
Marseille	→	Grenoble	4 h 30
Lyon	→	Grenoble	1 h 15

comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble ?

Les données du problème sont faciles à représenter par un graphe dont les arêtes sont étiquetées par les durées des trajets :

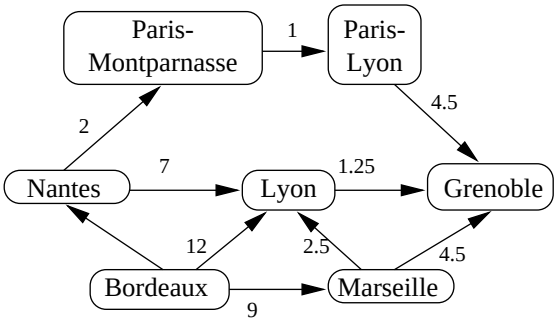
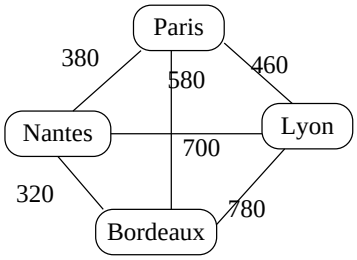


Fig. 8.1 – Choix d'un itinéraire

Le problème du voyageur de commerce

Un voyageur de commerce habitant Bordeaux doit effectuer une tournée passant par Lyon, Nantes et Paris :



Dans quel ordre devrait-il effectuer ses déplacements de façon à minimiser le trajet parcouru ?

C'est le problème classique du voyageur de commerce (TSP : Traveling Salesman Problem) : il s'agit de déterminer dans le graphe un circuit élémentaire passant par chaque sommet (circuit hamiltonien), de façon que la somme des distances d'un sommet à un autre soit minimale, les distances étant données par le tableau

	Bordeaux	Lyon	Nantes	Paris
Bordeaux		780	320	580
Lyon	780		700	460
Nantes	320	700		380
Paris	580	460	380	

Le problème pourrait être résolu en examinant tous les ordonnancements possibles des sommets au départ de Bordeaux ; si l'on part d'un graphe à n sommets, cela donne n! possibilités, ce qui conduit à un algorithme irréalisable, en pratique.

Planification de travaux

Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours) et de carreler (2 jours) la salle de bains, de refaire le parquet de la salle de séjour (6 jours) et de repeindre les chambres (3 jours), la peinture et le carrelage ne devant être faits qu'après réfection de l'installation électrique.

1. Si le propriétaire décide de tout faire lui-même, dans quel ordre doit-il procéder ?
2. Si la rénovation est faite par une entreprise et que chacune des tâches est accomplie par un employé différent, quelle est la durée minimale des travaux ?

1. On peut représenter les différentes tâches à effectuer par les sommets d'un graphe dont les arcs correspondent aux contraintes de précédence :

   Il s'agit de numéroter les sommets du graphe de façon à respecter les relations de précédence (les successeurs d'un sommet doivent avoir des numéros supérieurs à celui de ce sommet).
2. On peut représenter les différentes étapes de la rénovation sur un graphe dont les arcs sont étiquetés par la durée minimale séparant deux étapes :

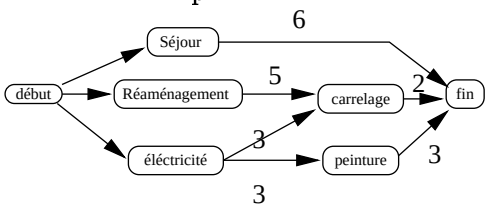


FIG. 8.2 – Planification de travaux

Il s'agit de déterminer la durée du plus long chemin du début à la fin des travaux.

Représentation des graphes en informatique

Structure associée à la représentation graphique

Un graphe peut être représenté par une liste de sommets, chacun étant caractérisé par son nom, une étiquette éventuelle, et la liste des arcs qui ont ce sommet pour origine (eux-mêmes caractérisés par leur but, et une étiquette éventuelle):

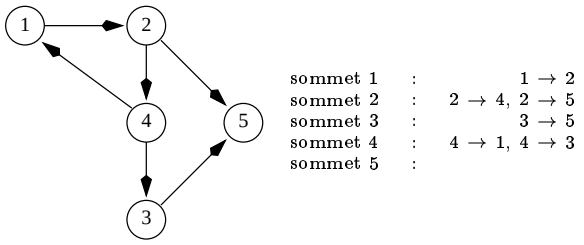


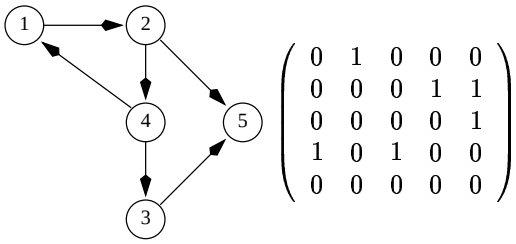
Fig.8.3 – Représentation par listes d'arcs

Avantages : simplicité de la mise à jour, facilité de parcours; Inconvénients : redondance de la représentation pour les graphes non orientés; temps d'accès aux données (adressage); le parcours ne s'effectue que dans le sens des arcs pour les graphes orientés. (Ce dernier inconvénient peut être supprimé en ajoutant, pour chaque sommet, la liste des arcs qui ont ce sommet pour but, au prix d'un encombrement de la mémoire).

Représentations matricielles

Matrice d'adjacence

Un graphe simple non étiqueté à n sommets numérotés peut être représenté par une matrice carrée $(n,\ n)$ d'entiers : l'élément $M_{[i,j]}$ vaut 1 s'il existe un arc allant du sommet i au sommet j, 0 sinon :



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

Fig.8.4 – Représentation par matrice d'adjacence

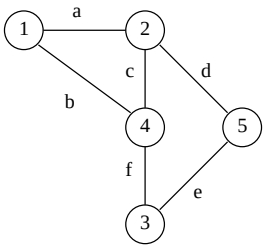
Dans le cas où les arcs sont étiquetés, la matrice sera constituée des étiquettes, à condition de pouvoir caractériser l'absence d'arc par une valeur particulière d'étiquette.

Avantages : rapidité des recherches, compacité de la représentation, simplicité des algorithmes de calcul.

Inconvénients : cette représentation ne convient qu'aux graphes simples ; redondance des informations pour les graphes non orientés ; stockage inutile de cas inintéressants (les zéros de la matrice), à examiner quand on parcourt le graphe (pour la complexité des algorithmes).

Matrice d'incidence

Un graphe non orienté à n sommets numérotés et p arcs numérotés peut être représenté par une matrice (n, p) d'entiers : l'élément $M_{[i,j]}$ vaut 1 si le sommet i est une extrémité de l'arc j, 0 sinon :



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

Fig.8.5 – Représentation par matrice d'incidence

Dans le cas où les arcs sont étiquetés, la matrice sera constituée des étiquettes, à condition de pouvoir caractériser l'absence d'arc par une valeur particulière d'étiquette.

Parcours des graphes en informatique

Parcours de graphes

Les algorithmes de parcours des arborescences s'adaptent aux graphes, en considérant que l'on parcourt chacune des arborescences associées aux sommets du graphe ; l'arborescence associée à un sommet est celle des descendants de ce sommet que l'on n'a pas encore visités.

Parcours en largeur

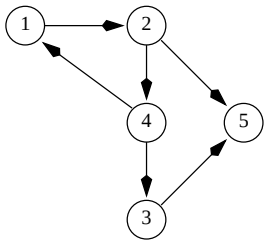


Fig.8.6 – Ordre de parcours : 1, 2, 4, 5, 3

Preuve du programme : chaque sommet est visité, donc enfilé au moins une fois ; or, on ne peut enfiler que des sommets non encore enfilés ; par conséquent, chaque sommet est enfilé exactement 1 fois.

```
parcourir un graphe en largeur :
 visiter tous les sommets

visiter un sommet s :
 SI s n'a pas encore ete enfile
  ALORS enfiler s ;
   TANT QUE la file nest pas vide
     soit n le premier sommet de la file ;
     {traitement de n}
     defiler ;
     enfiler tous les successeurs
     non encore enfiles de n
    FIN TANT QUE
 FIN SI
```

À chaque passage dans la boucle TANT QUE, on défile un sommet différent, donc chaque procédure `visiter` se termine. À la sortie de *visiter*, la file est vide ; donc chaque sommet aura été défilé, et par conséquent traité, exactement 1 fois.

Complexité : Chaque sommet est traité 1 fois, et chaque arc est examiné 1 fois (lors du traitement de son origine). La complexité est donc de l'ordre de max(|A|, |S|).

Parcours en profondeur (récursif)

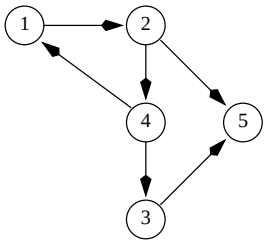


Fig.8.7 – Ordre de visite : 1, 2, 4, 3, 5

```
parcourir un graphe en profondeur :
 visiter tous les sommets

visiter un sommet s :
 SI s n'a pas encore ete visite
 ALORS
  {pre-traitement de s}
  visiter tous les successeurs de s
   {post-traitement de s}
FIN SI
```

Preuve du programme : chaque sommet est visité exactement 1 fois.

Complexité : chaque sommet est visité 1 fois, chaque arc d'origine est examiné 1 fois. La complexité est donc de l'ordre de max(|A|, |S|).

Plus court chemin dans un graphe

Algorithme de Floyd

On considère des graphes simples *valués*, c'est-à-dire dont chaque arc est muni d'un **poids** (nombre réel). Un chemin a pour poids la somme des poids des arcs qui le constituent.

Le problème des plus courts chemins consiste à déterminer le poids minimal d'un chemin d'un sommet à un autre, en supposant les poids positifs.

On représente un graphe orienté valué à n sommets par une matrice carrée (n, n) de nombres : $M_{[i,j]}$ a pour valeur le poids de l'arc $i \to j$ ($+\infty$ si cet arc n'existe pas).

Une variante de l'algorithme de Warshall permet de calculer la matrice D telle que $D_{[i,j]}$ soit le poids minimal d'un chemin de i à j :

Cet algorithme résout le problème des plus courtes distances de tout sommet du graphe à tout autre sommet, mais ne donne pas le chemin correspondant.

```
D = M ;
POUR i  1 a n
 POUR j de 1 a n
  SI D[j, i] < +inf
  ALORS
   POUR k allant de 1 a n
    SI D[j, k] > D[j, i] + D[i, k]
ALORS
     D[j, k] = D[j, i] + D[i, k]
FIN SI
   FIN POUR
  FIN SI
 FIN POUR
FIN POUR
```

Fig.8.8 – Floyd : Plus court chemin

Les algorithmes qui suivent permettent de déterminer les plus courts chemins d'un sommet particulier s du graphe à tout autre sommet. On notera $p(a \to b)$ le poids d'un arc $a \to b$.

Algorithme de Ford

```
POUR tout sommet t
d[s, t] = +inf
FINPOUR

d[s, s] = 0
REPETER
 POUR tout sommet t
 SI d[s, t] < +inf
 ALORS
  POUR tout successeur u de t
   SI d[s, u] > d[s, t] + p(t -> u)
   ALORS
    d[s, u] = d[s, t] + p(t -> u)
   FIN SI
  FIN POUR
FIN SI
FIN POUR

TANT QU'on modifie quelque chose
```

Fig.8.9 – Ford : Plus court chemin

Algorithme de Dijkstra

```
POUR tout sommet t
d[s, t] = +inf
FINPOUR
d[s, s] = 0

TANT QU'il reste des sommets non fixe's
 choisir un sommet t non fixe'
 tel que d[s, t] soit minimale ;
 fixer t ;
 POUR tout successeur u de t
  SI d[s, u] > d[s, t] + p(t -> u)
  ALORS
   d[s, u] = d[s, t] + p(t -> u)
  FIN SI
 FIN POUR
FIN TANT QUE
```

Fig.8.10 – Dijkstra : Plus court chemin

Exemple d'application de l'algorithme de Dijkstra. Soit à chercher les plus courts chemins depuis le sommet A dans le graphe suivant :

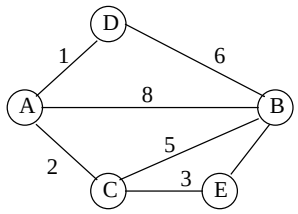
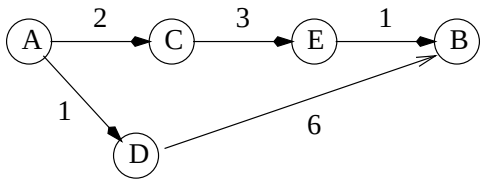


Fig.8.11 – Dijkstra : Exemple

Le tableau suivant donne les distances depuis A et les arcs marqués au cours des différentes étapes de l'algorithme :

sommets	A	B	C	D	E
initialement	0	∞	∞	∞	∞
on fixe A		8 (AB)	2 (AC)	1 (AD)	$+\infty$
on fixe D		7 (DB)	2		$+\infty$
on fixe C		7			5 (CE)
on fixe E		6 (EB)			
valeurs finales	0	6(EB)	2 (AC)	1 (AD)	5 (CE)

On obtient ainsi l'arborescence



Plus long chemin dans un graphe

Le problème consiste à déterminer le poids du plus *long* chemin *acyclique* reliant un sommet à un autre du graphe.

Il n'est pas possible d'adapter directement les algorithmes de plus court chemin, compte tenu de l'existence possible de cycles dans le graphe.

Dans le cas général, on peut parcourir l'arbre des chemins acycliques partant d'un sommet s, au moyen d'un algorithme de parcours en profondeur utilisant une liste :

```
parcourir le graphe :
POUR tout sommet t different de s
 d[s, t] = +inf
FINPOUR
d[s, s] = 0
visiter s

visiter un sommet t :
 ajouter t a la liste
 POUR tout successeur u de t qui n'est pas
 dans la liste
 SI d[s, u] = +inf
    ou d[s, u] < d[s, t] + p(t -> u)
 ALORS
  d[s, u] = d[s, t] + p(t -> u)
  FIN SI
  visiter u
 FIN POUR
 enlever t de la liste
```

Fig.8.12 – Le plus long chemin au moyen d'une liste

Dans le cas particulier d'un graphe acyclique, on peut adapter efficacement les algorithmes de plus court chemin à la recherche des poids maximaux.

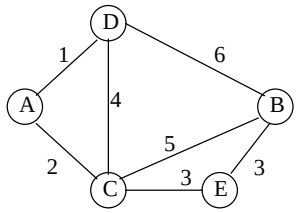
La modification des algorithmes de Ford ou de Floyd consiste simplement à changer le sens de l'inégalité ; la modification de l'algorithme de Dijkstra est un peu plus compliquée (algorithme de Bellmann).

Algorithme de Bellmann

```
POUR tout sommet t
d[s, t] = +inf
FINPOUR
d[s, s] = 0
fixer s
TANT QU'il reste des sommets non fixes
 choisir un sommet t non fixé dont tous
 les predecesseurs sont fixés
 POUR tout successeur u de t
  SI d[s, u] > d[s, t] + p(t -> u)
  ALORS
   d[s, u] = d[s, t] + p(t -> u)
  FIN SI
 FIN POUR
 fixer t
FIN TANT QUE
```

Fig.8.13 – Bellmann : Algorithme du plus long chemin d'un graphe

Exemple d'application de l'algorithme de Bellman Soit à chercher les plus longues distances depuis le sommet A dans le graphe suivant :



Le tableau suivant donne les plus longues distances depuis A calculées au cours des différentes étapes de l'algorithme :

sommets	A	B	C	D	E
initialement	0	∞	∞	∞	∞
on fixe A		∞	2	1	$+\infty$
on fixe C		7		6	5
on fixe D		7			5
on fixe E		8			
valeurs finales	0	8	2	6	5

Algorithme de Little

Le problème du choix d'un itinéraire (page 57) pourrait être résolu en examinant tous les ordonnancements possibles des sommets au départ de Bordeaux ; si l'on part d'un graphe à n sommets, cela donne $n!$ possibilités, ce qui conduit à un algorithme irréalisable, en pratique.

L'algorithme de Little

L'algorithme de Little est du type «séparation et évaluation» : on effectue un parcours en profondeur de l'arborescence binaire des choix possibles, en attribuant à chaque nœud n une évaluation par défaut $e_{(n)}$, et une valeur de regret à chaque choix auquel on renonce (évaluation par défaut du prix à payer pour renoncer à ce choix).

On remarque tout d'abord que l'on ne change pas le problème en soustrayant un nombre quelconque à une ligne ou une colonne de la matrice des distances entre les villes (cela revient à soustraire cette distance à tous les circuits hamiltoniens). On appellera **réduction** de la matrice l'opération consistant à soustraire de

chaque ligne le plus petit élément, puis à soustraire le plus petit élément de chaque colonne de la matrice ainsi obtenue.

L'évaluation par défaut associée à la racine de l'arborescence est la somme des nombres soustraits lors de cette réduction.

Chaque nœud de l'arborescence correspond à un choix (fait ou abandonné). L'évaluation par défaut associée à un nœud de type «choix fait» est la somme de l'évaluation par défaut associée au nœud père, et d'une évaluation par défaut du chemin restant à parcourir (obtenue par réduction de la matrice des liaisons encore possibles) ; il faut prendre soin d'éliminer les liaisons créant des circuits parasites, par exemple Nantes-Bordeaux si l'on a déjà choisi d'emprunter Bordeaux-Lyon et Lyon-Nantes).

Le regret associé à un choix abandonné est calculé de la façcon suivante : si on renonce à une liaison $s \to t$, il faudra sortir de s et entrer dans t par d'autres liaisons ; la valeur de regret associée à $s \to t$ est la somme du plus petit coût des liaisons $s \to s'$ et du plus petit coût des liaisons $t' \to t$ encore disponibles.

L'évaluation par défaut associée à un nœud de type «choix abandonné» est la somme du regret et de l'évaluation par défaut du nœud père.

On parcourt l'arborescence en profondeur, en ignorant les nœuds dont l'évaluation par défaut dépasse la valeur d'un circuit hamiltonien déjà trouvé.

```
algorithme de Little :
 maxi = +inf
 s[racine] = ville de depart
 calculer e[racine]
 visiter la racine

visiter un noeud n :
 SI e[n] <= maxi
 ALORS
  SI s[n] != ville de depart
  ALORS
   POUR toutes les liaisons
                s[n] -> v possibles
    calculer le regret de s[n] -> v
   FIN POUR
   soit s[n] -> v une liaison de
   regret maximal r
   cr\'eer les fils v et V de n
   s[v] = v
   calculer e[v]
   visiter v
   s[V] = s[n]
   e[V] = e[n] + r
   visiter V
  SINON
   maxi = max[maxi, e[n])
  FIN SI
 FIN SI
```

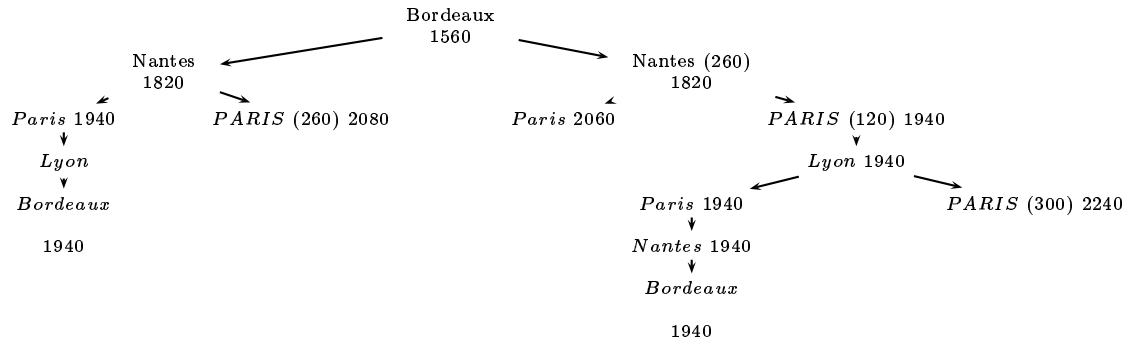


Fig.8.14 – Littel : Algorithme du plus long chemin d'un graphe

Troisième partie

Logique et algorithmique

1. Les expressions binaires

Ensembles binaires algébriques

On appelle variable ou fonction binaire, toute variable ou toute fonction ne pouvant prendre que l'une des deux *valeurs algébriques distinctes* $a \neq b$, à l'exclusion de toute autre.

L'*ensemble* «E_{ab}» des variables et des fonctions ainsi définies est appelé ensemble binaire algébrique.

Le choix de l'ensemble binaire E_{01} se justifie par l'adoption d'une valeur d'absorption «0» et d'une valeur neutre «1» qui font du *produit algébrique* une fonction binaire appartenant au même ensemble.

$$\overline{x} = 1 - x$$

Produit

Les expressions conjonctives sont utilisées dans différents contextes: pour les commentaires, dans les évaluations d'expressions, dans la fabrication des ordinateurs, plus précisément dans les unités centrales...

Conjonction en programmation

Ce programme rudimentaire et erroné, pour illustrer notre propos:

```
...
01   coup = 1 ;
02   while (1) {
04 /* code pas devine', coup <= 3 */
05     if (coup == 3) {
07       printf ("Pas trouve'\n") ;
08       exit (1) ;
09     }
10 /* code pas devine' */
11     printf ("Proposition : ") ;
12     scanf ("%d", &proposition) ;
13     if (code == proposition) {
15       printf ("Ok\n") ;
16       exit (0) ;
17     }
18 /* pas devine', coup +1 essais */
19     coup++ ;
20   }
   ...
```

Conjonction en logique

Une assertion conjonctive est de la forme: *P et Q*. Nous écrirons *PQ* ou *P.Q*, ce que certains auteurs notent: $P \wedge Q$, $P\&\&Q$ ou P AND Q.

Supposons qu'une telle proposition, par exemple dans le programme ci-contre «*Le code n'est pas deviné et l'on a fait moins de 3 essais*» soit vraie. L'usage habituel de la conjonction «et» est tel que nous entendons que «Le code n'est pas deviné» *et* «on a fait moins de 3 essais» sont également deux assertions vraies. Ceci conduit à poser les règles d'élimination suivantes:

Règles .e

n	$P.Q$			n	$P.Q$		
	...				...		
m	P	n, `.e`	et	m	Q	n, `.e`	

À la ligne `04`, nous faisons l'hypothèse *P.Q*, «`code pas devine' et coup <= 3`», donc à la ligne `10`, nous pouvons éliminer Q et écrire l'assertion «`code pas devine'`».

Cela tombe sous le sens. Inversement d'ailleurs, dans le cas où l'on sait que deux assertions P et Q sont vraies séparément, nous sommes disposés à affirmer que la proposition conjonctive *P.Q* est aussi vraie. D'où la règle d'introduction:

Règle .i

n	P	
m	Q	
	...	
	$P.Q$	n, m, `.i`

Conjonction en algèbre binaire

Soient n fonctions binaires $f_1, f_2, \ldots f_n$ telles que $f_i \in E_{01}, \forall i = 1, 2, \ldots n$, le produit algébrique de ces n fonctions appartient à l'ensemble E_{01}.

$$(P = f_1.f_2 \ldots f_n) \in E_{01}$$

Le produit P est en effet égal à l'unité lorsque toutes les fonctions en facteur sont simultanément égales à l'unité.

$$(f_1 = f_2 = \ldots = f_n = 1) \Longleftrightarrow (P = 1)$$

Il est nul dans tous les autres cas.

Conjonction électrique

Nous appellerons également le produit «*P*» fonction «ET» quand il fera l'objet d'application technologiques.

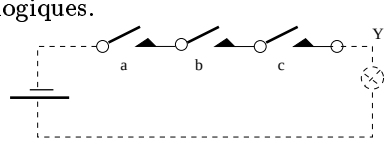


FIG. 1.1 – Fonction «ET» – **produit** $a.b.c = Y$

Produel

Disjonction en programmation

```
si m ≤ 0 ou m ≥ 13
afficher "Mois non valide" et s'arrêter
sinon
/* m > 0 et m < 13 */
```

Disjonction en logique

Une assertion *disjonctive* est de la forme : P *ou* Q. Nous écrirons $\left|\begin{matrix} P \\ Q \end{matrix}\right|$, ce que certains auteurs notent : $P \vee Q$, $P||Q$ ou P OR Q.

Supposons qu'une telle proposition, par exemple dans le programme ci-contre «$m \leq 0$ ou $m \geq 13$» soit vraie. L'usage habituel de la conjonction «ou» est tel que nous entendons qu'il suffit que l'une des deux assertions «$m \leq 0$» «$m \geq 13$» soit vraie pour que la phrase soit vraie et que le programme s'arrête en affichant le message d'erreur.

Disjonction en algèbre binaire

Il faut choisir un symbolisme simple qui puisse traduire dans l'expression écrite, la dualité qui caractérise les ensembles binaires et qui permette, en utilisant si possible les deux dimensions du plan, l'établissement de relations duales élémentaires.

Nous savons aussi, par dualité, qu'il est possible de faire correspondre au produit, la *fonction algébrique binaire*:

$$\pi = 1 - P$$
$$\pi = 1 - (1 - f_1).(1 - f_2)\ldots(1 - f_n) = \overline{\overline{f_1}.\overline{f_2}\ldots\overline{f_n}}$$
$$f_i \in E_{01}, \forall i = 1, 2, \ldots n) \Longrightarrow (\pi \in E_{01})$$

La fonction π est égale à zéro lorsque toutes les fonction f_i sont simultanément nulles,

$$f_1 = f_2 = \ldots = f_n = 0) \Longrightarrow (\pi = 0)$$

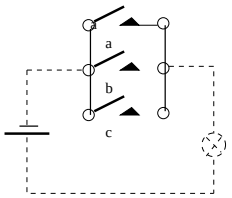
et qu'elle est égale à l'unité dans tous les autres cas, puisqu'il suffit qu'une fonction f_i au moins soit égale à l'unité pour que le produit algébrique $(1 - f_1).(1 - f_2)\ldots(1 - f_n)$ soit nul ; ce qui entraîne $\pi = 1$.

Nous conviendrons alors de représenter la fonction «π» , que nous apellerons «*produel*»[1] en groupant les termes qui la composent suivant une colonne verticale par analogie avec l'écriture horizontale du produit, pour traduire dans le symbolisme, la propriété de dualité.

$$\pi = \left|\begin{matrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{matrix}\right| = 1 - (1 - f_1)(1 - f_2)\ldots(1 - f_n) = \overline{\overline{f_1 f_2}\ldots\overline{f_n}}$$

Disjonction électrique

Nous appellerons également «π» , fonction «OU» dans le cas des applications technologiques.



$$Fig.1.2 - -Fonction\text{«OU»} - \textbf{produel} : \left|\begin{matrix} a \\ b \\ c \end{matrix}\right|$$
$$= 1 - (1 - a).(1 - b).(1 - c) = Z$$

Définition d'une structure logique binaire

La structure logique binaire se caractérise essentiellement par les deux opérations fondamentales qui ont des propriétés réciproques. Les symboles peuvent représenter des ensembles, des partitions d'ensembles, des groupes, des classes ou des éléments, voire même des opérateurs.

Les deux opérations sont, à la fois, **commutatives, associatives, réciproquement distributives** et possèdent, toutes deux, **la propriété d'idempotence** ; ce qui permet d'écrire successivement les égalités suivantes :

1. Les racines latines «pro» et « dualis» ont été choisies dans la constitution du néologisme « produel» pour marquer d'une part la nécessité de conserver la dualité dans les expressions binaires et aussi par analogie avec le mot «produit» .

Produit	*Produel*
$P = a.b.c$	$\pi = \left\| \begin{matrix} a \\ b \\ c \end{matrix} \right\|$
	$= 1-(1-a).(1-b).(1-c)$
– élément neutre «1»	– élément neutre «0»
$\|1.f\| = f$	$\left\| \begin{matrix} 0 \\ f \end{matrix} \right\| = f$
– élément absorbant «0»	– élément absorbant «1»
$\|0.f\| = 0$	$\left\| \begin{matrix} 1 \\ f \end{matrix} \right\| = 1$
– commutatif: $\| ab \| = \| ba \|$	– commutatif: $\left\| \begin{matrix} a \\ b \end{matrix} \right\| = \left\| \begin{matrix} b \\ a \end{matrix} \right\|$
– associatif: $\| a \| bc \| = \| ba \| c \| = \| cba \|$	– associatif: $\left\| \begin{matrix} a \\ b \\ c \end{matrix} \right\| = \left\| \begin{matrix} b \\ a \\ c \end{matrix} \right\| = \left\| \begin{matrix} c \\ b \\ a \end{matrix} \right\|$
– distributif: $\left\| \begin{matrix} ab \\ ac \\ d \end{matrix} \right\| = \left\| \begin{matrix} a & b \\ & c \\ & d \end{matrix} \right\| = \left\| \begin{matrix} d \\ b \\ c \end{matrix} \left\| \begin{matrix} a \end{matrix} \right\| = \left\| \begin{matrix} d & d \\ b & a \\ c & \end{matrix} \right\| \right.$	–distributif: $\left\| \begin{matrix} a & a \\ b & c \end{matrix} \right\| \left\| \begin{matrix} d \end{matrix} \right\| = \left\| \begin{matrix} a \\ bc \end{matrix} \right\| = \| d \| \left\| \begin{matrix} bc \\ a \end{matrix} \right\| = \left\| \begin{matrix} dbc \\ da \end{matrix} \right\|$
– idempotent: $\| aa \ldots a \| = \| a \|$	– idempotent: $\left\| \begin{matrix} a \\ a \\ \vdots \\ a \end{matrix} \right\| = \| a \|$
$P = \overbrace{a.a \ldots a}^{n} = a^n = a$	$\pi = \left\| \begin{matrix} a \\ a \\ \vdots \\ a \end{matrix} \right\} = 1-(1-a)^n$
pour $a = 0$ $(0)^n = 0$	pour $a = 0$ $\pi = 1-(1)^n = 0$
pour $a = 1$ $(1)^n = 1$	pour $a = 1$ $\pi = 1-(1)^n = 1$
Si dans un produit, deux facteurs sont complémentaires, le produit est nul.	Si dans un produel, deux facteurs duals sont complémentaires, le produel est égal à l'unité
$P = f.\overline{f}.P_1$	$\pi = \left\| \begin{matrix} f \\ \overline{f} \\ \pi_1 \end{matrix} \right\| = 1-(1-f).f.(1-\pi_1)$
$P = f.(1-f)P_1 = (f-f^2).P_1$	$= 1-(f-f^2).(1-\pi_1 = 1-0.(1-\pi_1)$
$P = (0.P_1) = 0$	$\pi = \left\| \begin{matrix} 1 \\ \pi_1 \end{matrix} \right\| = 1$
$(f.\overline{f}) = 0$	$\left\| \begin{matrix} f \\ \overline{f} \end{matrix} \right\| = 1$

Théorème de De Morgan

Ce théorème est contenu implicitement dans les définitions précédentes.

$$1 - \left\| \begin{matrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{matrix} \right\| = \overline{\left\| \begin{matrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{matrix} \right\|} = \overline{f_1}.\overline{f_2} \ldots \overline{f_n} = (1-f_1).(1-f_2) \ldots (1-f_n)$$

Le complément d'un produel est égal au produit des compléments des facteurs duals qui le composent.

$$\overline{f_1.f_2 \ldots f_n}s = \left\| \begin{matrix} \overline{f_1} \\ \overline{f_2} \\ \vdots \\ \overline{f_1} \end{matrix} \right\| = 1-(f_1.f_2 \ldots f_n)$$

Le complément d'un produit est égal au produel des comléments des facteurs qui le composent.

Fonctions canoniques et tables de vérité

Toute fonction binaire peut s'exprimer soit par un produit de produels soit par un produel de produits. Les expressions obtenues sont appelées «*fonctions canoniques*» .

Si une fonction binaire dépend de «n» variables distinctes $x_1, x_2, \ldots x_n$, nous ne pouvons envisager au total que 2^n combinaisons de valeurs distinctes (0 ou 1) de ces «n» variables. Si la fonction binaire est égale à l'unité pour «p» combinaisons, elle est nécessairement égale à zéro pour les «$2^n - p$» combinaisons complémentaires et nous avons dans tous les cas $p < 2^n$. Nous pouvons donc écrire la fonction sous la forme d'un *produel de produits* que nous appellerons, par définition, «*première forme canonique*».

$$F_1 = \left\| \begin{matrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \ldots \\ \ldots \\ f_p(x_1, x_2, \ldots, x_n) \end{matrix} \right\|$$

Il est également possible d'exprimer la même fonction «F_1» par un *produit de produels* appeleé, par définition, «*deuxième forme canonique*»:

$$F_1 = |f_1'(x_1, x_2, \ldots, x_n)||\ldots||f_q'(x_1, x_2, \ldots, x_n)|$$

Nous appellerons «*transposition*», *le passage, pour une même fonction, de la première à la deuxième forme canonique et réciproquement.*

Établissement de la première forme canonique

On inscrit horizontalement les variables dans un ordre quelconque, puis on porte successivement sous ces variables dans le sens horizontal, les combinaisons de valeurs pour lesquelles la fonction est égale à l'unité. Il suffit alors de remplacer respectivement dans le tableau obtenu, chaque valeur «1» par la variable «x_k» de la même colonne et chaque valeur «0» par le complément $\overline{x_k}$ de la variable de la même colonne.

Exemple :

Établir la première forme canonique d'une fonction de trois variables x_1, x_2, x_3, égale à l'unité lorsqu'une des variables est égale à l'unité, les deux autres étant nulles.

La table de vérité s'écrit :

x_1	x_2	x_3	f
1	0	0	1
0	1	0	1
0	0	1	1

La première forme canonique s'établit immédiatement à partir de la table de vérité :

$$f = \left| \begin{array}{l} x_1.\overline{x}_2.\overline{x}_3 \\ \overline{x}_1.x_2.\overline{x}_3 \\ \overline{x}_1.\overline{x}_2.x_3 \end{array} \right|$$

Établissement de la deuxième forme canonique

Pour établir la deuxième forme canonique relative aux combinaisons de valeurs de «n» variables pour lesquelles cette fonction est nulle, on inscrit verticalement les variables $x_1, x_2, \ldots, x_n$ dans un ordre quelconque et successivement dans le même ordre vertical, les combinaisons de valeurs pour lesquelles $f = 0$. Il suffit alors d'écrire les produit des produels obtenus en remplaçant respectivement dans ce tableau, chaque valeur «0» par la variable «x_k» de la même ligne et chaque valeurs «1» par le complèment «$\overline{x}_k$» de la variable de la même ligne.

Exemple :

Établir la deuxième forme canonique d'une fonction de trois variables x_1, x_2, x_3, égale à zéro lorsque deux au moins des trois variables sont égales à l'unité.

La table de vérité s'écrit :

x_1	0	1	1	1
x_2	1	0	1	1
x_3	1	1	0	1
f	0			

Comme nous l'avons indiqué précédemment, nous en tirons le tableau :

x_1	0	1	1	1
x_2	1	0	1	1
x_3	1	1	0	1

d'où la fonction cherchée :

$$f = \left| \begin{array}{l|l|l|l} x_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_1 \\ \overline{x}_2 & x_2 & \overline{x}_2 & \overline{x}_2 \\ \overline{x}_3 & \overline{x}_3 & x_3 & \overline{x}_3 \end{array} \right|$$

Présentation des tables de vérité

Tables de vérité complète.

– Nous dirons qu'une table de vérité est complète lorsqu'elle fait apparaître la totalité des «2^n» (combinaisons de valeurs possibles, relatives aux «n») variables dont dépend la fonction.

Nous conviendrons d'inscrire les valeurs (0 ou 1) que prend la fonction à droite d'un double trait vertical de séparation et sur la même ligne que la combinaison correspondante des valeurs des variables.

Ainsi la table de vérité complète d'une fonction de trois variables, $F(a, b, c)$, peut s'écrire par exemple :

Table de vérité

	a	b	c	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Il est souvent intéressant de faire figurer dans une colonne, à gauche, des repères décimaux qui correspondent aux nombres représentés par les chiffres binaires a, b, c, en plaçant ces nombres dans l'ordre naturel. Cette disposition permet, en particulier, de s'assurer qu'aucune combinaison n'a été oubliée.

Une table de vérité complète autorise, en suivant les règles énoncées précédemment, *l'écriture immédiate de la fonction sous ses deux formes canoniques.*

En ce qui concerne l'éxemple donné nous pouvons écrire :

$$F = \left| \begin{array}{l} \overline{a}\overline{b}\overline{c} \\ a\overline{b}\overline{c} \\ a\overline{b}\overline{c} \\ abc \end{array} \right| = \left| \begin{array}{l|l|l|l} a & a & a & \overline{a} \\ b & \overline{b} & \overline{b} & b \\ \overline{c} & c & \overline{c} & \overline{c} \end{array} \right|$$

La première forme utilise les combinaisons $F = 1$, (0, 4, 6, 7) et la seconde les combinaisons pour les combinaisons $F = 0$, (1, 2, 3, 5).

Tables de vérité incomplètes

Une fonction binaire est entièrement définie si l'on connaît seulement les combinaisons de valeurs des variables pour lesquelles elle conserve la même valeur (0 ou 1).

Nous appellerons, par définition, table de vérité incomplète, le tableau dans lequel sont inscrites ces combinaisons. Nous préciserons à droite de ce tableau, la valeur correspondante de la fonction. Pour chaque fonction il existe donc deux tables de vérité incomplètes.

Les deux tables de vérité incomplète qui correspondent à la fonction précédente $F(a,b,c)$ peuvent sécrire suivant que l'on choisit pour «F» la valeur «0» ou la valeur «1» :

$F=1$

	a	b	c	F
0	0	0	0	
4	1	0	0	
6	1	1	0	1
7	1	1	1	

$F=0$

	a	b	c	F
1	0	0	1	
2	0	1	0	
3	0	1	1	0
5	1	0	1	

Une table de vérité incomplète ne permet d'écrire que l'une des deux formes canoniques. La propriété de dualité la rend cependant suffisante pour définir complètement une fonction binaire.

Tables de vérité réduites

Ce sont des tables de vérité incomplètes dans lesquelles certaines combinaisons sont groupées afin de tenir compte d'une partie ou de la totalité des *adjacences* qui existent entre elles. Ces adjacences étant repérées dans la table par le symbole «ϕ» qui signifie que la valeur prise par la variable peut être indifféremment «0» ou «1». Nous pouvons tirer de l'exemple précédent différentes tables de vérité réduites, parmi lesquelles les deux suivantes :

	a	b	c	F
0–4	ϕ	0	0	
6–7	1	1	ϕ	1

	a	b	c	F
1–5	ϕ	0	1	
2–3	0	1	ϕ	0

L'adjacence, comme nous le verrons quand nous étudierons les simplifications, a pour effet de supprimer la variable *biforme* (directe et complémentée) dans le produit ou le produel qui résulte des combinaisons groupées. Une table de vérité réduite ne donne donc plus une forme cannonique mais une forme déjà simplifiées. Dans le cas envisagé, nous tirons de la première table de vérité réduite :

$$F=\left|\begin{matrix}\overline{b}\overline{c}\\ ab\end{matrix}\right|$$

De la seconde table de vérité réduite nous tirons :

$$F=\left|\begin{matrix}a & \overline{b}\\ \overline{c} & b\end{matrix}\right|$$

Simplification des fonctions binaires

Mise en facteur et développement

L'ensemble binaire «E_{01}» définit un anneau commutatif automorphe, puisque «$E_{01}=E_{10}$» et que l'on a choisi deux lois algébriques internes de composition qui sont le produit «$P=xy$» et le produel :

$$\pi=1-(1-x)(1-y)=\left|\begin{matrix}x\\ y\end{matrix}\right|$$

Mise en facteur dans un produel

Considérons le produel :

$$F=\left|\begin{matrix}\varphi f_1\\ \varphi f_2\end{matrix}\right|=1-(1-\varphi f_1)(1-\varphi f_2)$$

L'expression algébrique développée s'écrit :
$F=\varphi f_1+\varphi f_2-\varphi^2 f_1 f_2$, en utilisant le théorème d'idempotence $\varphi^2=\varphi$, $F=\varphi f_1+\varphi f_2-\varphi f_1 f_2$, que nous pouvons écrire :

$$F=\varphi[1-(1-f_1)(1-f_2)]=\varphi\left|\begin{matrix}f_1\\ f_2\end{matrix}\right|$$

Ainsi se trouve démontrée l'identité réciproque :

$$\boxed{\left|\begin{matrix}\varphi f_1\\ \varphi f_2\end{matrix}\right|\equiv\varphi\left|\begin{matrix}f_1\\ f_2\end{matrix}\right|}$$

Mise en «facteur dual» dans un produit

Considérons la fonction :

$$F=\left|\begin{matrix}\varphi\\ f_1\end{matrix}\right|\left|\begin{matrix}\varphi\\ f_2\end{matrix}\right|$$

Le développement algébrique de «F» s'écrit :

$$F=[1-(1-\varphi)(1-f_1)][1-(1-\varphi)(1-f_2)]=$$

$$1-(1-\varphi)(1-f_1)-(1-\varphi)(1-f_2)+(1-\varphi)^2(1-f_1)(1-f_2)$$

Le théorème d'idempotence permet d'écrire $=(1-\varphi)^2=(1-\varphi)$, d'où l'expression de «$F$» :

$$F=1-(1-\varphi)[(1-f_1)+1(1-f_2)-(1-f_1)(1-f_2)]$$

$$=1-(1-\varphi)(1-f_1f_2)=\left|\begin{matrix}\varphi\\ f_1f_2\end{matrix}\right|$$

Propriétés des fonctions carrées biformes

Définitions

Nous appellerons *fonction carrée biforme*, une fonction biforme comprenant quatre termes groupés en carré suivant un produit de deux produels de deux facteurs duals, ou suivant un produel de deux produits de deux facteurs direct :

$$\left|\begin{array}{c|c}\varphi A\\ \overline{\varphi}\end{array}\right| \text{ et } \left|\begin{array}{c|c}\varphi & \overline{\varphi}\\ A_1 & B_1\end{array}\right|$$

sont des *fonctions carrées biformes en «φ»*.

Propriétés des fonctions carrées biformes

Les fonctions carrés biformes ont un aspect dual qui laisse prévoir des propriétés particulièrement intéressantes.

$$f_1\left|\begin{array}{c}x\\ f_2\end{array}\right| = \left|\begin{array}{c|c}\overline{x}x & x\\ f_1 & f_2\end{array}\right| = \left|\begin{array}{c|c|c}\overline{x} & x & x\\ f_1 & f_1 & f_2\end{array}\right| = \left|\begin{array}{c|c}\overline{x} & x\\ f_1 & f_1 f_2\end{array}\right|$$

$$\left|\begin{array}{c}\overline{x}\varphi_1\\ \varphi_2\end{array}\right| = \left|\begin{array}{c|c}\overline{x}\varphi_1 & \\ \overline{x} & \varphi_2\\ x & \end{array}\right| = \left|\begin{array}{c|c}x\varphi_2 & \\ \overline{x} & \varphi_1\\ & \varphi_2\end{array}\right|$$

$$\varphi_1\left|\begin{array}{c|c}x & \overline{x}\\ f_1 & f_2\end{array}\right| = \left|\begin{array}{c|c|c}x\overline{x} & x & \overline{x}\\ \varphi_1 & f_1 & f_2\end{array}\right| = \left|\begin{array}{c|c}x & \overline{x}\\ \varphi_1 f_1 & \varphi_1 f_2\end{array}\right|$$

$$\left|\begin{array}{c}\overline{x}f_1\\ xf_2\\ \varphi_1\end{array}\right| = \left|\begin{array}{c|c}\overline{x}f_1 & \\ xf_2 & \\ \overline{x} & \varphi_1\\ x & \end{array}\right| = \left|\begin{array}{c|c}\overline{x} & f_1\\ & \varphi_1\\ x & f_2\\ & \varphi_1\end{array}\right|$$

Si une fonction carrée biforme se présente sous la forme d'un produit de produels,

$$P = \left|\begin{array}{c|c}\varphi & \overline{\varphi}\\ A & B\end{array}\right|$$

nous pouvons l'écrire sous la forme de produels de produits en effectuant les produits élémentaires :

$$P = \left|\begin{array}{c}\varphi B\\ A\overline{\varphi}\\ AB\end{array}\right| = \left|\begin{array}{c|c}\varphi B & \\ A\overline{\varphi} & \\ AB & \varphi\\ & \overline{\varphi}\end{array}\right| = \left|\begin{array}{c|c}\varphi & B\\ & AB\\ \overline{\varphi} & A\\ & AB\end{array}\right|$$

$$\left|\begin{array}{c}A\\ AB\end{array}\right| = B \text{ et } \left|\begin{array}{c}A\\ AB\end{array}\right| = A$$

D'où

$$P = \left|\begin{array}{c}\varphi B\\ A\overline{\varphi}\end{array}\right| = \left|\begin{array}{c|c}\varphi & B\\ A & \overline{\varphi}\end{array}\right|$$

Théorème *Une fonction carrée biforme n'est pas modifiée par la suppression ou le tracé d'un trait vertical médian, à condition de disposer les facteurs complémentaires suivant une diagonale du carré correspondant à son expression symbolique.*

Les facteurs «A» et «B» sont appelés simplement *facteurs diagonaux* de la fonction carrée biforme.

Ce théorème nous permet de passer facilement d'un produel à un produit et vice-versa, sans introduire de terme redondant. Ce qui justifie l'attention particulière consacrée à l'étude des fonctions carrés biformes.

Simplifications par transposition, mises en facteurs et adjacences.

Nous avons constaté dans ce qui précède que les transpositions, les mises en facteurs et les adjacences sont des moyens qui permettent d'opérer des simplifications sur les fonctions binaires.

La méthode que nous proposons réunit ces trois moyens de façon efficace et systématique. Elle permet de simplifier une fonction binaire mise sous forme canonique avec le maximum de rapidité.

Nous savons qu'une *forme canonique* est en générale une *fonction carrée biforme* lorsque l'une des variables a été mise en facteur partielle[2]

$$F(x_1, x_2, \ldots, x_n) = \left|\begin{array}{c|c}x_i G\\ H\overline{x_i}\end{array}\right| = \left|\begin{array}{c|c}x_i & G\\ H & \overline{x_i}\end{array}\right|$$

Si «F» est une fonction canonique, «H» et «G» sont également des fonctions canoniques qui ne contiennent plus la variable «x_i».

Elles sont donc également carrées biformes et les fonctions diagonales résiduelles sont toujours des fonctions canoniques sur lesquelles il est donc possible d'effectuer des transpositions.

Soit «p» le nombre de produits ou de produels élémentaires d'une fonction canonique. Le nombre total des variables litérales qui apparaissent dans la fonction est égale à «$p \times n$» si «n» désigne le nombre des variables indépendantes.

Le nombre total de variables littérales qui apparaissent après la mise en facteur partiel de la variable «x» :

$p(n-1)+2$, si «x_i» est une variable biforme, et $P(n-1)+1$, si «x_i» est une varaible monoforme.

Une fonction binaire peut être simplifiée, en opérant successivement des mises en facteur partielles, des transpositions et des réductions par adjacences.

2. La mise en facteur partielle correspond à une disjonction.

Théorème *Lorsqu'une fonction binaire mise sous forme canonique, est simplifiée par mise en facteur partielle d'une variable, suivie d'une réduction par transposition; la simplification est optimale lorsque la mise en facteur partielle a été faite par rapport à une variable pour laquelle le module de la différence $|p_0 - p_1|$ est maximal. Dans cette formule «p_0» peut représenter le nombre de fois que la variable est écrite sous forme directe, et «p_1» le nombre de fois qu'elle est écrite sous forme complémentée, dans la fonction avant mise en facteur.*

Méthode pratique

Pour opérer en pratique une simplification par transpositions mises en facteurs et adjacences, on procède de la manière suivante :

- On écrit d'abord la fonction sous forme canonique si elle ne l'était pas, et on la transpose dans le cas où le nombre de termes (produits ou produels élémentaires) est supérieur à «2^{n-1}» (n étant le nombre de variables). On calcule ensuite, pour chaque variable, le module $|p_0 - p_1|$. On écrit la fonction carrée biforme relative à la variable, ou à l'une des variables «x_i» qui correspond au maximum du module $|p_0 - p_1|$. «H» et «G» étant les fonctions diagonales de la fonction carrée biforme obtenue, on détermine le nombre «a» des termes communs à ces fonctions diagonales.
  Si «H» et «G» ne sont pas réductible par transposition, on simplifie par adjacences en écrivant selon le cas :

$$\left|\begin{array}{c} x_i H_1 \\ G_1 \overline{x_i} \\ J_1 \end{array}\right| ou \left|\begin{array}{c|c} x_i & H_1 \\ G_1 & \overline{x_i} \end{array}\right| K_1$$

  «H_1» et «G_1» étant obtenus à partir des fonctions «H» et «G» en supprimant dans ces dernières les termes communs. J_1 et K_1 représentent donc respectivement et selon le cas, le produel ou le produit des termes communs à «G» et «H».
- Lorsque l'une des fonctions diagonales, «H» par exemple, est réductible par transpositions, c'est-à-dire que le nombre de termes qu'elle comprend est égal à $2^{n-2} + k$, on calcule «k» que l'on compare au nombre d'adjacences «a».
- Si $2k \leq a$, on procède comme précédemment.
- Si $a < 2k$, on transpose le fonction «H» mais on écrit «G_1» et «J_1» ou «K_1» en tenant compte des adjacences.
  En appelant «H_t» la fonction transposée de «H» ($H_t = H$), on obtient selon le cas :

$$\left|\begin{array}{c} x_i H_t \\ G_1 \overline{x_i} \\ J_1 \end{array}\right| ou \left|\begin{array}{c|c} x_i & H_t \\ G_1 & \overline{x_i} \end{array}\right| K_1$$

  On poursuit la simplification en répétant les mêmes opérations pour les fonctions canonique H_t, H_1, G_1, J_1 ou K_1, jusqu'à épuisement du nombre de variables.

Exemple Simplifier la fonction $F(a, b, c, d)$ égale à l'unité pour les combinaisons de valeurs binaires qui correspondent aux nombres 2, 3, 4, 5, 6, 7, 9, 10, 23, 13 des quatre variables rangées dans l'ordre a, b, c, d.

La fonction, écrite sous la première forme canonique, comprendrait dix produits alors que le nombre total de combinaisons de valeurs des quatre variables est égal à $2^3 = 16$.

Nous pouvons donc simplifier par transposition en écrivant la fonction sous la deuxième forme canonique qui comporte les $16 - 10 = 6$ combinaisons complémentaires $0, 1, 8, 11, 14, 15$.

$$F = \left|\begin{array}{c|c|c|c|c|c} a & \overline{a} & a & \overline{a} & a & \overline{a} \\ b & b & b & \overline{b} & \overline{b} & \overline{b} \\ c & c & c & c & \overline{c} & \overline{c} \\ d & d & \overline{d} & \overline{d} & \overline{d} & \overline{d} \\ \\ 0 & 1 & 8 & 11 & 14 & 15 \end{array}\right|$$

Le calcul des modules $|p_0 - p_1|$ fournit :

- pour «a» $|p_0 - p_1| = 0$
- pour «b» $|p_0 - p_1| = 0$
- pour «c» $|p_0 - p_1| = 2$
- pour «d» $|p_0 - p_1| = 2$

Nous pouvons donc mettre «c» ou «d» en facteur dual. Établissons la fonction carrée biforme relative à «d» par exemple :

$$F = \left|\begin{array}{cc|cccc} & d & & & \overline{d} & \\ a & \overline{a} & a & \overline{a} & a & \overline{a} \\ b & b & b & \overline{b} & \overline{b} & \overline{b} \\ c & c & c & c & \overline{c} & \overline{c} \end{array}\right|$$

Nous constatons qu'il existe une adjacence, indiquée par les flèches, mais que les fonctions diagonales ne sont pas susceptibles d'être simplifiées par transpositions.

Nous écrirons donc :

$$F = \left|\begin{array}{c|c} a & d \\ b & \overline{a} \\ c & b \\ & c \end{array}\right| \left|\begin{array}{c|c|c} \overline{a} & a & \overline{a} \\ \overline{b} & \overline{b} & \overline{b} \\ c & \overline{c} & \overline{c} \\ & \overline{d} & \end{array}\right|$$

Sans qu'il soit besoin de calculer les valeurs de $|p_0 - p_1|$, nous voyons que $\overline{b}$ peut être mis en facteur dual dans la fonction diagonale :

$$\left|\begin{array}{c|c|c} \overline{a} & a & \overline{a} \\ \overline{b} & \overline{b} & \overline{b} \\ c & \overline{c} & \overline{c} \end{array}\right| = \left|\begin{array}{c|c|c} & \overline{b} & \\ \overline{a} & a & \overline{a} \\ c & \overline{c} & \overline{c} \end{array}\right|$$

$\left|\begin{array}{c|c|c} \overline{a} & a & \overline{a} \\ c & \overline{c} & \overline{c} \end{array}\right|$ peut être simplifié par transposition. Cette fonction est égale au complément du produel $\left|\begin{array}{c} a \\ c \end{array}\right|$ qui n'y figure pas et qui correspond à la quatrième combinaison possible des deux variables «a» et «c».

$$\left|\begin{array}{c|c|c} \overline{a} & a & \overline{a} \\ c & \overline{c} & \overline{c} \end{array}\right| = \boxed{\begin{array}{c} a \\ c \end{array}} = \overline{a}\,\overline{c}$$

Finalement :

$$F = \left|\begin{array}{c|c|c} a & d & \overline{b} \\ b & \overline{a} & \overline{a}\,\overline{c} \\ c & b & \overline{d} \\ & c & \end{array}\right| = \left|\begin{array}{c|c} c & \overline{b} \\ ad & \overline{a}\,\overline{c} \\ b & \overline{d} \end{array}\right|$$

Nous pouvons, en pratique, opérer directement sur les tables de vérités. Nous pouvons, en ce qui concerne l'exemple proposé, écrire la table de vérité incomplète qui correspond aux valeurs «zéro» de la fonction «F» et qui comprend les $16 - 10 = 6$ combinaisons repérées par les nombres décimaux $0, 1, 8, 11, 14$ et 15.

		d	c	b	a	F
	0	0	0	0	0	|
	1	0	0	0	1	|
	8	1	0	0	0	0
→	11	1	0	1	1	|
adjacence	14	1	1	1	0	|
relative à «c» →	15	1	1	1	1	|
valeurs de $\lvert p_0 - p_1 \rvert$		2	2	0	0	

Le calcul des modules $|p_0 - p_1|$ nous indique que nous devons mettre «d» ou «c» en facteur. Choisissant «c» nous pouvons écrire :

$$F = \left|\begin{array}{c|c} c & \overline{c} \\ \varphi_1 & \varphi_2 \end{array}\right| C_1$$

«C_1» corespond à une seule adjacence et s'écrit $C_1 = \left|\begin{array}{c} \overline{a} \\ \overline{b} \\ \overline{d} \end{array}\right|$

«φ_1» et «φ_2» ne se simplifient pas par transposition et nous pouvons établir :

d	b	a	φ_1
0	0	0	
0	0	1	0
1	0	0	
$\lvert p_0 - p_1 \rvert$	1	3	1

d	b	a	φ_2
1	1	0	0

Nous tirons de ces tables, $\varphi_1 = \left|\begin{array}{c} b \\ \varphi 3 \end{array}\right|$ et $\varphi_2 = \left|\begin{array}{c} a \\ \overline{b} \\ \overline{d} \end{array}\right|$

La fonction «φ_3» peut être simplifiée par transposition puisqu'elle fait apparaître trois combinaisons distinctes des deux variables «a» et «d» alors qu'il en exite $2^2 = 4$ au total.

d	a	φ_3
1	1	1

$\varphi_3 = a\,d$

Nous pouvons écrire ainsi :

$$F = \left|\begin{array}{c|c|c} b & a & \overline{a} \\ c & \overline{b} & \overline{b} \\ ad & \overline{c} & \overline{d} \\ & \overline{d} & \end{array}\right|$$

Après mis en facteur dual de $\left|\begin{array}{c} \overline{b} \\ \overline{d} \end{array}\right|$, nous obtenons :

$$F = \left|\begin{array}{c|c} b & \overline{b} \\ c & \overline{d} \\ ad & \overline{ac} \end{array}\right|$$

En utilisant les propriétés des fonctions carrées biformes, nous pouvons écrire également :

$$F = \left|\begin{array}{c|c} b & \overline{d} \\ & \overline{a}\,\overline{c} \\ \overline{b} & c \\ & ad \end{array}\right|$$

N.B. Il résulte de l'étude qui vient d'être faite que les formes de produits de produels ou inversement, de produels de produits, correspondent très rarement à des formes optimales relativement aux variables littérales. Dans le cas qui nous occupe, les formes

$$F = \left|\begin{array}{c|c|c|c} a & b & \overline{a} & \overline{b} \\ b & c & \overline{b} & \overline{c} \\ c & d & \overline{d} & \overline{d} \end{array}\right| et\, F = \left|\begin{array}{c} c\overline{b} \\ a\overline{b}d \\ \overline{a}b\overline{c} \\ b\overline{d} \end{array}\right|$$

comprennent respectivement 12 et 10 variables littérales. Elles ne sont pas optimales, puisque nous pouvons les réduire à huit variables par mise en facteur partielle. Ce résultat montre l'impossibilité d'obtenir des formes équivalentes optimales par la simple recherche des implicants comme l'avaient cru Quine, Mc Cluskey ou Scheinman.

2. Un nouveau sophisme

Un problème de logique.

Le directeur de la prison fait comparaître trois détenus de choix et leur communique l'avis suivant :

«Pour des raisons que je n'ai pas à vous rapporter maintenant, messieurs, je dois libérer un d'entre vous. Pour décider lequel, j'en remets le sort à une épreuve que vous allez courir, s'il vous agrée.

« Vous êtes trois ici présents. Voici cinq disques qui ne diffèrent que par leur couleur : trois sont blancs, et deux sont noirs. Sans lui faire connaître duquel j'aurai fait choix, je vais fixer à chacun de vous un de ces disques entre les deux épaules, c'est-à-dire hors de la portée directe de son regard.

« Dès lors, tout loisir vous sera laissé de considérer vos compagnons et les disques dont chacun d'eux se montrera porteur, sans qu'il vous soit permis, bien entendu, de vous communiquer l'un à l'autre le résultat de votre inspection. Ce qu'au reste votre intérêt seul vous interdirait. Car c'est le premier à pouvoir en conclure sa propre couleur qui doit bénéficier de la mesure libératoire dont nous disposons.

« Encore faudra-t-il que sa conclusion soit fondée sur des motifs de logique, et non seulement de probabilité. A cet effet, il est convenu que, dès que l'un d'entre vous sera prêt à en formuler une telle, il franchira cette porte afin que, pris à part, il soit jugé sur sa réponse.»

Ce propos accepté, on pare nos trois sujets chacun d'un disque blanc, sans utiliser les noirs, dont on ne disposait, rappelons-le, qu'au nombre de deux. Comment les sujets peuvent-ils résoudre le problème ?

La solution parfaite.

Après s'être considérés entre eux un certain temps, les trois sujets font ensemble quelques pas qui les mènent de front à franchir la porte. Séparément, chacun fournit alors une réponse semblable qui s'exprime ainsi :

«Je suis un blanc, et voici comment je le sais. Étant donné que mes compagnons étaient des blancs, j'ai pensé que, si j'étais un noir, chacun d'eux eût pu en inférer ceci : «Si j'étais un noir moi aussi, l'autre, y devant reconnaître immédiatement qu'il est un blanc, serait sorti aussitôt, donc je ne suis pas un noir.» Et tous deux seraient sortis ensemble, convaincus d'être des blancs. S'ils n'en faisaient rien, c'est que j'étais un blanc comme eux. Sur quoi, j'ai pris la porte, pour faire connaître ma conclusion.» C'est ainsi que tous trois sont sortis simultanément forts des mêmes raisons de conclure.

Cette solution, qui se présente comme la plus parfaite que puisse comporter le problème, peut-elle être atteinte à l'expérience ? Nous laissons à l'initiative de chacun le soin d'en décider.

Discussion du sophisme.

On appelle A le sujet réel qui vient conclure pour lui-même, B et C ceux réfléchis sur la conduite desquels il établit sa déduction. Si la conviction de B, nous dira-t-on, se fonde sur l'expectative de C, l'assurance de celle-là doit logiquement se dissiper avec la levée de celle-ci ; réciproquement pour C par rapport à B ; et tous deux de rester dans l'indécision. Rien ne nécessite donc leur départ dans le cas où A serait un noir. D'où il résulte que A ne peut en déduire qu'il soit un blanc.

À quoi il faut répliquer d'abord que toute cette cogitation de B et de C leur est imputée à faux, puisque la situation qui seule pourrait la motiver chez eux de voir un noir n'est pas la vraie, et qu'il s'agit de savoir si, cette situation étant supposée, son développement logique leur est imputé à tort. Or il n'en est rien. Car, dans cette hypothèse, c'est le fait qu'aucun des deux n'est parti le premier qui donne à chacun à se penser comme blanc, et il est clair qu'il suffirait qu'ils hésitassent un instant pour que chacun d'eux soit rassuré, sans doute possible, dans sa conviction d'être un blanc. Car l'hésitation est exclue logiquement pour quiconque verrait deux noirs. Mais elle est aussi exclue réellement, dans cette première étape de la déduction, car, personne ne se trouvant en présence d'un noir et d'un blanc, il n'est question que personne sorte pour la raison qui s'en déduit. Mais l'objection se représente plus forte à la seconde étape de la déduction de A. Car, si c'est à bon droit qu'il est venu à sa conclusion qu'il est un blanc, en posant que, s'il était noir, les autres ne tarderaient pas à se savoir blancs et devraient sortir, voici qu'il lui faut en revenir, aussitôt l'a-t-il formée, puisque au moment d'être mû par elle, il voit les autres s'ébranler avec lui.

L'objection logique, va se représenter la même avec la réitération au mouvement et reproduire chez chacun des sujets le même doute et le même arrêt ; mais cette fois A ne peut tirer de l'arrêt commun qu'une conclusion sans équivoque. C'est que, s'il était un noir, B et C n'eussent pas dû s'arrêter. Une seule hésitation, en effet, est suffisante à ce que l'un à l'autre ils se démontrent que certainement ni l'un ni l'autre ne sont des noirs.

Quatrième partie

Annexes

A. Systèmes de numération

Généralités

Il est aisé, connaissant la base du système de numération utilisé, d'exprimer un nombre dans un autre système. Il suffit pour cela d'exprimer les chiffres et les poids dans le système de numération choisi.

Exemples :

- Dans le système à base « six » $(0, 1, 2, 3, 4, 5)$ le nombre « 324» est représenté en décimal par :
$$3 \times (6)^2 + 2 \times (6) + 4 = 124$$
- Transposons le nombre 3243312_5 dans le système héxadécimal (base 16), en passant par le système décimal.

$$\begin{aligned} 3243312_5 &= 3 \times (5)^6 + 2 \times (5)^5 + 4 \times (5)^4 + 3 \times (5)^3 + 3 \times (5)^2 + 1 \times (5) + 2 \\ &= 56082_{10} \end{aligned}$$

Pour calculer la valeur en base 16, on effectue les divisions successives par «16» du nombre décimal :

56082	16		
080	3505	16	
00082	030	219	16
000(2)	145	059	(13)
	0(1)	(11)	(D)
		(B)	

D'où

$$3243312_5 = \mathrm{DB12}_{16}$$

Numération binaire

La correspondance entre nombres binaires et décimaux s'établit aisément selon les tableaux suivants :

décimal	*binaire*
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

décimal			*binaire*
		2	10
2^2	=	4	100
2^3	=	8	1000
2^4	=	16	10000
2^5	=	32	100000
2^6	=	64	1000000
2^7	=	128	10000000
2^8	=	256	100000000
2^9	=	512	1000000000
2^{10}	=	1024	10000000000
2^{11}	=	2048	100000000000
2^{12}	=	4096	1000000000000

Pour passer d'un nombre binaire, au même nombre exprimé dans le système à base $b = (2)^n$, on groupe les chiffres binaires n par n en partant de la droite, puis on remplace chacun des groupes obtenus à la place qu'il occupe par le chiffre du système à base $(2)^n$ qui lui correspond.

B. Les seize opérateurs logique

a	b	y_0
0	0	0
0	1	0
1	0	0
1	1	0

$\mathbf{y_0 = 0}$

$y_0 = \left|\begin{array}{c|c|c|c} a & a & \overline{a} & \overline{a} \\ b & \overline{b} & b & \overline{b} \end{array}\right|$

$y_0 = \left|\begin{array}{cc} a\overline{a} \\ b\overline{b} \end{array}\right|$

0

a	b	y_1
0	0	0
0	1	0
1	0	0
1	1	1

$aetb$

$y_1 = ab$

$y_1 = \left|\begin{array}{c|c|c|c} a & a & \overline{a} & \overline{a} \\ b & \overline{b} & b & \overline{b} \end{array}\right| = \left|\begin{array}{c|c} a & \overline{a} \\ b\overline{b} & b \end{array}\right|$

$y_1 = a \left|\begin{array}{c|c} \overline{a} \\ b \end{array}\right| = \left|\begin{array}{cc} a\overline{a} \\ ab \end{array}\right|$

1

a	b	y_2
0	0	0
0	1	0
1	0	1
1	1	0

$\mathbf{y_2 = a\overline{b}}$

$y_2 = \left|\begin{array}{c|c|c|c} a & a & \overline{a} & \overline{a} \\ b & \overline{b} & b & \overline{b} \end{array}\right| = \left|\begin{array}{c|c} a & \overline{a} \\ b\overline{b} & \overline{b} \end{array}\right|$

$y_2 = a \left|\begin{array}{c|c} \overline{a} \\ \overline{b} \end{array}\right| = \left|\begin{array}{cc} a\overline{a} \\ a\overline{b} \end{array}\right|$

2

a	b	y_3
0	0	0
0	1	0
1	0	1
1	1	1

$\mathbf{y_3 = a}$

$y_3 = \left|\begin{array}{c|c|c|c} a & a & \overline{a} & \overline{a} \\ b & \overline{b} & b & \overline{b} \end{array}\right| = \left|\begin{array}{c|c} a & \overline{a} \\ b\overline{b} & \overline{b} \end{array}\right| = \left|\begin{array}{cc} a\overline{b} \\ ab \end{array}\right|$

$y_3 = a \left|\begin{array}{c|c} \overline{b} \\ b \end{array}\right| = \left|\begin{array}{cc} a\overline{b} \\ ba \end{array}\right|$

3

a	b	y_4
0	0	0
0	1	1
1	0	0
1	1	0

$\mathbf{y_4 = \overline{a}b}$

$y_4 = \left|\begin{array}{c|c|c|c} a & \overline{a} & \overline{a} & \overline{a} \\ b & b & \overline{b} & \overline{b} \end{array}\right| = \left|\begin{array}{c|c} a & \overline{a} \\ b & \overline{b}b \end{array}\right|$

$y_4 = \left|\begin{array}{c|c} a & \overline{a} \\ b & \end{array}\right| = \left|\begin{array}{cc} a\overline{a} \\ b\overline{a} \end{array}\right|$

4

a	b	y_5
0	0	0
0	1	1
1	0	0
1	1	1

$\mathbf{y_5 = b}$

$y_5 = \left|\begin{array}{cc} \overline{a}b \\ ab \end{array}\right| = \left|\begin{array}{c|c} \overline{a} & \\ a & b \end{array}\right|$

$y_5 = \left|\begin{array}{c|c|c|c} a & \overline{a} & a\overline{a} & ab \\ b & \overline{b} & b & b\overline{a} \end{array}\right|$

5

a	b	y_6
0	0	0
0	1	1
1	0	1
1	1	0

$y_6 = \left|\begin{array}{cc} \overline{a}b \\ a\overline{b} \end{array}\right|$

$y_6 = \left|\begin{array}{c|c} \overline{a} & a \\ \overline{b} & b \end{array}\right|$

6

a	b	y_7
0	0	0
0	1	1
1	0	1
1	1	1

$y_7 = \left|\begin{array}{c} a \\ b \end{array}\right|$

$y_7 = \left|\begin{array}{c} \overline{a}b \\ a\overline{b} \\ ab \end{array}\right| = \left|\begin{array}{c|c} & \overline{a}b \\ a & \overline{b} \\ & b \end{array}\right|$

$y_7 = \left|\begin{array}{cc} \overline{a}b \\ a \end{array}\right| = \left|\begin{array}{cc} \overline{a}b \\ aa \end{array}\right|$

7

a	b	y_8
0	0	1
0	1	0
1	0	0
1	1	0

$\mathbf{y_8 = \overline{a}\overline{b}}$

$y_8 = \left|\begin{array}{c|c|c|c} a & \overline{a} & \overline{a} & \overline{a} \\ b & b & \overline{b} & \overline{b} \end{array}\right| = \left|\begin{array}{c|c|c} a & \overline{a} & \\ b & b\overline{b} & \end{array}\right|$

$y_8 = \left|\begin{array}{cc} a\overline{a} \\ b\overline{a} \end{array}\right| = \left|\begin{array}{c|c|c} a & \overline{a} & \\ b & & \end{array}\right| = \overline{\left|\begin{array}{c} a \\ b \end{array}\right|}$

8

a	b	y_9
0	0	1
0	1	0
1	0	0
1	1	1

$\mathbf{y_9 = \left|\begin{array}{c} \overline{a}\overline{b} \\ ab \end{array}\right|}$

$y_9 = \left|\begin{array}{c|c} a & \overline{a} \\ b & \end{array}\right|$

$y_9 = \left|\begin{array}{c|c} a & b \\ b & \overline{a} \end{array}\right| = \left|\begin{array}{cc} ab \\ b\overline{a} \end{array}\right|$

9

a	b	y_{10}
0	0	1
0	1	0
1	0	1
1	1	0

$\mathbf{y_{10} = \overline{b}}$

$y_{10} = \left|\begin{array}{c|c|c|c} a & a & \overline{a} & \\ b & \overline{b} & a & \overline{b} \end{array}\right| = \left|\begin{array}{c|c} a & \overline{a} \\ b & \overline{b} \end{array}\right|$

$y_{10} = \left|\begin{array}{cc} a\overline{a} \\ \overline{b} \end{array}\right| = \left|\begin{array}{cc} a\overline{b} \\ \overline{b}a \end{array}\right|$

10

a	b	y_{11}
0	0	1
0	1	0
1	0	1
1	1	1

$\mathbf{y_{11} = b \to a}$

$y_{11} = \left|\begin{array}{c} \overline{a}\overline{b} \\ a\overline{b} \\ ab \end{array}\right| = \left|\begin{array}{c|c} & \overline{a}\overline{b} \\ a & \overline{b} \\ & b \end{array}\right|$

$y_{11} = \left|\begin{array}{cc} \overline{a}\overline{b} \\ a \end{array}\right| = \left|\begin{array}{cc} \overline{a}\overline{b} \\ aa \end{array}\right| = \left|\begin{array}{c} a \\ \overline{b} \end{array}\right|$

11

a	b	y_{12}
0	0	1
0	1	1
1	0	0
1	1	0

$\mathbf{y_{12} = \overline{a}}$

$y_{12} = \left|\begin{array}{cc} \overline{a}\overline{b} \\ \overline{a}b \end{array}\right| = \overline{a} \left|\begin{array}{c|c} \overline{b} \\ b \end{array}\right| = \left|\begin{array}{c|c} \overline{a} & \\ b & \overline{b} \end{array}\right|$

$y_{12} = \left|\begin{array}{c} \overline{a} \\ b\overline{b} \end{array}\right| = \left|\begin{array}{cc} \overline{a}\overline{b} \\ b\overline{a} \end{array}\right|$

12

a	b	y_{13}
0	0	1
0	1	1
1	0	0
1	1	1

$\mathbf{y_{13} = \left|\begin{array}{c} \overline{a} \\ b \end{array}\right|}$

$y_{13} = \left|\begin{array}{c} \overline{a}\overline{b} \\ \overline{a}b \\ ab \end{array}\right| = \left|\begin{array}{c|c} \overline{a} & \overline{b} \\ & b \\ ab & \end{array}\right| = \left|\begin{array}{cc} \overline{a}\overline{a} \\ ab \end{array}\right|$

$y_{13} = \left|\begin{array}{c} \overline{a}\overline{b} \\ b\overline{a} \\ ab \end{array}\right| = \left|\begin{array}{c|c|c} \overline{a} & \overline{b} & \\ b & \overline{a} & \\ ab & & \end{array}\right| = \left|\begin{array}{cc} \overline{a}\overline{b} \\ b \end{array}\right| \left|\begin{array}{c} \\ \overline{a} \end{array}\right|$

13

a	b	y_{14}
0	0	1
0	1	1
1	0	1
1	1	0

$y_{14} = \left|\begin{array}{c} \overline{a} \\ \overline{b} \end{array}\right| = \overline{a\overline{b}}$

$y_{14} = \left|\begin{array}{c} \overline{a}\overline{b} \\ \overline{a}b \\ a\overline{b} \end{array}\right| = \left|\begin{array}{c|c} \overline{a} & \overline{b} \\ b & \\ a\overline{b} & \end{array}\right| = \left|\begin{array}{c} \overline{a} \\ a\overline{b} \end{array}\right|$

$y_{14} = \left|\begin{array}{c} \overline{a}\overline{b} \\ a\overline{b} \end{array}\right|$

14

a	b	y_{15}
0	0	1
0	1	1
1	0	1
1	1	1

$\mathbf{y_7 = 1}$

$y_{15} = \left|\begin{array}{c} \overline{a}\overline{b} \\ \overline{a}b \\ a\overline{b} \\ ab \end{array}\right| = \left|\begin{array}{c|c} \overline{a} & \overline{b} \\ b & \\ & \\ a & \overline{b} \\ & b \end{array}\right|$

$= \left|\begin{array}{cc} \overline{a} & \overline{b} \\ a & b \end{array}\right|$

15

C. Calculs remarquables en logique

Règles opératoires			Calculs remarquables			Liste des 16			
Règles de base									
p	10		p	1100			p	1100	
(R$\sim$) $\sim p$	01		q	1010			q	1010	
p	1100		$\sim p$	0011		(ϕ)	1	0000	0
q	1010		$\sim q$	0101		($p \wedge q$)	2	1000	pq
(R$\wedge$) $p \wedge q$	1000	$\leftrightarrow\leftrightarrow$	$\sim p \vee \sim q$	0111	M_1		3	0100	$p\overline{q}$
							4	0010	$\overline{p}q$
(R$\vee$) $p \vee q$	1110	$\leftrightarrow\leftrightarrow$	$\sim p \wedge \sim q$	0001	M_2 ni p ni q ($p \perp q$)		5	0001	$\overline{pq}$
(R$\supset$) $p \supset q$	1011	$\Longleftrightarrow$	$\sim q \supset \sim p$	1011		($p[q]$)	6	1100	p
						($q[p]$)	7	1010	q
$[q \supset p]$	1101	$\Longleftrightarrow$	$[\sim p \supset \sim q]$	1101		($p \equiv q$)	8	1001	$\overline{pq}$ / pq
						($p \vee \vee q$)	9	0110	$\overline{p}q$ / $p\overline{q}$
							10	0101	$\overline{q}$
							11	0011	$\overline{p}$
Règles surnuméraires									
(R$\equiv$) $p \equiv q$)	1001					($p \vee q$)	12	1110	p / q
						($q \supset p$)	13	1101	$\overline{p}$ / q
(R**w**) p **w** q)	0110					($p \supset q$)	14	1011	$\overline{q}$ / p
						($q \mid p$)	15	0111	$\overline{pq}$
						($\mathcal{U}$)	16	1111	1

D. Proverbes de programmation[1]

Ne violez pas les règles avant de les apprendre.

Étude du programme

pp 21, 23, 25, 31

- 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'analyse 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 biais
- 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*, ECP 1998, p. 10.

E. Table ASCII

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}.

Code ASCII

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

- 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}).

Le jeu de caractères ISO-8859-1

Pour pouvoir lire et écrire des documents avec accents, il est d'abord nécéssaire de s'entendre sur un jeu de caractères communs à tous. À première vue, la situation n'est pas rose : à l'exception de certains dinosaures qui dépendent encore du jeu EBCDIC, tout le monde comprend le jeu US-ASCII, mais cette norme ne compte que 128 caractères, dont aucun accentué. Pour les accents, il faut un jeu de caractères 8 bits, mais les fabricants ne se sont jamais entendus sur le codage des 128 caractères supplémentaires. Les micros PC utilisent un codage qui leur est propre, les MACS un autre, les terminaux DEC encore un autre, etc.

Heureusement il existe une norme internationale, ISO-8859-1, à laquelle on peut se fier. Cette norme est déjà utilisée par la quasi-totalite des systemes UNIX supportant les caractères 8 bits, les terminaux DEC VT220 et postérieurs, les PC sous MS-WINDOWS (à part quelques codes non standard), les AMIGAS, et surement bien d'autres systèmes ou terminaux. En plus de n'être pas particulière à un seul système, la norme ISO a l'avantage de ne pas définir de codes entre 128 et 159, domaine utilisé par certains terminaux pour des codes de contrôle.

En fait, ISO-8859-1 n'est qu'une partie de la norme ISO-8859, qui spécifie au moins 9 jeux de caractères pour diverses langues. ISO-8859-1 convient aux langues ouest-européennes, -4 à la scandinavie, -6 à l'arabe, etc.

En parcourant la liste des caractères, ne perdez pas votre temps à chercher la ligature 'œ' utilisée en français: elle n'y est pas. La petite histoire veut que ce manque soit du à une absence du représentant français sur le comité ISO-8859 au moment crucial de l'adoption finale ; on en aurait profité pour remplacer les 'œ' et 'Œ' (liés), qui étaient dans la proposition originale, par les signes de multiplication et de division. Si ça vous chatouille, vous pouvez vous consoler en songeant que le clavier d'une machine à écrire ne comprend pas non plus cette ligature.

Il existe en français des mots qui utilisent œcollés et des mots qui utilisent œséparés. Dans le premier cas, les deux lettres forment un seul son (ex: œsophage, œdeme, ...) et peuvent même ne constituer qu'une partie de la retranscription d'un son : dans cœur et œuf, le group O, E dans l'O, U ne fait qu'un seul son. Dans le second cas, les deux lettres forment soit deux sons séparés (ex : coexister), soit une diphtongue (ex : moelle). Il est à noter aussi que la prononciation du C devant Œ(liés) présente quelques subtilités : si tout le monde est d'accord sur la prononciation du mot cœur, savez-vous prononcer des mots comme cœlacante, cœlioscopie ...? L'utilisation ou la non-utilisation de la ligature a donc une incidence phonétique au même titre que les accents.

Les ligatures "œ"en français semblent être limitées aux mots suivants :

bœuf, cœur, chœur, fœtus, manœuvre, mœurs, nœud, œcumenisme, œdeme, œdipien, œil, œnologie, œsophage, œuf, œuvre, sœur, vœu, stœchiometrie, des noms propres et les dérivés de cœlome et ceux de œil.

Notons que les ligatures sont la raison de nombreuses discussions, parfois vives, sur le sujet : notamment certains arguent que "œ" n'est pas qu'une ligature typographique vu que celle-ci change la prononciation dans certains cas. Notons que le jeu de caractère ISO-8859-0 en cours d'homologation semble supporter "œ".

On remarquera que la majorité sont des mots très courants. La ligature "æ" quant a elle n'est plus usitée (à part pour le prénom Læticia et quelques expressions latines). « & » est une ancienne ligature (cf. M. GREVISSE, « Le bon usage », paragraphes 84ss.)

Certains systèmes n'utilisant pas le jeu standard ISO Latin-1 autorisent certaines ligatures (notamment MACOS). Les logiciels X-press et Pro-Lexis sous cet environnement supportent quelques ligatures comme « fi ».

Le système typographique commun sous environnements UNIX TeX[2] supporte également les ligatures (noeud s'écrit alors nœud, certaines ligatures sont automatiques). Ce système typographique libre écrit notamment par DONALD KNUTH (pour la base) et de nombreux contributeurs pour les extensions (p.ex. LaTeX, un langage de structuration de documents) est également disponible sous environnements non UNIX.

Le site
`ftp://ftp.uni-erlangen.de/pub/doc/ISO`
possède quelques documents intéressants sur les standards ISO, y compris 8859. Un autre site digne de mention est
`ftp://ftp.ulg.ac.be/pub/docs/iso8859/`,
qui contient deux textes d'André Pirard traitant de l'usage de l'ISO-8859-1 et contenant des tables de traductions entre ce dernier et quelques autres jeux de caractères d'usage courant. Ces textes se trouvent aussi en
`ftp://ftp.sri.ucl.ac.be/pub/ISO.8859-1`.

2. À l'évidence ce poly est réalisé avec LaTeX. Pensez-vous qu'un autre outil (gratuit) serait capable de la même prouesse ?

F. Le TDA liste

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     ;
}
```

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 ;
```

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") ;               \
}
```

Application : le fichier usager.c

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

/* fonctions pour le 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) ;
}
```

G. Le TDA pile

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)) ;                            \
}
*/
```

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) ;
    }
}
```

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)  ;
}
```

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") ;
}
```

H. Le TDA file

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)) ;            \
}
```

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) ;
    }
}
```

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)  ;
}
```

I. Le TDA arbre

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 ("0ter") ; 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  ;
}
```

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 () ;
```

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 ;                                                      \
}
```

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`

Table des figures

Index