UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Algorithmes d'allocation dynamique de mémoire

Crutzen, P.

Award date:
1985

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Algorithmes d'allocation dynamique de mémoire

Mémoire présenté par

P. CRUTZEN

en vue de l'obtention du titre de
Licencié et Maître en Informatique

1984-1985

Promoteur : M. H. LEROY

RUE GRANDGAGNAGE, 21 B - 5000 NAMUR (BELGIUM)

INTRODUCTION

L'allocation dynamique de mémoire s'entend dans un contexte de gestion de variables ou structures dont la configuration n'est pas déterminée a priori mais évolue fortement lors du déroulement du programme qui les manipule. Il n'est plus question de réserver de la place à ces structures dès la compilation, mais d'allouer cet espace selon les besoins instantanés. La gestion de cet espace requiert des techniques particulières. Depuis que certains langages, tels le LISP, sont implémentés via les structures de liste, ou qu'ils offrent à l'utilisateur la possibilité de créer et faire évoluer de telles constructions, de nombreux algorithmes ont vu le jour, qu'ils soient de niveau "implémentation" ou de niveau "utilisateur". Si certains d'entre eux sont considérés avec tout le respect qu'on attribue aux "solutions classiques", d'autres par contre sont jugés insatisfaisants et des recherches continuent dans le but de résoudre plusieurs problèmes spécifiques.

Au départ, il y a une certaine indétermination qui pèse sur le concept de structure de données, et ce à deux titres. D'abord parce que les variables structurées et pointeurs permettent de créer des édifices échappant à une modélisation mathématique stricte. Ensuite parce que d'un point de vue implémentation, il y a peu d'espoir qu'une stratégie "meilleure que toutes les autres dans tous les cas" voie le jour. Seule une appréciation statistique permet de juger de l'adéquation d'une solution particulière à une situation générale. Néanmoins, la plupart du temps,

il est possible de dégager une solution satisfaisante, performante pour un bon nombre de configurations. Il est à noter d'ailleurs que c'est souvent la plus simple : c'est un des enseignements que nous avons retirés de l'étude des algorithmes qui foisonnent dans les publications diverses.

L'objectif de cette étude est donc une synthèse et une présentation d'un certain nombre de concepts et algorithmes relevant des structures de données dynamiques. Nous sommes conscients que cette approche du problème, basée sur la littérature existante plus que sur des investigations inédites, n'offre pas un caractère tout à fait nouveau. D'un autre côté, l'esprit qui sous-tend ce travail se veut celui d'un cours sur les techniques d'implémentation et de programmation pouvant faire suite à ceux qui sont dispensés à l'institut.

Le premier chapitre est consacré à quelques rappels et mises au point, notamment en ce qui concerne les notations. Le concept de structure de données pouvant recouvrir diverses interprétations, il convenait de se mettre bien d'accord sur les définitions et aspects retenus.

Quelles sont les différentes structures de liste fréquemment employées ? Piles, files, listes circulaires, double chaînage, etc... sont autant de notions sur lesquelles se penche le chapitre II. Sans être exhaustif, il présente bon nombre de constructions et d'algorithmes qui leur sont attachés.

Une des hypothèses retenues dans le chapitre II est que tous les éléments qui interviennent dans les diverses structures sont de même taille et même découpe. Toutes les applications ne s'accomodent pas de cette restriction, tous les langages ne sont pas aussi stricts. Une autre organisation de l'allocation dynamique d'espace-mémoire est présentée dans le chapitre III, sans cette contrainte.

L'aspect le plus crucial de l'allocation dynamique est la non extensibilité de la mémoire centrale. Mais ce n'est pas parce que tout l'espace libre a été consommé que

le programme consommateur est condamné : il existe, dispersées dans la mémoire, des cellules disponibles qu'il s'agit de récupérer. Quelques techniques de récupération (garbage collection) sont étudiées dans le chapitre IV.

Nous avons dit que l'implémentation du langage LISP était basée sur les structures de liste. Un aperçu de cette implémentation est proposé en annexe; comme de nombreux problèmes rencontrés préalablement s'y retrouvent, on peut aussi considérer que cette annexe sert d'exemple général aux chapitres précédents.

Nous tenons beaucoup à remercier Monsieur le professeur Leroy pour l'aide et l'intérêt qu'il a portés à ce travail, ainsi que toutes les personnes qui m'ont encouragé.

CHAPITRE I

STRUCTURES DE DONNÉES

1.1. DONNÉES

Tout programme traite de l'information, ou plus exactement des représentations d'information (le nom d'un client n'est pas ce client, le chiffre 30 n'est pas le nombre de kilomètres qu'il y a entre Liège et Verviers, ...); ces représentations sont elles-mêmes sujettes à un codage binaire : on parle alors de la représentation interne. Lors de l'exécution d'un programme, ces représentations internes d'information ou <u>données</u> se trouvent en mémoire centrale (le problème des données mémorisées sur support externe ne nous intéresse pas directement), rangées selon divers principes. De façon générale, nous dirons que les données sont contenues dans des tables. Une table pourrait être considérée comme une collection d'informations, mais c'est insuffisant. En effet, les relations qui existent entre les différentes unités d'information importent quasi autant que les données elles-mêmes. Les tables sont <u>structurées</u>, et la connaissance des relations entre données est indispensable au programmeur de l'application qui manipulera cette structure.

Dans les cas les plus simples, l'information est représentée (à un niveau logique) par des variables simples (de types entier, réel, caractère, etc...). Les tables sont

réduites à un seul élément. Oublions ce cas trivial pour aborder des concepts plus évolués tels que tableaux à *n* dimensions, listes linéaires, arbres binaires, etc... Il s'agit ici d'organiser une collection de données, de même nature ou non, et de répondre aux questions

- quel est le premier élément ?
- quel est le dernier ?
- quel est l'élément qui précède (ou suit) un élément donné ?
- etc

La manière de programmer ces différentes réponses dépend bien évidemment de la structure globale à laquelle on est confronté, dont il faut dès lors maîtriser les techniques de représentation et de manipulation.

Ce chapitre a pour but d'introduire certains concepts et notations, tandis que le suivant exposera un certain nombre de structures de données du double point de vue de leurs propriétés et des algorithmes qui en permettent le traitement.

1.2. DESCRIPTION LOGIQUE DES STRUCTURES DE DONNÉES

L'élément d'information dans une table est l'article, qui sera également appelé, selon le contexte, enregistrement, noeud ou élément. Un article est décomposable en un certain nombre de champs, qui sont des parties identifiables et distinctes de l'enregistrement.

Un ou plusieurs champs de l'article peuvent consister en des références ou pointeurs vers d'autres articles de la table. D'un point de vue interne (voir 1.3.), le contenu d'un tel champ est l'adresse de l'article ainsi référencé. A priori, un pointeur est susceptible de désigner un article de n'importe quel type. Si tel n'est pas le cas, le pointeur est dit typé puisque les seuls articles qu'il peut référencer sont d'un type bien fixé. Un pointeur peut aussi ne désigner aucun article (indépendemment de son caractère typé ou non), si sa valeur ne correspond pas à une adresse. Convention-

nellement, *nil* ou 0 représentent cette valeur.

Le type (et donc la structure) des articles référencés par pointeurs est déterminé à la compilation, soit grâce au type du pointeur (PASCAL), soit grâce au contexte (PL/1). Une discussion plus approfondie sur l'usage et les notations attachées aux pointeurs se trouve au paragraphe 1.4.4. de ce chapitre.

Définir une structure de données, c'est créer un nouveau type. Abstraitement, un nouveau type est décrit par les opérations qui lui sont associées. Tous les langages ne sont pas "orientés objets", loin s'en faut, aussi le concepteur d'une application est-il encore souvent amené à définir ce nouveau type, pour pouvoir le manipuler, en fonction de types déjà connus. Il existe diverses méthodes de composition, qui sont rappelées ici par des exemples.

1.2.1. JUXTAPOSITION

```
type DATE = (JOUR : integer; MOIS : integer; ANNEE : integer);
type PERSONNE = (NOM : string; PRENOM : string; DATE-NAIS : date);
type SOCIETE = (NOM : string; CAPITAL : integer; SIEGE : string);
```

1.2.2. ENUMÉRATION

```
type ETAT-CIVIL = ('marié' , 'célibataire' , 'divorcé' , 'veuf');
```

1.2.3. SÉPARATION DES CAS ET PARTIE VARIABLE

```
type EMPLOYE = (NOM : string; ETAT : état-civil;
                    if état = 'marié' then (NOM-CONJOINT : string);
                    if état = 'célibataire then ();
                    if état = 'divorcé' then (DATE-DIVORCE : date);
                    if état = 'veuf' then () );
type COMPTE = (SOLDE : integer; TITULAIRE : (personne | société));
```

Le symbole | se lit "ou" (exclusif).

1.2.4. RÉCURSION

type *LISTE* = (*nil* | *liste-non-vide*);

type *LISTE-NON-VIDE* = (*ELEMENT* : *compte*; *liste*);

nil est ici utilisé pour arrêter la structure récursive. Il est clair qu'il existe d'autres moyens de définir une liste, probablement plus utilisés, qui font appel à un champ pointeur qui, si sa valeur est non *nil*, indique le compte suivant dans la liste. Néanmoins, cette approche est déjà liée à l'implémentation en mémoire, qui ne doit pas entrer en ligne de compte à ce niveau.

1.3. REPRÉSENTATION INTERNE

La représentation interne d'un article occupe un ou plusieurs mots-mémoire consécutifs, qui constituent une cellule. L'adresse d'un article est celle de sa cellule, c'est-à-dire celle du premier mot-mémoire qu'il occupe. Les adresses sont généralement relatives mais dans cet exposé elles seront considérées absolues (sauf indication explicite du contraire). Ceci dit, venons-en à la représentation interne des divers types mentionnés ci-haut.

La séparation ne pose pas de gros problèmes d'implémentation. Un objet de type T (type défini par

type $T = (T_1 \mid T_2 \mid \ldots \mid T_n)$

où T_i est de type connu) sera représenté par un indicateur de type, par exemple un entier i compris entre 1 et n, qui détermine le type T_i de l'objet considéré, et par un objet de ce type. Si les types T_i donnent lieu à des représentations internes différentes (taille des cellules ou structure de ces cellules), une solution pourrait être de réserver systématiquement la place maximale (on y reviendra).

Considérons à présent le cas d'une définition

type $T = (x_1 : T_1; \ldots ; x_n : T_n)$

par juxtaposition et sans récursion, telle que la taille requise pour les objets de type T_i (i = 1..n) est connue : il suffit de réserver une cellule dont la taille est la somme des tailles des cellules logeant les objets de type T_i.

La représentation interne d'un objet d'un type défini par énumération sera tout simplement un entier compris entre 1 et le nombre de valeurs énumérées.

Celle des objets d'un type à partie variable se déduit des deux cas précédents; la partie fixe joue le rôle de l'indicateur.

La représentation interne d'objets de type LISTE pose durement le problème de la taille variable et imprévisible. Pourtant, il suffit de considérer la liste comme une suite de blocs d'information de taille fixe mais en nombre variable. Ceux-ci peuvent être implémentés séquentiellement, dans des cellules contiguës, ou reliés entre eux par des pointeurs (dans ce cas-ci, un pointeur par bloc dont le contenu est l'adresse du suivant). Le pointeur du dernier bloc vaudra *nil*. Pour atteindre le premier bloc, il sera nécessaire de disposer d'un pointeur d'entrée, dont l'emplacement et la taille seront connus dès la compilation (méthodes classiques).

Cette technique est utilisable également pour les objets à partie variable ou à séparation de cas : l'objet est considéré comme ayant une certaine taille fixe, celle d'un pointeur et de la partie fixe et d'un indicateur éventuel, tandis que le reste de l'enregistrement se trouve localisé sur un "tas" (heap), référencé par le pointeur. Le tas est réservé aux structures spéciales.

La gestion de ce tas requiert des techniques particulières d'allocation. L'allocateur est un mécanisme attribuant dynamiquement des cellules au programme au fur et à mesure que celui-ci en réclame. Ces cellules sont tirées d'un réservoir de mots-mémoire, contigus ou non, qui sera appelé zone libre ou espace libre. C'est en grande partie à ce mécanisme qu'est consacrée cette étude.

1.4. Remarques et conventions

1.4.1. Données et structures

La façon de définir une liste qui est donnée ci-dessus (1.2.4.) est plus logique que pratique. Souvent en effet, comme signalé alors, le programmeur définira (par juxtaposition) un type COMPTE dont un des champs est une référence. Il a à sa disposition les moyens de réserver de la place pour placer en mémoire centrale un nouvel objet de type COMPTE. Dans certains cas, il aura même la possibilité de restituer les cellules dont le contenu ne l'intéresse plus. C'est le programmeur qui gère la structure de liste. Notons que dans le but de faciliter cette gestion, on a mis au point des machines et processus conçus dans le but de gérer les listes et autres structures chaînées. Cet attirail peut s'avérer efficace pour de grosses applications, mais fort lourd dans les cas les plus simples.

Il y a donc deux façons de considérer une structure de données : soit il s'agit de l'article lui-même, structuré en champs et sous-champs de types élémentaires, soit il s'agit de la collection des articles, considérées comme un tout et à laquelle s'appliquent des opérateurs spécifiques. Le concepteur d'applications envisage souvent le second aspect dans son étude, la structure pouvant être un fichier et ses modes d'accès, une liste triée, un tableau, ... C'est également cette interprétation qui retiendra notre attention, mais du point de vue de l'allocation dynamique de mémoire.

1.4.2. Taille fixe et variable

Lorsqu'une structure de données rassemble des articles de même type, c'est-à-dire qu'elle est composée de cellules de même taille et que la localisation dans ces cellules des champs pointeurs est toujours la même, on parle de cellules unitaillées. A l'inverse, il arrive que les cellules intervenant

dans une structure de données soient de taille variable, auquel cas elles sont dites multitaillées. Sauf mention explicite, seules les premières seront étudiées, malgré qu'elles ne soient pas les plus fréquentes.

1.4.3. PARTIES STATIQUES ET DYNAMIQUES

L'affirmation selon laquelle la représentation interne d'un article est localisée dans une cellule (mots-mémoire consécutifs) est mise en défaut lorsqu'on évoque la localisation d'un objet à partie variable ou à séparation de cas. En fait, la donnée elle-même, l'information pertinente, se trouve dans une cellule, sur le tas, référencée par un pointeur se trouvant dans la pile des variables directement accessibles. Une variable peut donner lieu à une représentation à deux niveaux, les parties statiques et dynamiques. La structure-mémoire de la première est connue à la compilation du type, la seconde dépend d'une élaboration dans le programme (c'est par exemple le cas des tableaux à bornes dynamiques).

1.4.4. POINTEURS ET ADRESSES

1.4.4.1. NIVEAU DE L'UTILISATEUR. L'utilisateur ne connait pas la notion d'adresse. L'intérêt des pointeurs, typés ou non, est qu'ils évitent de devoir connaître la structure de la représentation interne des objets de tout type. Imaginons un pointeur P désignant un objet dont les différents champs sont C_1, C_2, C_3 et NEXT, ce dernier étant aussi un pointeur. Si P est différent de *nil*, il y a moyen, par son intermédiaire, d'atteindre les divers champs de l'objet ainsi référencé. On envisage des notations telles que

$P.C_1$, $P.C_2$, $P.C_3$, P.NEXT (N.1)

ou

$C_1(P)$, $C_2(P)$, $C_3(P)$, NEXT(P). (N.2)

Une notation plus universelle, liée d'ailleurs aux langages qui ne disposent pas de variables structurées, est d'imaginer

que l'article est la réunion des éléments de même indice P de quatre tableaux C_1, C_2, C_3 et NEXT, ce qui conduit à la formulation

$$C_1(P),\ C_2(P),\ C_3(P),\ NEXT(P). \qquad (N.3)$$

La valeur *nil* est alors représentée par le nombre 0, les tableaux étant supposé être de la dimension [1..x]. La restriction essentielle est qu'il n'y a pas moyen (aisément) d'implémenter de la sorte une structure à séparation de cas. Néanmoins, c'est cette notation qui sera le plus fréquemment employée dans cette étude, d'autant plus que le plus souvent, nous travaillerons avec des articles supposés unitaillés.

Pour atteindre l'objet que NEXT référence, pour autant que sa valeur ne soit pas *nil*, on écrira de même, selon le formalisme retenu,

$$P.NEXT.C_1 \text{ (etc...)},\ C_1(NEXT(P)) \text{ ou } C_1[NEXT[P]].$$

1.4.4.2. NIVEAU DE L'IMPLEMENTATION. La mémoire elle-même peut être regardée comme un tableau de mots-mémoire, soit MM. Les indices sont alors équivalents aux adresses, et des opérations arithmétiques peuvent leur être appliquées. La notion de pointeur, elle, perd son sens dans la mesure où (normalement) le contenu des mots-mémoire ne nous intéresse pas directement. Le transfert du contenu d'un mot-mémoire d'adresse Q vers le mot-mémoire d'adresse P sera simplement indiqué par

$$MM[P] \leftarrow MM[Q].$$

Cependant, nous supposerons fréquemment qu'un même mot-mémoire peut contenir plusieurs informations distinctes; on se retrouve en quelque sorte avec un article structuré, dont il faut pouvoir atteindre les champs. Dans cette situation, nous emploierons des notations telles que (N.3). Celles-ci seront aussi utilisées lorsqu'on manipulera des structures de données à articles unitaillés dont les champs ne nous sont pas indifférents, ce qui permet notament d'écrire des algorithmes généraux valables pour les niveaux utilisateur et implémentation.

Puisque les pointeurs sont remplacés par des indices et que ceux-ci sont confondus avec les adresses, c'est ce dernier terme qui sera le plus usité et des expressions telles que P := NEXT(Q) + 1 (où P et Q sont des adresses) ne nous émouvrons donc pas du tout (alors que si P et Q étaient des pointeurs...).

Un dernier cas à traiter est celui où le contenu d'une variable x, logée dans une cellule aux caractéristiques indéfinies quant à la structuration et à la taille, est transféré dans une cellule d'adresse P, dont bien sûr la taille est implicitement la même que celle de la première. A cette occasion, c'est la notation

CELL(P) ← x

qui sera employée, en lieu et place de l'instruction

<u>for</u> i := 0 <u>to</u> c - 1 <u>do</u> MM(P+i) ← MM(Q+i)

(où c est la taille des deux cellules et Q l'adresse de celle où se trouve la variable x).

En conclusion, que nous employions les mots indice, référence, pointeur, tous ces vocables recouvrent une seule notion, à savoir la localisation ou l'adresse d'une valeur ou d'un champ dans un tableau.

CHAPITRE II

STRUCTURES CLASSIQUES

2.1. DIFFÉRENTES LISTES LINÉAIRES

Une pile est une liste telle que les ajouts et retraits d'éléments se font à une extrémité uniquement (le sommet; l'autre extrémité est appelée fond). Essentiellement, c'est un système "last-in-first-out".

Une file concrétise le système "first-in-first-out" : les ajouts se font à une extrémité (la fin ou la queue), les retraits à l'autre (le début ou le front).

Nous appellerons listes ouvertes les listes telles que les ajouts et les retraits s'effectuent indifféremment à l'une ou l'autre extrémité. Si les retraits ne sont permis que d'un côté, on parle de listes ouvertes en entrées; si cette restriction concerne les ajouts, on parle de listes ouvertes en sortie.

Ces différentes structures relèvent toutes de la liste linéaire, à savoir un ensemble de n ($n \geq 0$) articles dont les propriétés structurelles concernent les positions relatives des articles selon une seule dimension. Ainsi, pour $n > 0$, on a un premier article, un dernier article; pour $1 < k < n$, le kème article précède le (k+1)ème et suit le (k-1)ème.

Les opérations s'appliquant aux listes linéaires sont par exemple

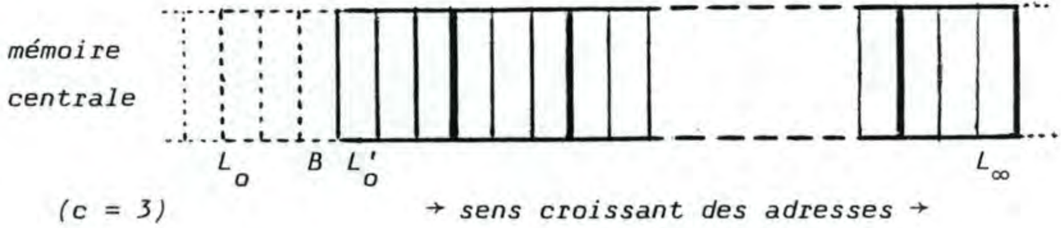
- accès au kème article
- retrait du kème article, insertion entre les kème et (k+1)ème articles
- fusion ou éclatement de listes
- etc.

Conventionnellement, si A désigne une liste linéaire et x une variable, la suite de symboles A <== x signifie qu'un nouvel article dont la valeur est celle de la variable x est ajouté à la liste, tandis que x <== A indique le retrait d'un article de la liste et l'affectation de sa valeur à la variable x. Si A est vide, cette opération ne peut évidemment pas être effectuée; sinon, l'article retiré est le premier accessible.

2.2. ALLOCATION SÉQUENTIELLE

La manière la plus simple d'installer une liste en mémoire centrale est de lui allouer, article par article, des cellules contiguës et ce séquentiellement. Si c est le nombre de mots-mémoire d'une cellule (et que les articles sont tous de même taille), l'adresse du (k+1)ème article est généralement celle du kème augmentée de c. La première adresse disponible est notée L'_0 et la dernière L_∞. Ce sont des constantes. Au lieu de L'_0, on préfère dans certains cas employer la constante $L_0 = L'_0 - c$, de sorte que quel que soit j, l'adresse de la jème cellule est $L_0 + c*j$. Enfin, B, adresse du mot-mémoire précedent immédiatement le premier de la liste, sera parfois utilisé ($B = L'_0 - 1$). On parle de l'adresse de base de la liste. On supposera toujours qu'il y a un nombre entier de cellules dans la zone qui leur est réservée, c'est-à-dire que c divise $L_\infty - B$. Il reste à insister sur le fait que le premier article de la liste n'occupe pas nécessairement la première cellule disponible; on s'en rendra vite compte en étudiant le cas des files. Lorsque l'on parlera du sommet ou du fond d'une pile ou du

front ou de la queue d'une file, il s'agira toujours de la cellule située au sommet ou au fond de la pile ou au front ou en queue de la file.



La zone de mémoire octroyée à une liste linéaire est limitée. Il se peut donc que l'ajout d'un élément ne soit pas possible, tout l'espace libre ayant été alloué à la structure. Overflow est le terme qui désignera cette situation. Inversément, x <== A est vide de sens si la liste A ne contient aucun article; dans ce cas, on parle d'underflow.

2.2.1. GESTION SÉQUENTIELLE D'UNE LISTE

2.2.1.1. PILES. Soit A une pile d'éléments de taille c (en nombre de mots-mémoire) et T une variable indiquant son sommet (ou plus exactement, conformément à ce qui a été dit ci-haut, l'adresse de la cellule se trouvant au sommet de A). T = L_o si et seulement si la pile est vide. La pile A est supposée pouvoir contenir un maximum de M articles. La zone où elle peut croître est donc constituée de M*c mots-mémoire, compris entre ceux d'adresses $L'_o = B + 1$ et $L_\infty = B + M*c$.

Selon ces conventions, l'algorithme correspondant à A <== x s'écrit :

(A.1) $T := T + c;$
<u>if</u> $T > L_\infty$
<u>then</u> *overflow*
<u>else</u> $CELL(T) \leftarrow x;$

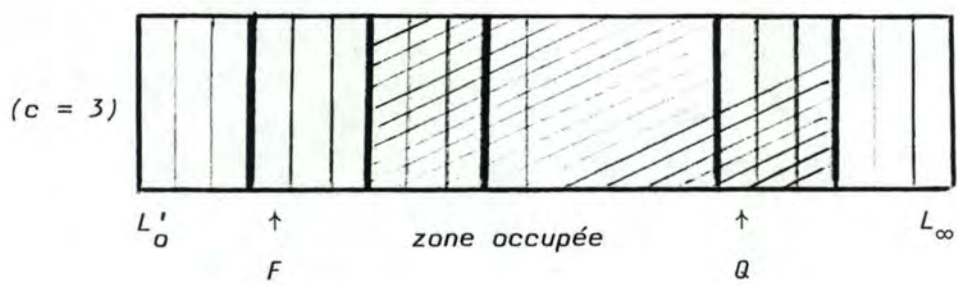
L'algorithme de retrait (x <== A) :

```
(A.2)  if T = L_o
       then underflow
       else begin
              x ← CELL(T);
              T := T - c
            end;
```

2.2.1.2. FILES. La gestion d'une file requiert deux variables F et Q indiquant respectivement le front et la queue de la file. Si on ajoute et retire des éléments de façon à ce qu'il y ait toujours au moins un élément dans la file A, on va voir F et Q prendre des valeurs de plus en plus grandes et une situation d'overflow pourra survenir alors qu'il y a encore beaucoup de place inoccupée... mais du côté du front. On contourne ce problème en décrétant que l'article suivant celui qui est logé dans la dernière cellule se trouve dans la première cellule. On adopte ainsi une configuration circulaire. Initialement, F = Q = L'_o. Aux opérations A <== x et x <== A correspondent alors les algorithmes suivants :

```
(A.3)  if Q > L_∞
       then Q := L'_o
       else Q := Q + c;
       if Q = F
       then overflow     (*)
       else CELL(Q) ← x;

(A.4)  if Q = F
       then underflow
       else begin
              if F > L_∞
              then F := L'_o
              else F := F + c;
              x ← CELL(F)
            end;
```

(*) En fait, il n'y a pas véritablement overflow, on pourrait ajouter un article; mais alors, la vacuité et la complétude de la file seraient signalées par la même condition F = Q.



2.2.1.3. LISTES OUVERTES. Elles aussi seront gérées circulairement. Les algorithmes (A.3) et (A.4) restent d'application pour l'insertion en queue ou le retrait au front. Les autres cas sont l'insertion au front et le retrait en queue :

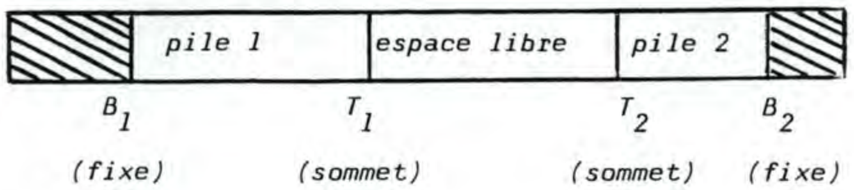
```
(A.5)  T := F;     { T est une variable intermédiaire }
       if T = L'o
       then F := L∞ - c + 1
       else F := F - c;
       if F = Q
       then overflow
       else CELL(T) ← x;

(A.6)  if Q = F
       then underflow
       else begin
              x ← CELL(Q);
              if Q = L'o
              then Q := L∞ - c + 1
              else Q := Q - c
              end;
```

2.2.1.4. REMARQUES. La situation d'underflow n'est pas une catastrophe; on peut imaginer que dans les algorithmes ci-dessus, "underflow" est un appel à une procédure dont le but est d'avertir que la liste est vide; on peut même concevoir une boucle de vidage de la liste dont la condition d'arrêt est justement cet underflow. Par contre, l'overflow est susceptible de provoquer la sortie en erreur du programme.

2.2.2. GESTION SÉQUENTIELLE DE PLUSIEURS LISTES

De nombreux programmes ont à gérer plus d'une liste. Fréquemment, d'ailleurs, ces listes sont des piles. Chacune d'elles a une taille dynamique et on ne lui fixe pas de taille maximale, d'autant moins qu'il est rare que toutes les piles atteignent leur taille maximale en même temps.

2.2.2.1. CAS DE DEUX PILES. La configuration à donner à la mémoire dans le cas de deux piles semble évidente (voir figure ci-dessous) : les deux piles croissent et se rétractent indépendemment l'une de l'autre, l'overflow n'arrivant que lorsque tout l'espace libre a entièrement été consommé ($T_1 > T_2$). La gestion d'un tel système est aisée à réaliser. Quand le nombre de piles à implémenter dépasse deux, il n'est plus possible d'adopter la même technique, dont les deux postulats sont primo que l'overflow n'arrive que lorsque l'espace libre est réduit au vide et secundo que le fond de chaque pile est à une adresse fixe. Pour satisfaire la première exigence avec plus de deux piles, il faut se passer de la seconde, ce qui aura pour conséquence que l'adresse de base des piles n'est plus constante.



2.2.2.2. CAS DE PLUSIEURS PILES. Penchons-nous sur le problème de la maintenance de n piles $A_1, \ldots, A_n$ en mémoire centrale vis-a-vis d'articles de même taille, c mots-mémoire. Nous aurons besoin de deux tableaux, $B(1..n+1)$ et $T(1..n)$ tels que $B(i)$ et $T(i)$ indiquent respectivement la base et le sommet de la ième pile. L'élément $B(n+1)$ est introduit pour harmoniser les notations et sa valeur est $L_\infty - c + 1$, adresse de la dernière cellule de la zone allouée aux n piles. $B(1)$ est l'adresse de la cellule précédant immédiatement cette zone, soit L_o. On supposera comme en 2.2. que c divise $L_\infty - B$. Initialement, $B(i) = T(i) = L_o$ $(1 \leq i \leq n)$. L'ajout d'un élément (contenu de la variable x) se fait selon l'algorithme suivant :

```
(A.7)  T(i) := T(i) + c;
       if T(i) > B(i+1)
       then overflow
       else CELL(T(i)) ← x;
```

Pour le retrait (avec affectation à la variable x) de la ième pile :

```
(A.8)  if T(i) = B(i)
       then uderflow
       else begin
              x ← CELL(T(i));
              T(i) := T(i) - c
            end;
```

p i l e 1 pile 2 ... p i l e i

L_o B L'_o — $B(1)$ — $T(1)$ $B(2)$ — $B(i)$ — $T(i)$ — $B(n+1)$ — L_∞

(c = 3)

Lorsqu'un overflow survient (et il est clair que selon nos hypothèses, la première insertion va en provoquer un — à moins que l'ajout ne se fasse à la $n$ème pile), nous avons la possibilité de reconfigurer la zone mémoire, en déplaçant les piles.

L'idée est que dès qu'on voudra ajouter un article à la ième pile ($i < n$), de manière à ce que sa taille soit plus grande qu'elle ne l'a jamais été, il y aura overflow. Dans cette situation, trois possibilités se présentent :

.a. *trouver le plus petit k tel que* - $i < k \leq n$
- $T(k) < B(k+1)$

si un tel k existe, faire

```
for j := T(k) + c - 1 downto B(i+1) + c do
    MM(j+c) ← MM(j);
for j := i + 1 to k do
    begin
    B(j) := B(j) + c;
    T(j) := T(j) + c;
    end;
```

sinon,

.b. *trouver le plus grand k tel que* - $1 \leq k < i$
- $T(k) < B(k+1)$

si un tel k existe, faire

```
for j := B(k+1) + c to T(i) - 1 do     (*)
    MM(j-c) ← MM(j);
for j := k + 1 to i do
    begin
    B(j) := B(j) - c;
    T(j) := T(j) - c
    end;
```

sinon,

.c. *si* $T(k) = B(k+1)$ *pour tout* $k \neq i$, *c'est qu'il n'y a plus de place.*

(*) Vu l'algorithme (A.7), T(i) a déjà été incrémenté.

Remarquons qu'il faut remanier l'algorithme (A.7) de façon à avoir quelque chose comme

```
(A.7') T(i) := T(i) + c;
       if T(i) > B(i+1)
       then overflow;     { appel ci-dessus }
       if <succès>
       then CELL(T(i)) ← x
       else <sortie en erreur>;
```

Il y a d'autres initialisations possibles. Par exemple, si on s'attend à ce que toutes les piles aient plus ou moins la même dimension, il est préférable de partitionner dès le départ l'espace libre en n zones de même longueur. On trouve les adresse de base de chaque pile par la formule

$$B(j) = T(j) = L_o + c * \left\lfloor \frac{(j-1)}{n} * \frac{(L_\infty - B)}{c} \right\rfloor .$$

Ne perdons pas de vue que nous travaillons avec des nombres entiers (d'où les symboles $\lfloor$ et $\rfloor$ indiquant une partie entière) et surtout qu'entre deux piles se trouve toujours un nombre de mots-mémoire multiple de c (soit un nombre entier de cellules). De toute façon, un bon démarrage permet d'éviter trop d'overflows en début du déroulement d'un programme, mais on arrive toujours à une situation générale indépendente du point de départ.

2.2.2.3. ALGORITHME DE GARWICK. Une amélioration plus substentielle consiste à trouver de la place pour plus d'un article lors de la reconfiguration de la mémoire. Cette opération demande en effet assez bien de temps, ce qui poussa Garwick à suggérer une complète restructuration de l'espace mémoire à chaque overflow, basée sur le changement de taille de chacune des piles depuis la dernière reconfiguration. L'algorithme qu'il propose utilise un

tableau supplémentaire, soit OLDT(1..n), qui retient les anciennes valeurs du tableau T. Les tableaux sont initialisés comme précédemment, avec de plus OLDT = T. Des variables locales sont également nécessaires : SUM et INC, ainsi que deux tableaux D et NEWB de même format que T.

La situation est la suivante : nous avons commencé l'exécution de l'algorithme (A.7) (ou plus exactement (A.7')) et un overflow survient. Après l'exécution de l'algorithme de Garwick, soit (A.7') se poursuivra (succès de la reconfiguration), soit nous saurons qu'il n'y a plus de place libre en mémoire.

```
procedure garwick;

sum := L∞ - B;    inc := 0;
for j := 1 to n do
     begin
     sum := sum - (T(j) - B(j));
     if T(j) > OLDT(j)
     then begin
          D(j) := T(j) - OLDT(j);
          inc := inc + D(j)
          end
     else D(j) := 0
     end;

{ sum = espace libre;  D = tableau des accroissements;

             n
inc =        Σ   D(j) }
            j=1

if sum < 0
then EXIT;  <insuccès>
{ else } α := (0.1/n)*(sum/c);
β := (0.9/inc)*(sum/c);
NEWB(1) := B(1);
:
```

```
for j := 2 to n do
    NEWB(j) := NEWB(j-1) + T(j-1) - B(j-1) +
               c*⌊α⌋ + c*⌊β*D(j-1)⌋;
reconfiguration;
for j := 1 to n do
    OLDT(j) := T(j);
```

Il faut maintenant donner la procédure "reconfiguration" :

```
procedure reconfiguration;

{ pour chaque pile, nous avons l'adresse de base pour la
  nouvelle configuration ainsi que l'adresse de base et
  le sommet actuels }

j := 1;
while j ≤ n do
      begin
      j := j + 1;
      if NEWB(j) < B(j)
      then begin
           δ := B(j) - NEWB(j);
           for k := B(j) + c to T(j) + c - 1 do
               MM(k-δ) ← MM(k);
           B(j) := NEWB(j);
           T(j) := T(j) - δ
           end;
      .
      .
      .
      .
      .
      .
```

```
      .
      .
      .
      .
      .
      .
if NEWB(j) > B(j)
then begin
        "trouver le plus petit r ≥ j et tel que
         NEWB(r+1) ≤ B k+1 "        (*)
        for h := r downto j do
            begin
            δ := NEWB(h) - B(h);
            for k := T(h) + c - 1 downto B(h) + c do
                MM(k+δ) ← MM(k);
            B(h) := NEWB(h);
            T(h) := T(h) + δ;
            end;
        j := k
        end
end;
```

(*) un tel r existe toujours si NEWB(n+1) = B(n+1).

Dans la procédure "Garwick", on calcule les nouvelles adresses de base des piles 2 à n. Est-on sûr qu'aucune de ces adresses ne va dépasser L_∞ ? On observe tout d'abord que quel que soit i compris entre 2 et n , on a NEWB(i) $\geq$ NEWB(i-1). Il suffit dès lors de s'assurer que le calcul d'un élément fictif NEWB(n+1) donne une valeur qui ne dépasse pas $L_\infty - c + 1$. Or on a :

$$
\begin{array}{l}
NEWB(n+1) = NEWB(n) + T(n) - B(n) + c*\lfloor\alpha\rfloor + c*\lfloor\beta*D(n)\rfloor \\
NEWB(n) = NEWB(n-1) + T(n-1) - B(n-1) + c*\lfloor\alpha\rfloor + c*\lfloor\beta*D(n-1)\rfloor \\
\quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
NEWB(1) = B(1)
\end{array}
$$

d'où

$$NEWB(n+1) = NEWB(1) + \sum_{i=1}^{n} (T(i)-B(i)) + n*c*\lfloor\alpha\rfloor + \sum_{i=1}^{n} c*\lfloor\beta*D(i)\rfloor$$

$$\leq B(1) + L_\infty - B - sum + c*n*(0.1/n)*(sum/c) + c*(0.9/inc)*(sum/c)*inc$$

$$= B - c + 1 + L_\infty - B = L_\infty - c + 1 .$$

Cet algorithme donne de bons résultats lorsque l'espace libre vaut environ la moitié de l'espace total. Son exécution requiert trop de temps lorsque la mémoire est en passe de se remplir, de sorte qu'on pourrait imaginer qu'il y a effectivement overflow quand, dans la procédure Garwick, le calcul de la quantité SUM donne un résultat inférieur à une certaine valeur Smin qui correspond à un taux de remplissage relativement élevé.

2.3. ALLOCATION CHAÎNÉE

Plutôt que de ranger en cellules contiguës les articles d'une liste (ce qui peut poser des problèmes de manipulation), on préférera souvent adjoindre à chacun d'eux un champ supplémentaire dont le contenu sera un pointeur indiquant l'adresse de l'article suivant. La valeur spéciale *nil* ou 0 est attribuée à l'indicateur du dernier article de la liste (1.3. et 1.4.4.). Relevons quelques points de comparaison entre l'allocation séquentielle et l'allocation chaînée :

- L'allocation chaînée requiert plus de place (un champ de plus par article), mais évite dans certaines situations des répétitions malvenues; de plus, ce pointeur peut être logé dans une partie de la cellule qui ne serait pas attribuée à l'article mais suffisament grande pour le contenir.
- Retraits et insertions d'éléments sont plus faciles et surtout peuvent s'opérer à n'importe quel endroit d'une

liste chaînée.

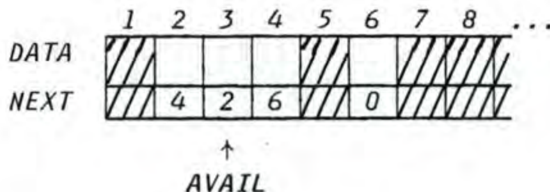
- Pour un accès au kème élément, par contre, l'allocation séquentielle est plus rapide; cependant, ce genre de requête n'est pas le plus fréquent.
- Fusion et éclatement de listes sont plus aisés dans un schéma chaîné.
- Les tables séquentielles ne permettent pas d'implémenter des structures plus complexes que les listes linéaires.

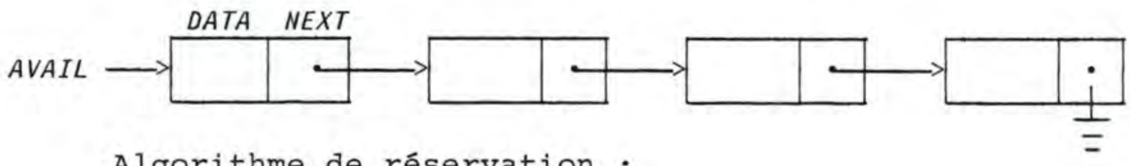
Conformément à ce qui a été dit en 1.4.4., nous modélisons la situation de la façon suivante : un tableau DATA et un tableau NEXT, de même dimension, sont à notre disposition; le premier élément de chacun d'eux est à l'indice 1. Pour un même indice i, DATA(i) et NEXT(i) constituent un élément de la structure.

2.3.1. LA LISTE LIBRE

Utiliser l'allocation chaînée impose généralement l'existence d'un mécanisme qui trouve l'espace libre pour un nouvel article à insérer. Les cellules libres sont regroupées en une liste spéciale, dite libre. Le champ adresse NEXT d'une cellule donnée de la liste libre contient l'adresse de l'élément suivant; c'est comme on l'a dit le nombre 0 qui correspond à la valeur *nil*. L'indice d'entrée (premier élément de la liste libre) sera désigné par AVAIL. Les deux opérations fondamentales sont la réservation d'une cellule dont l'indice est à rendre dans la variable x et la libération d'une cellule d'indice x, respectivement notées x <== AVAIL et AVAIL <== x. En début de l'exécution d'un programme utilisant ce genre de structure, il faut constituer la liste AVAIL. Ceci peut se faire en chaînant tous les emplacements susceptibles d'être alloués; AVAIL sera alors l'adresse du premier de ces emplacements et le dernier pointeur vaudra 0. La figure ci-dessous reprend un exemple selon nos notations et la représentation

classique qui lui correspond.





Algorithme de réservation :

```
(A.9)  if AVAIL = 0
       then overflow
       else begin
              x := AVAIL;
              AVAIL := NEXT(AVAIL)
            end;
```

Algorithme de libération :

```
(A.10) NEXT(x) := AVAIL;
       AVAIL := x;
```

Remarque : Comme annoncé, on se place dans le cas où tous les articles sont de même taille (ce qui nous permet d'utiliser le modèle des tableaux). Les problèmes occasionés lorsque cette hypothèse n'est pas vérifiée sont étudiés plus loin (chapitre III).

On se pose de nouveau la question de savoir quelle attitude adopter vis-à-vis d'une situation d'overflow (épuisement de la liste libre). S'il n'y a pas moyen de trouver d'autres cellules, le programme est condamné; sinon, il faut pouvoir trouver les emplacements libres (c'est-à-dire inaccessibles à partir du programme utilisateur) et reconstituer une liste libre. C'est la tâche du "garbage collector"

ou "ramasse-miette", qui est étudié plus en détail au chapitre IV.

De toute façon, voici une formule imaginée dans un contexte où une partie de la mémoire est allouée à des listes séquentielles dont on ne peut pas prévoir les tailles. Dans notre modèle, traduisons cette situation ainsi : les listes libre et chaînées se trouvent dans la première partie des tableaux DATA et NEXT, jusqu'à et y compris l'indice ZMAX. Les listes séquentielles occupent le fond des tableaux, depuis l'indice SEQMIN. Il peut donc exister un certain espace entre les adresses ZMAX et SEQMIN. Tant que ZMAX est inférieur à SEQMIN, tout va bien.

```
.a. initialement : AVAIL := 0;
                   ZMAX := 0

.b. réservation : if AVAIL ≠ 0
                  then begin
                       x := AVAIL;
                       AVAIL := NEXT(AVAIL)
                       end;
                  else begin
                       ZMAX := ZMAX + 1;
                       if ZMAX ≥ SEQMIN
                       then overflow
                       else x := ZMAX
                       end;
```

.c. si on tente de diminuer la valeur de SEQMIN au cours du programme, il faut s'assurer qu'on ne provoque pas un overflow.

.d. la libération est inchangée.

Remarquons encore que cette technique pourrait être utilisée même si SEQMIN était une constante car elle évite le chaînage initial des cellules libres.

2.3.2. IMPLÉMENTATION DES PILES, FILES, LISTES OUVERTES

2.3.2.1. PILES. Nous supposerons dans ce paragraphe que l'opération P <== AVAIL est toujours couronnée de succès. Pour des articles de même taille, divisés comme tantôt en deux champs DATA et NEXT, l'insertion au sommet d'une pile A (sommet repéré par une variable T) se fait très simplement via un pointeur auxiliaire P :

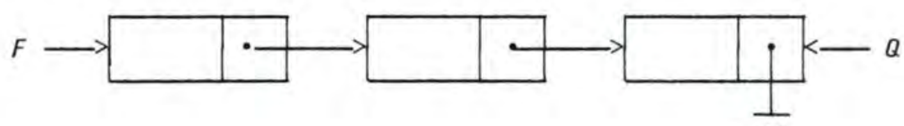
```
(A.11) P <== AVAIL;
       DATA(P) := x;
       NEXT(P) := T;
       T := P;
```

Le retrait d'un élément de la liste et l'affectation de sa partie DATA à une variable x se fait tout aussi simplement :

```
(A.12) if T = 0
       then underflow
       else begin
            P := T;
            T := NEXT(P);
            x := DATA(P);
            AVAIL <== P
            end;
```

Si T désignait précedemment le premier mot-mémoire de la dernière cellule utilisée dans une pile implémentée séquentiellement, il est maintenant un pointeur vers la dernière cellule de la structure mais la position de celle-ci vis-à-vis des autres est tout à fait sans importance.

2.3.2.2. FILES. Il nous faut maintenant deux pointeurs : F vers la cellule correspondant au front de la file (côté retraits) et Q vers la queue (côté ajouts).



Une insertion requiert les étapes suivantes :

```
(A.13) P <== AVAIL;
       DATA(P) := x;
       NEXT(P) := 0;
       NEXT(Q) := P;
       Q := P;
```

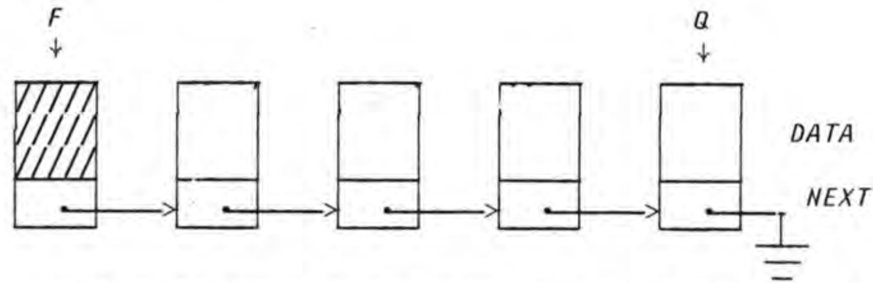
Si la file est réduite à un seul élément, on a Q = F et NEXT(F) = 0. Il n'est pas possible d'arriver à une telle situation à partir d'une file vide sans modifier (A.13) (sauf "chipotage"). Pour ne pas créer un cas particulier dans (A.13), le mieux est encore d'ajouter un élément supplémentaire du çoté du front. Initialement donc on a une file vide et F = Q, NEXT(F) = 0, DATA(F) est sans importance.

Le retrait d'un élément en début de file s'écrit alors

```
(A.14) if F = Q then underflow
       then underflow
       else begin
              P := NEXT(F);
              x := DATA(P);
              AVAIL <== F ;
              F := P
            end;
```

Remarque : En fait, on cherche à exprimer la condition de vacuité le plus simplement possible, par exemple Q = F, et ce sans influence sur les algorithmes. On a vu

que sans élément supplémentaire, le passage d'une file vide à une file à un élément pose un problème dans la mesure où F et Q soivent tous deux changer de valeur, ce qui n'est pas le cas lorsqu'on ajoute un élément à une liste non vide. L'uniformité étant un grand principe dans l'étude de ces algorithmes, cet artifice est souvent jugé indispensable.



2.3.2.3. LISTES OUVERTES. Les deux opérations qui s'ajoutent à la gestion d'une file pour la transformer en une liste ouverte sont le retrait en queue et l'ajout au front. Malheureusement, si cette dernière ne pose pas de problème délicat, le retrait d'un élément en queue de liste ne permet pas de retrouver l'élément "précédent" (à moins que de parcourir la liste du front à la queue en mémorisant les adresses des articles atteints, ce qui est long et signe d'une faiblesse certaine). Une solution plus élégante est celle du double chaînage, étudié dans le paragraphe 2.5.

2.4. LISTES CIRCULAIRES

Considérons l'implémentation chaînée d'une liste linéaire dont les éléments sont des articles à deux champs, DATA et NEXT. Le champ NEXT du dernier article vaut normalement 0. Si maintenant on force de pointeur à référencer le premier article de la liste, on donne naissance à une structure dont tous les éléments peuvent être atteints à partir d'un quelconque d'entre eux : c'est la propriété essentielle des listes circulaires. Pour accéder aux éléments d'une liste circulaire, il nous faut l'adresse de

l'un d'eux. Comme dans le cas des files, il s'avère intéressant pour ne pas dire indispensable, afin d'éviter les cas particuliers, d'inclure un élément <u>header</u> dans la liste dont l'adresse sera désignée par H. Si la liste est vide, on a NEXT(H) = H.

Le parcours d'une liste non vide s'effectue de la façon suivante, via un pointeur auxiliaire CURRENT :

```
(A.15) CURRENT := NEXT(H);
       while CURRENT ≠ H do
             begin
             < action sur DATA(CURRENT) >;
             CURRENT := NEXT(CURRENT)
             end;
```

Selon cet algorithme, le premier article atteint est celui désigné par NEXT(H) tandis que le dernier est celui dont le champ NEXT vaut H. Vis-à-vis de cette distinction, les trois opérations suivantes sont essentielles :

insertion en tête :

```
(A.16) P <== AVAIL;  { supposé couronné de succès }
       DATA(P) := x;
       NEXT(P) := NEXT(H);
       NEXT(H) := P;
```

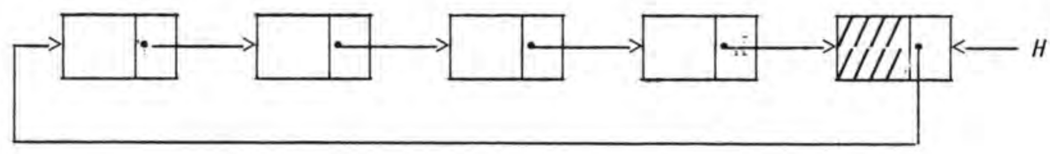
insertion en queue :

```
(A.17) P <== AVAIL;
       DATA(H) := x;
       NEXT(P) := NEXT(H);
       NEXT(H) := P;
       H := P;
```

retrait du premier article :

```
(A.18) if NEXT(H) = H
       then underflow
       else begin
            P := NEXT(H);
            x := DATA(P);
            NEXT(H) := NEXT(P);
            AVAIL <== P;
            end;
```

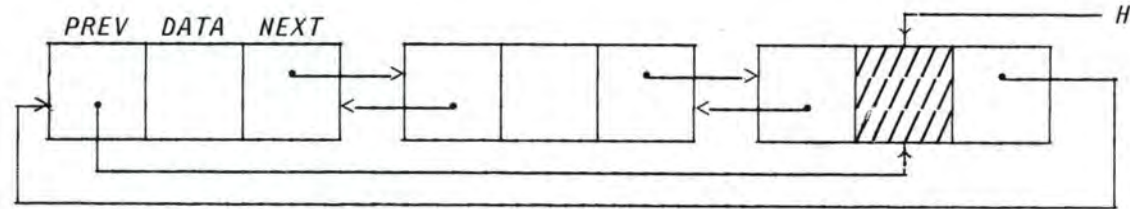
Ces trois opérations sont celles qui permettent de gérer une liste ouverte en entrée (voir 2.1.). En particulier, une liste circulaire peut être utilisée comme une pile ou comme une file, en combiant respectivement (A.16) et (A.18) ou (A.17) et (A.18).



2.5. DOUBLE CHAÎNAGE

La limitation essentielle des listes ainsi chaînées en mémoire centrale est qu'il n'est pas possible d'atteindre directement l'élément précédant un élément donné sans partir du début ou sans parcourir toute la structure (listes circulaires), en mémorisant les adresses succesives. Le problème ne se pose pas en allocation séquentielle car les cellules sont contiguës. La solution, déjà évoquée, consiste à ajouter à chaque élément un deuxième champ pointeur, soit PREV, indiquant l'élément précédent; NEXT continue à pointer sur l'article suivant. Bien sûr, le premier article a son champ PREV qui vaut *nil* et il en va de même pour le champ NEXT du dernier. Ces restrictions (plus exactements ces assymétries) peuvent être levées si, comme pour les listes

circulaires, on adjoint d'office un "header" à toute liste doublement chaînée. On a alors le schéma suivant (H est l'adresse du "header") :



La propriété essentielle des listes doublement chaînées est que si P est un pointeur vers un élément quelconque de la liste (y compris le header),

$$NEXT(PREV(P)) = P = PREV(NEXT(P)).$$

On n'a pas une aussi grande symétrie en l'absence de header, c'est pourquoi son usage est quasiment indispensable. Voyons rapidement comment s'effectuent ajouts et retraits.

Retrait d'un élément quelconque (sauf le header) référencé par un pointeur P (supposant que cet élément existe) :

```
(A.19) x := DATA(P);
       PREV(NEXT(P)) := PREV(P);
       NEXT(PREV(P)) := NEXT(P);
       AVAIL <== P;
```

Insertion d'un élément du côté "next" de l'article référencé par P :

```
(A.20) Q <== AVAIL;   { supposé couronné de succès }
       DATA(Q) := x;
       NEXT(Q) := NEXT(P);
       PREV(Q) := P;
       NEXT(P) := Q;
       PREV(NEXT(P)) := Q;
```

Pour une insertion du côté "prev", il suffit d'inverser partout NEXT et PREV.

L'efficacité des listes doublement chaînées se mesure dans d'autres opérations, telles que la suppression d'une telle structure et son retour à la liste libre ou l'insertion d'une liste L_2 devant une autre, L_1, dont voici les algorithmes :

(A.21)
```
P := AVAIL;
AVAIL := H;
NEXT(H) := P;
```

On suppose toujours que la liste AVAIL est gérée via le champ NEXT, le champ PREV pouvant avoir une valeur quelconque. Cet algorithme est également applicable aux listes circulaires.

(A.22)
```
{ les headers des listes L1 et L2 sont aux adresses H1 et
  H2 respectivement }

if NEXT(H2) ≠ H2
then begin
     PREV(NEXT(H1)) := PREV(H2);
     NEXT(PREV(H2)) := NEXT(H1);
     PREV(NEXT(H2)) := H1;
     NEXT(H1) := NEXT(H2);
     NEXT(H2) := H2;
     PREV(H2) := H2;
     end;
```

L'hypothèse d'articles de même taille est toujours d'application. Si tel n'est pas le cas, quelques mises au point sont nécessaires; on en trouvera l'essentiel dans le chapitre III où une liste doublement chaînée d'articles de taille variable est mise en oeuvre.

2.6. STRUCTURES ARBORESCENTES

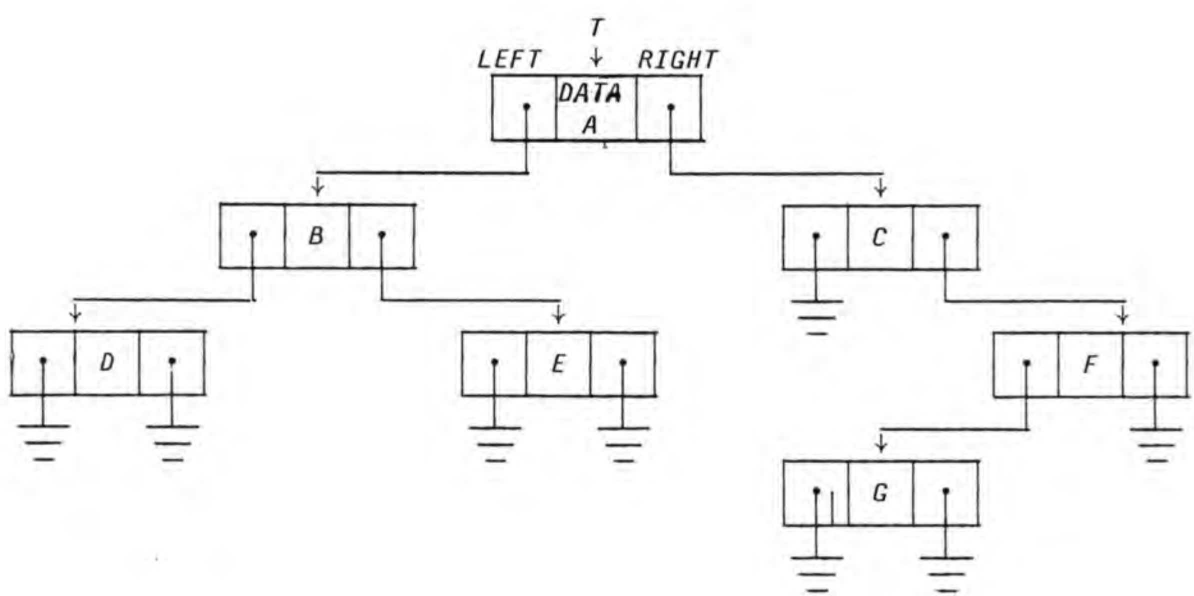
Soit T un ensemble fini non vide. Il s'agit d'un arbre si on y a distingué un élément particulier, appelé racine de T, et si les autres éléments de T sont partitionnés en m ($m \geq 0$) arbres $T_1, \ldots, T_m$ appelés sous-arbres de T.

La récursivité semble être une caractéristique intrinsèque des structures arborescentes; on ne s'étonnera donc pas du caractère récursif de la définition.

Les arbres binaires forment une catégorie d'arbres particulière et intéressante à plus d'un titre. On peut les redéfinir formellement comme ensembles finis, soit vides, soit contenant une racine et comprenant deux arbres binaires disjoints, appelés sous-arbres gauche et droit.

L'étude des arbres sort du cadre de cette étude, mais il serait peut-être bon de rappeler et synthétiser divers principes de représentation interne (toujours sous l'hypothèse d'articles de même taille).

La définition même suggère une représentation des arbres binaires qui utiliseraient deux pointeurs par articles, LEFT et RIGHT, et un pointeur auxiliaire d'entrée T permettant d'atteindre la racine. Si l'arbre est vide, T = 0. Sinon, la situation est schématisée comme suit :

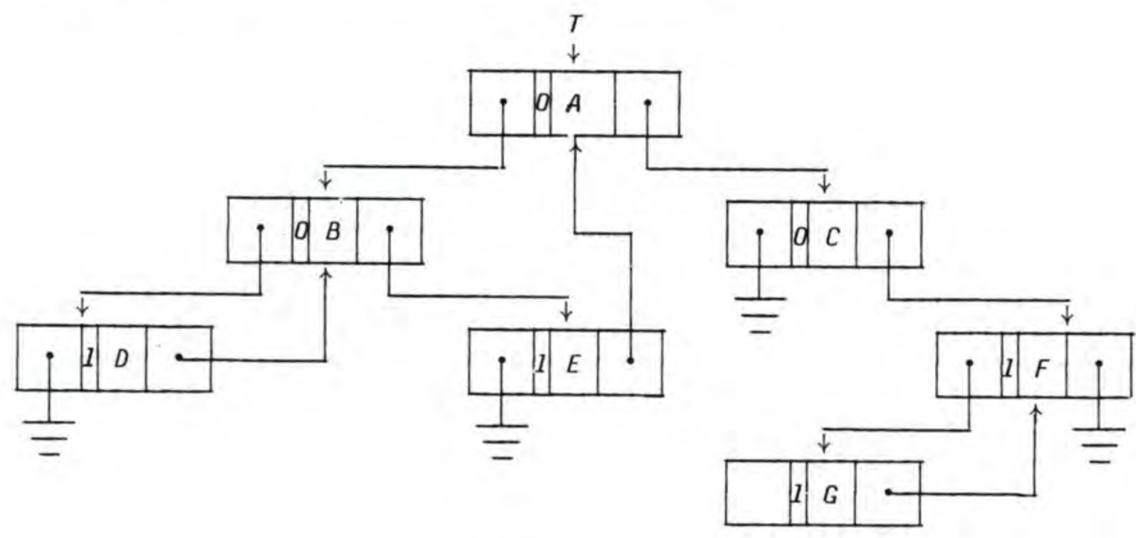


On remarque que la majorité des pointeurs valent *nil* , ce qui peut paraître peu économique. Perlis et Thornton eurent l'idée d'employer les champs RIGHT valant 0 à d'autres fins, tandis qu'un champ TT (un bit) signalait ce fait (TT = 0 : champ RIGHT "normal"; TT = 1 : champ RIGHT "réutilisé". A quoi leur servit l'espace récupéré ? Tout simplement à enchaîner les divers éléments de l'arbre pour en faciliter le parcours dans l'ordre infixé et pour éviter l'emploi d'une pile dans l'algorithme (récursif) réalisant ce parcours.

Ainsi, pour la première implémentation, on parcourt les éléments de l'arbre représenté ci-avant dans l'ordre D B E A C G F; l'algorithme de parcours classique s'écrit

```
(A.23) procedure visiter(T);

        if T ≠ 0
        then begin
                visiter(LEFT(T));
                < action sur DATA(T) >;
                visiter(RIGHT(T));
             end;
```

Si maintenant on adopte la représentation proposée par Perlis et Thornton, on obtient le schéma

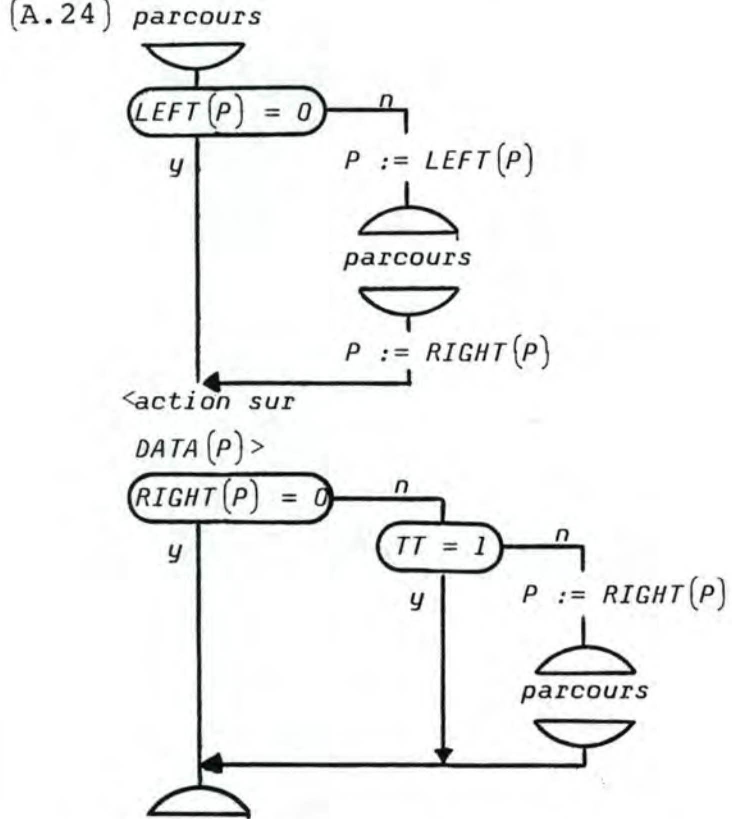


et l'algorithme qui permet le parcours de la structure dans le même ordre, s'il n'est pas plus simple que le précédent, ne nécessite pas l'implémentation d'une pile.

On introduit une procédure PARCOURS sans paramètre dont les spécifications sont, étant donnée l'adresse P de la racine de l'arbre non vide qu'on veut explorer,

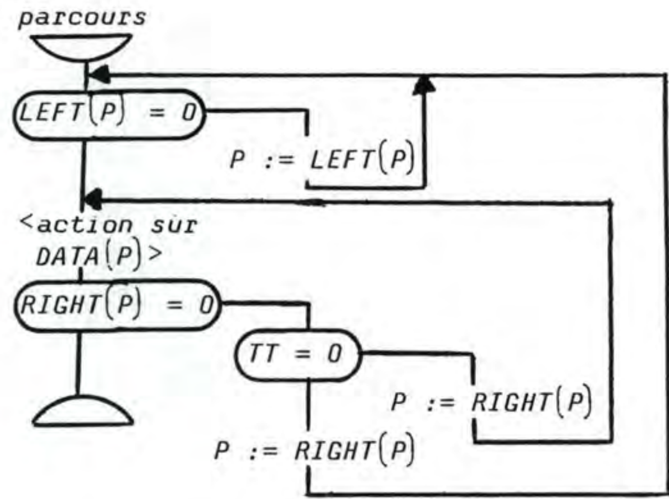
- réaliser le parcours infixé de l'arbre
- forcer P à l'adresse du dernier élément atteint dans cet ordre.

Ceci donne un premier algorithme récursif :



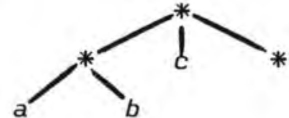
Le seul problème qui est à prendre en considération lorsqu'on tente de lever la récursivité est celui de l'adresse de retour. On arrive au résultat suivant :

(A.24')



2.7. STRUCTURES DE LISTE

Une <u>structure de liste</u> est une structure de données qui généralise les précédentes. Sa définition est récursive comme l'était celle des arbres : une structure de liste est une suite finie (éventuellement vide) d'atomes ou de structures de liste. Un <u>atome</u> est un élément sur la nature duquel on ne s'étend pas, du moment qu'il soit distinct d'une structure de liste. Les arbres sont des structures de liste. L'inverse n'est pas vrai : certaines structures de liste n'ont pas de représentation arborescente. Par exemple, la liste L = ((a,b),c,()) correspond à l'arbre

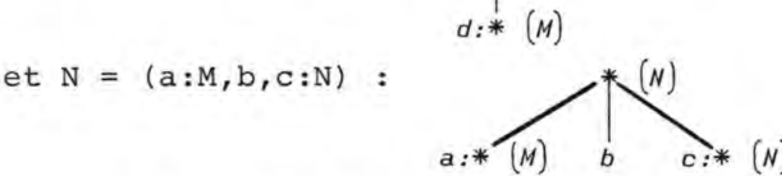


où l'astérisque désigne une liste dont les éléments "les fils". Mais des listes telles que M = (M) ou N = (M,b,N) ne sont pas représentables de cette manière, à cause de leur caractère récursif.

Si maintenant chaque élément d'une structure de liste qui n'est pas un atome est étiquetté, on peut donner une

représentation partielle de toute liste : M = (d:M)
devient

et N = (a:M,b,c:N) :



L'intérêt de ces étiquettes est qu'elles deviennent en quelque sorte porteuses d'information, tout comme l'astérisque pouvait être assimilé à une tête de liste.

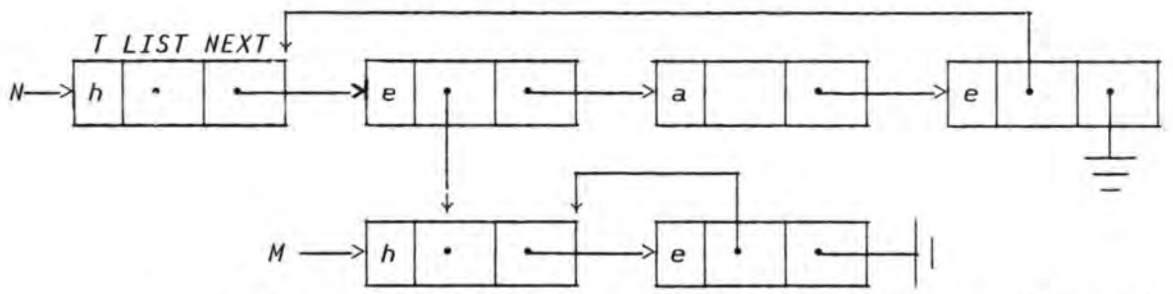
La représentation interne fait donc appel à un type d'article unique mais à trois fonctions différentes selon la valeur d'un champ T. Les autres champs sont NEXT et LIST, pointeurs.

- Si T vaut "h", l'enregistrement est en fait un header; il y en a un par structure de liste. Le champ NEXT indique le premier article de la structure (*nil* si cette structure est vide). Le champ LIST est inutilisé.

Les éléments d'une structure de liste sont enchaînés comme dans une liste linéaire par des pointeurs NEXT; ce sont soit des atomes, soit des étiquettes.

- Si T vaut "e", on a affaire à une étiquette et dans ce cas, le pointeur LIST référence le header de la liste qui correspond à cet élément.
- Si T vaut "a", l'élément est un atome. L'information est soit directement contenue dans la cellule (champ supplémentaire DATA ou utilisation du champ LIST), soit localisée dans un autre endroit de la mémoire et référencée par le pointeur LIST. Nous envisagerons de préférence cette dernière option.

La liste N donnée ci-dessus donne lieu à la représentation interne suivante (avec une représentation évidente de la liste M) :



Dans cette figure, N et M sont les pointeurs d'entrée des listes N et M respectivement. On confondra souvent d'ailleurs le nom de la liste avec le nom de son pointeur d'entrée.

Les cellules sont donc subdivisées en au moins trois champs : NEXT et LIST pointeurs, T indiquant la nature de l'article. Ajoutons-y un champ M, un bit, qui nous sera utile pour le marquage des cellules lors d'un parcours de la structure de liste.

Essentiellement donc, d'un point de vue interne, une structure de liste n'est rien de plus qu'une liste linéaire dont les éléments peuvent contenir des références ou des pointeurs vers d'autres structures de liste. Tout comme pour les listes linéaires, on peut envisager d'enchaîner les éléments en structure de liste circulaire ou doubement chaînée.

Deux points importants doivent être mis en évidence : le parcours d'une structure de liste et l'abandon d'une cellule. Pour le premier, donnons l'algorithme suivant, classique et récursif, qui permet de parcourir entièrement une structure de liste (à chaînage simple) sans boucler :

```
(A.25) procedure parcours P ;   { P : pointeur d'entrée }

          if P ≠ 0
          then if M(P) = 0
               then begin
                    M(P) := 1;         { marquage de la cellule }
                    if T(P) = "e" then parcours(LIST(P));
                    parcours(NEXT(P));
                    end;
```

Pour ce qui est de l'abandon par le programme utilisateur d'une cellule référencée par un pointeur P, il ne suffit plus de programmer AVAIL <== P comme précédemment. En effet, étant donné la capacité d'une structure de liste à grandir et s'amenuiser de façon imprévisible, il n'est pas toujours aussi simple qu'il n'y paraît d'affirmer qu'une cellule est effectivement superflue : d'autres pointeurs peuvent la référencer. Ce problème crucial est l'objet du chapitre IV.

Enfin, avant de clôturer définitivement ce chapitre, signalons que le langage LISP est entièrement basé sur les structures de liste (un peu différentes de ce qui a été exposé ici). On trouvera en annexe un aperçu de l'implémentation de ce langage.

CHAPITRE III

ALLOCATION DYNAMIQUE

3.1. INTRODUCTION

Les applications ne font pas nécessairement appel à des structures où tous les articles auraient la même taille. On peut se ramener à une telle situation, soit en imposant comme taille de chaque article celle du plus grand (ce qui n'est pas très économique), soit au contraire en réservant pour chaque article une cellule plutôt petite, quitte à en enchaîner plusieurs pour un élément dont la taille dépasserait la capacité d'une cellule.

Néanmoins, cette philosophie n'est pas raisonnablement applicable pour certaines applications et l'on préfère que les articles de taille variable se partagent une zone de mémoire commune. Sont alors nécessaires des algorithmes de réservation et libération de blocs-mémoire (ensemble de mots-mémoire consécutifs) de taille tout à fait aléatoire. Le terme cellule sera abandonné dans ce chapitre au profit du mot bloc, plus conforme aux dénominations usuelles. Si les tailles des articles sont variables, il en va de même pour leur type et il devient difficile d'utiliser le modèle des tableaux comme dans le chapitre précédent. Cependant, comme nous allons nous intéresser de près au contenu de certains mots-mémoire, c'est une notation du même type qui va

être employée. Il y a donc un décalage entre les structures manipulées (ensembles de mots-mémoire) et les notations s'y rapportant (parties de mots-mémoire).

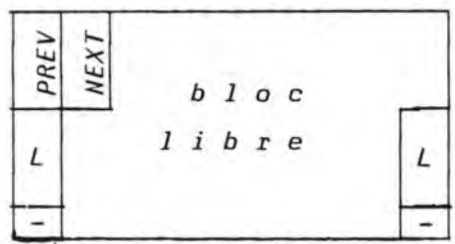
3.2. La liste libre

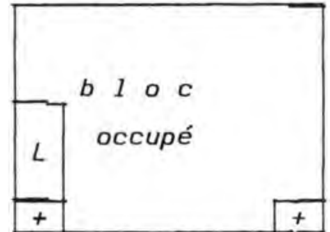
3.2.1. Structure des blocs

A un moment donné, la zone de mémoire où évoluent les structures de données se présente comme une succession de blocs de taille quelconque, certains occupés par de l'information "en usage" (blocs "pleins"), d'autres non (blocs "vides"). On peut voir ces derniers comme des trous dans l'espace-mémoire.

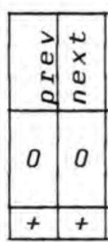
Le premier problème qui se pose est celui de la représentation de cette zone et plus particulièrement de cet espace libre dispersé en trous. Une solution très simple est évidemment d'enchaîner ces trous en une liste libre (voir chapitre II). Les blocs vides peuvent se succéder dans l'ordre de leur taille (des plus grands aux plus petits et vice-versa), ou de leur adresse, ou dans un ordre quelconque, en chaînage simple ou double. Une variable AVAIL donne l'adresse du premier bloc de la liste libre. Chaque bloc de la liste devra être suffisamment grand pour contenir une ou deux adresses et un indicateur de sa taille.

Pour des raisons de performance, et également pour faciliter la libération (3.2.3.), on adopte la structure suivante pour chaque bloc (double chaînage, ordre quelconque pour la liste libre) :





Le souci d'harmoniser les notations conduit à introduire un bloc supplémentaire, dont l'adresse est AVAIL et dont la structure est simplement celle-ci :



Si P est l'adresse d'un bloc libre, les différents champs seront ainsi atteints :

- S(P), L(P), PREV(P) pour le premier mot-mémoire, représentant respectivement un indiacteur ("-" en l'occurence), la longueur du bloc (en nombre de mots-mémoire) et l'adresse du bloc libre précédent;
- NEXT(P+1) pour le deuxième mot-mémoire : adresse du bloc libre suivant;
- L(P+l-1) et S(P+l-1) pour le dernier mot-mémoire (donc la longueur est ici supposée être l), reprenant la longueur et l'indicatif.

Similairement, pour un bloc plein, on aura les champs L(P), S(P) et S(P+l-1). En toute généralité, remarquons que l'expression P + l - 1 devrait être remplacée par P + L(P) - 1.

3.2.2. RÉSERVATION

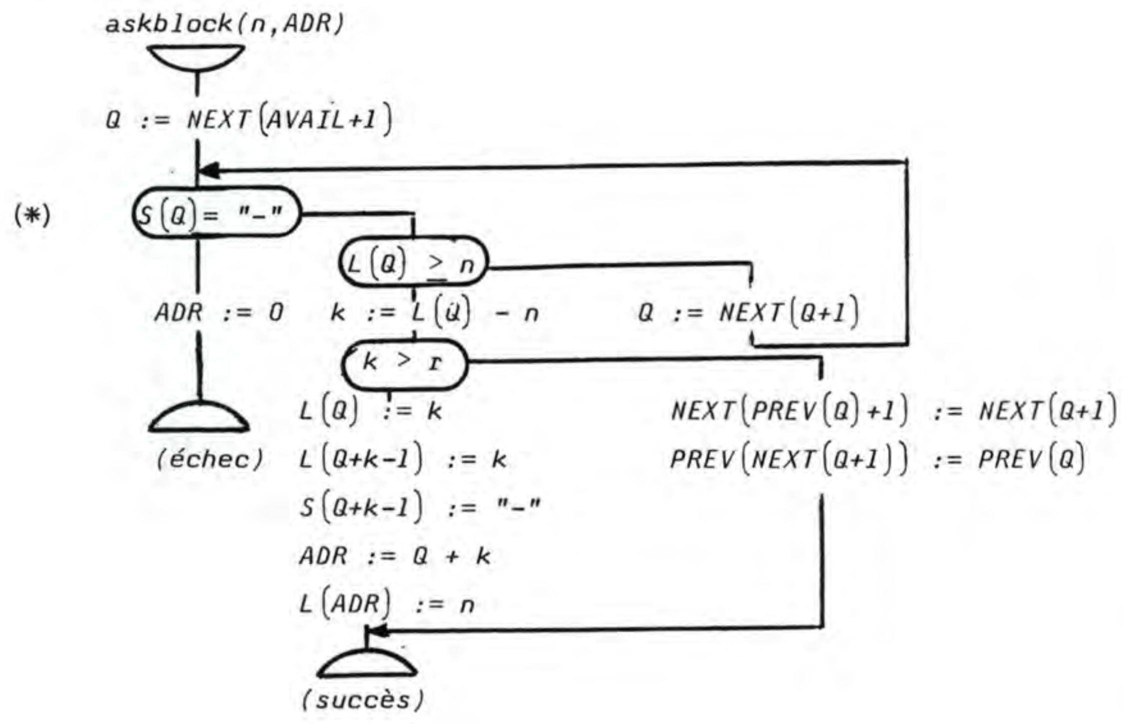
Il s'agit de mettre au point un algorithme permettant de trouver et de réserver un bloc libre de n mots-mémoire au moins. Trois politiques sont possibles, connues sous les noms "best-fit", "first-fit" et "worst-fit". La première consiste à réserver un bloc dont la taille est la plus proche de n (au moins égale !), la deuxième prend le premier bloc de la liste qui convient et la troisième, le bloc dont la taille est maximale.

A priori, la méthode "best-fit" semble la plus inté-

ressante, car elle gaspille peu de mémoire. Cependant, elle provoque un phénomène de "gruyérisation" pour le moins indésirable : prolifération de trous minuscules inutilisables. De plus, elle exige un parcours complet de la liste (à moins qu'elle ne soit "triée"), ce qui demande plus de temps. La méthode "worst-fit" est parfois utilisée, notament lorsque la taille des blocs à réserver est considérable, par exemple lors de l'allocation de mémoire par le système aux processus. Son avantage est que les résidus sont encore de grands blocs. Actuellement, en tous cas pour l'application signalée, la pagination rend désuète une telle méthode. Le plus généralement, c'est la méthode "first-fit" qui donne les meilleurs résultats.

Quelle que soit la politique envisagée, il faut choisir un bloc de taille m ($m \geq n$) et réduire celle-ci à $m - n$. Si $m = n$, on doit ôter le bloc de la liste libre. Si $m - n = 1$, il y a un problème car le mot-mémoire subsistant après l'allocation des n autres est incapable, à lui tout seul, de former un bloc libre. On l'inclut alors automatiquement dans le bloc réservé, ce qui ne pose pas de difficulté puisque, via le champ L, le programme utilisateur qui a fait la demande peut contrôler quelle est la longueur du bloc qu'on lui a effectivement réservé. De façon générale, on peut déterminer un nombre r tel que si $m - n \leq r$, alors le bloc entier est réservé (pas de bloc libre dont la longueur serait inférieure ou égale à r).

Supposons que le programme utilisateur ait besoin d'un bloc de n mots-mémoire : il fait appel à une procédure ASKBLOCK à deux paramètres, qui sont respectivement le nombre de mots-mémoire nécessaires et une référence ADR passée par variable qui indiquera, en cas de succès, l'adresse du bloc réservé; en cas d'échec, la valeur 0 est retournée. La politique suivie est celle du "first-fit".



(*) S(Q) = "-" signifie qu'on a parcourut la liste libre sans avoir trouvé de bloc suffisament long.

3.2.3. Libération

Lorsqu'un bloc occupé est abandonné, il vaut mieux le rendre à la liste libre qui, elle, a tendance à s'épuiser. Une possibilité est de ne rien faire directement et de partir à la recherche des blocs abandonnés au moment où une requête à la liste libre ne peut être satisfaite. Cette politique, apparentée au garbage collector, n'est cependant pas conseillée dans ce cas. D'abord, parce que la récupération s'accomode mieux de cellules unitaillées; ensuite, parce que l'exécution du "ramasse-miettes" ralentit celle du programme dans des proportions insoutenables; enfin, parce que la libération immédiate d'un bloc dont on ne se sert plus permet éventuellement la création immédiate d'un bloc libre plus grand, par adjonction du bloc libéré à ses voisins, s'ils sont libres. C'est le caractère immédiat qui est important,

comme l'illsutre le cas suivant:

- libération immédiate :
  - a) premier moment : libération d'un bloc abandonné et fusion avec le voisin, libre.
  - b) second moment : réservation d'un bloc.
- libération retardée :
  - c) premier moment : un bloc libre et un bloc abandonné mais non libéré.
  - d) deuxième moment : réservation d'un bloc.
  - e) troisième moment : libération par garbage collector ou technique apparentée.

a)	/////	*libéré*	*libre*		//
b)	/////	/////////	*libre*		//
c)	/////	*non restitué*	*libre*		//
d)	/////	*non restitué*	////////	*libre*	//
e)	/////	*libre*	////////	*libre*	//

Si le bloc libéré est coincé entre deux blocs occupés, il faut l'ajouter à la liste libre; si un des blocs adjacents est libre (et l'autre pas), on agrandit celui-là; s'ils sont tous les deux libres, on aura en fait un bloc libre de moins dans la liste, mais beaucoup plus grand. Dans tous les cas où il y a fusion de blocs libres, celui qui se trouvait déjà dans la liste libre en est retiré et on insère un nouveau bloc, plus grand. Notons que si la zone où on travaille est comprise entre les mots-mémoire d'adresses L_0 et L_∞ inclus, $S(L_0-1) = S(L_\infty+1) =$ "+".

```
procedure FREEBLOCK(ADR);

if S(ADR-1) = "-"        { le bloc précédent est libre }
then begin
        Q := ADR - L(ADR-1);
        NEXT(PREV(Q)+1) := NEXT(Q+1);
        PREV(NEXT(Q+1)) := PREV(Q);
        L(Q) := L(Q) + L(ADR);
        ADR := Q
     end;
Q := ADR + L(ADR);
if S(Q) = "-"            { le bloc suivant est libre }
then begin
        NEXT(PREV(Q)+1) := NEXT(Q+1);
        PREV(NEXT(Q+1)) := PREV(Q);
        L(ADR) := L(ADR) + L(Q)
     end;
L(ADR+L(ADR)-1) := L(ADR);
S(ADR+L(ADR)-1) := "-";
S(ADR) := "-";
NEXT(ADR+1) := NEXT(AVAIL+1);
PREV(ADR) := AVAIL;
NEXT(AVAIL+1) := ADR;
PREV(NEXT(AVAIL+1)):= ADR;
```

3.3. LE "BUDDY SYSTEM"

La représentation binaire des nombres a incité certains chercheurs à développer une technique d'allocation dynamique particulière. La longueur de chacun des blocs est une puissance de 2 (1, 2, 4, 8, etc), et la zone elle-même est supposée avoir une longueur de 2^m mots-mémoire; nous conviendrons qu'elle s'étend des adresse 0 à $2^m - 1$. Si la zone que demande le programme utilisateur n'est pas de longueur 2^k pour un certain k, on lui adjoint d'office autant de mots-

mémoire qu'il n'en faut pour arriver à une telle longueur. En tête de chaque bloc, un bit (champ T) est réservé pour la gestion du système : il est à 1 si le bloc est libre, à 0 sinon. Le principe de fonctionnement du "buddy system" est simple : on alloue des blocs de longueurs 2^k; si aucun bloc n'a cette taille, on "éclate" en deux un bloc de longueur $2^{(k+1)}$: on dit qu'ils donnent naissance à deux "buddies" (jumeaux). Plus tard, si ces deux blocs redeviennent libres, ils sont fusionnés en un seul. Ce procédé peut se maintenir indéfiniment.

Cette méthode, quand elle s'applique, est très rapide. Son efficacité réside en ce que connaissant l'adresse d'un bloc et sa taille, on connaît immédiatement l'adresse de son jumeau (le bloc de même taille qui, fusionné au premier, rendra le bloc dont tous deux sont issus). Justifions cela : l'adresse d'un bloc de taille 2^k est un multiple de 2^k ce qui, du point de vue binaire, signifie qu'il y a au moins k "zéros" à droite. C'est évidemment vrai pour la zone toute entière; supposons que cela soit vérifié pour un bloc de longueur $2^{(k+1)}$: son adresse binaire à la forme

$\text{xx....xx}\underbrace{\text{0...0}}_{k+1}$, où x représente indifféremment 1 ou 0.

Quand il éclate, les adresses des deux blocs issus sont

$\text{xx....xx}\underbrace{\text{00...0}}_{k+1}$ et $\text{xx....xx1}\underbrace{\text{0...0}}_{k}$, ce qui suffit.

Donc, de façon générale, si $\alpha_k(\beta)$ désigne l'adresse du jumeau d'un bloc de taille 2^k et d'adresse β, on a

$$\alpha_k(\beta) = \begin{cases} \beta + 2^k \text{ si } \underline{\text{mod}}\ 2^{(k+1)} = 0 \\ \beta - 2^k \text{ si } \underline{\text{mod}}\ 2^{(k+1)} = 1 \end{cases}$$

et ici le test $\beta\ \underline{\text{mod}}\ 2^{(k+1)}$ est un test sur le (k+1)ème bit.

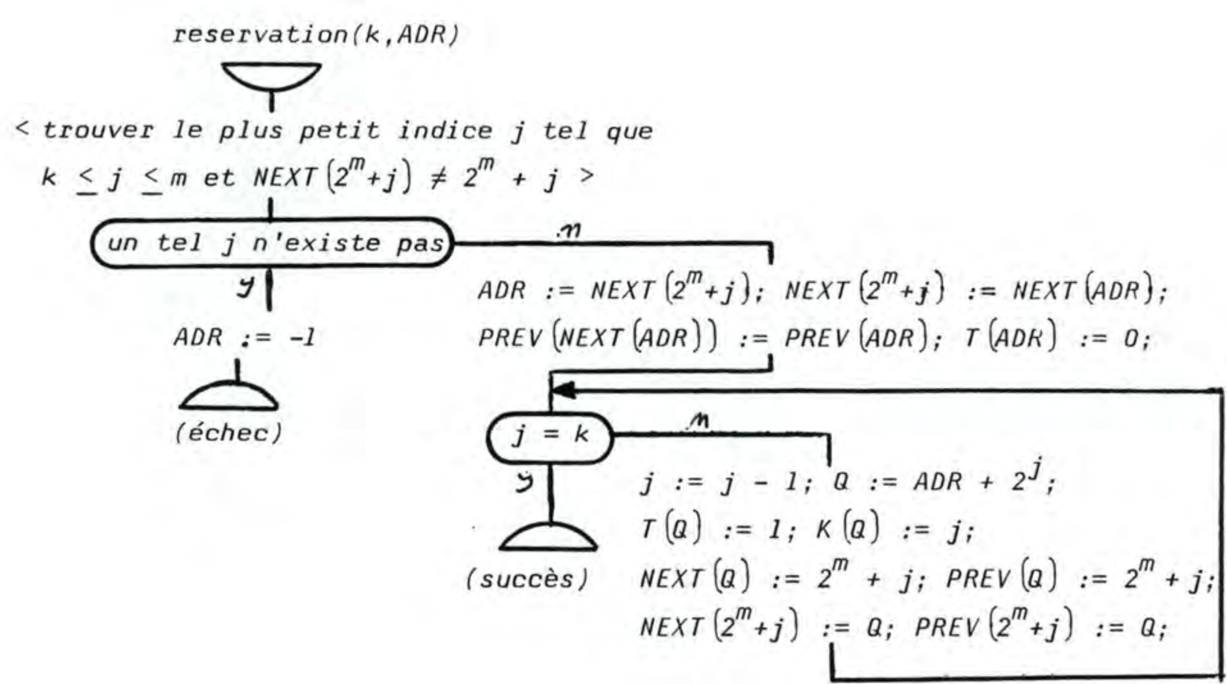
En plus du champ T (dont la valeur est alors 1 au lieu de 0), tout bloc de taille 2^k contient les deux champs références classiques PREV et NEXT ainsi qu'un champ K dont la valeur est k. Tous ces champs sont supposés pouvoir tenir dans un seul mot-mémoire.

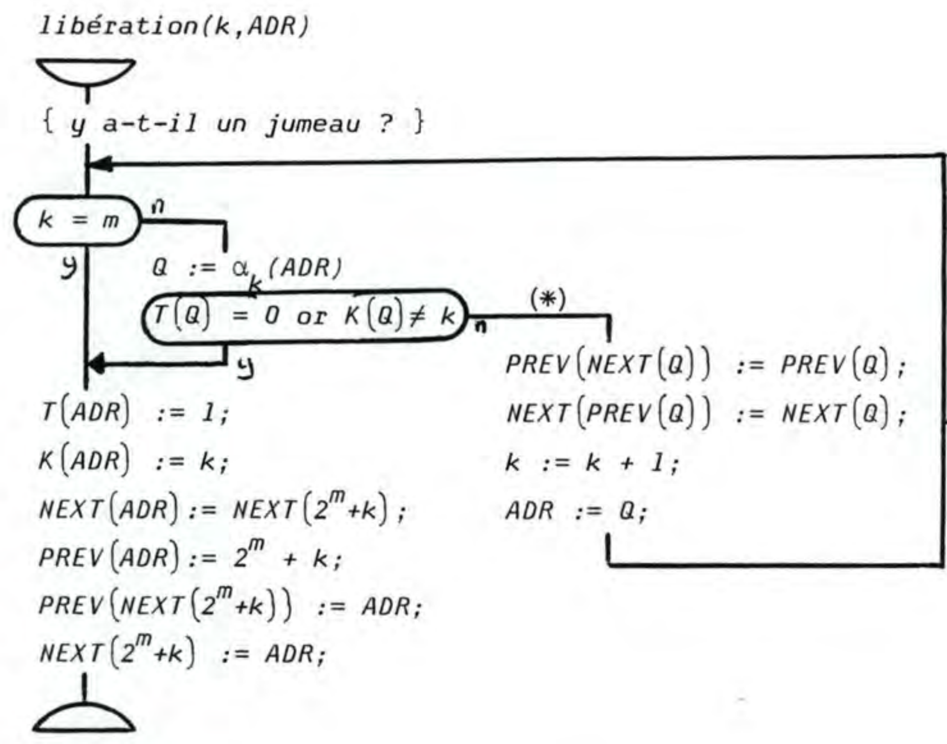
Les mots-mémoire 2^m à $2^m + m$ sont utilisés pour être les têtes des listes doublement chaînées des blocs libres de longueur 1 à 2^m respectivement : on y distingue deux champs NEXT et PREV tels que $NEXT(2^m+i)$ est l'adresse du premier bloc de la liste libre des blocs de longueur 2^i et $PREV(2^m+i)$ celle du dernier.

Initialement, on a la situation que voici :

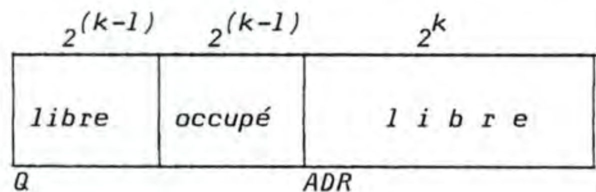
$NEXT(2^m) = PREV(2^m) = 0;$
$NEXT(0) = PREV(0) = 2^m;$
$T(0) = 1;\ K(0) = m;$
pour tout $0 < k \leq m,\ NEXT(2^m+k) = PREV(2^m+k) = 2^m + k.$

Regardons maintenant les algorithmes que l'on peut proposer pour la réservation et la libération d'un bloc; le programme utilisateur fait appel au gérant du "buddy system" via deux procédures, dont les paramètres sont k et ADR, correspondant le premier à la taille du bloc (2^k) et le second à son adresse (inconnue au départ lors de la réservation).





(*) Le test K(Q) = k est essentiel, car le jumeau du bloc qu'on traite pourrait avoir éclaté en deux blocs de taille $2^{(k-1)}$ dont le premier serait libre et le second occupé, comme l'illustre la figure ci-dessous.



3.4. LE THÉORÈME DES 50 POURCENTS

Il n'est pas facile d'analyser ces algorithmes d'un point de vue de leur complexité temporelle et surtout de leur dégradation, du temps pendant lequel ils se déroulent sans problèmes, les taux optimaux de remplissage de la mémoire, etc... surtout que les cas de figure sont nombreux et peuvent "convenir" particulièrement bien ou mal à un algorithme donné. Néanmoins, il est une règle qu'on a mise en évidence digne d'être citée; elle se rapporte aux deux

premières procédures de ce chapitre, ASKBLOCK et FREEBLOCK :

> "Si ces deux algorithmes sont utilisés continuellement de façon à ce que le système tende vers un équilibre où il y aurait en moyenne N blocs réservés, chacun d'eux ayant une durée de vie indépendante des autres, et où la quantité K (voir ASKBLOCK) prend une valeur supérieure à r avec la probabilité p, alors le nombre moyen de blocs libres tend approximativement vers (1/2)pN".

Le nom 50 pourcents vient de ce que si p est proche de 1 (ce qui arrive si r est très petit et si les tailles des blocs libres sont rarement égales d'un bloc à l'autre), le nombre de blocs libres est plus ou moins la moitié du nombre de blocs occupés.

On peut aisément comprendre cette règle. Considérant une configuration mémoire quelconque, les blocs occupés se répartissent en trois catégories, selon qu'ils ont pour voisins deux, un ou aucun blocs libres. La libération d'un bloc de la première catégorie diminue d'une unité le nombre de blocs libres; ce nombre est inchangé lors de la libération d'un bloc de la deuxième catégorie; enfin, il augmente lorsque l'on libère un bloc du troisième type. Notons A, B et C les nombres de blocs de chacune de ces catégories, et N (respectivement M) le nombre de blocs occupés (respectivement libres). On a donc :

$$N = A + B + C$$
$$M = (1/2)(2A + B + \varepsilon).$$

ε vaut 0, 1 ou 2 selon les conditions aux limites de la zone-mémoire, quantité négligeable si M et N sont grands. Il suffit de considérer le schéma ci-dessous pour se convaincre de la validité de la deuxième formule :

libre	1	libre	2	3	3	2	libre	2	2	libre	1	libre

L'hypothèse d'un système en équilibre conduit à dire que la probabilité que M croisse d'une unité vaut la probabilité que M décroisse d'une unité. Or M croît d'une unité à la libération d'un bloc de la troisième catégorie et M décroît d'une unité à la libération d'un bloc de la première catégorie, et lorsque la réservation d'un bloc se fait au détriment d'un bloc libre complet, trop petit pour être scindé en un bloc occupé et en un bloc libre, ce qui arrive pour la fraction (1-p) des blocs occupés.

Cette constatation se traduit par la formule

$$C = A + (1-p)N$$

d'où on tire succesivement (négligeant ε)

$$N = 2A + B + (1-p)N$$
$$2A + B = pN$$
$$M = (1/2)pN.$$

CHAPITRE IV

TECHNIQUES DE RÉCUPÉRATION

4.1. INTRODUCTION

Lorsque les premiers langages basés sur les structures de liste ont été implémentés, il a fallu résoudre le problème de la récupération de cellules "inaccessibles". L'intérêt de cette récupération s'est considérablement accru depuis que les langages offrent eux-mêmes la possibilité de gérer de telles structures de données (variables structurées, références, pointeurs). On sait par exemple que 10 à 30 % du temps d'exécution de grands programmes LISP sont perdus dans la récupération, ce qui pose également des problèmes pour les applications en temps réel.

Une cellule n'est plus accessible lorsqu'aucun des pointeurs ou des références accessibles ne la désigne. Si ces cellules ne sont pas rendues directement à l'allocateur, elles encombrent la mémoire et une situation d'overflow peut survenir : c'est le ramasse-miettes (récupérateur, garbage collector) qui a la tâche de retrouver ces cellules et de les restituer à l'espace libre.

L'action d'un récupérateur comprend généralement deux phases :

a) identification de l'espace-mémoire inutilisé, inaccessible;
b) restitution de cet espace-mémoire à l'espace libre de

l'utilisateur.

La phase a) est elle-même réalisable de deux façons :

a1) utilisation de compteurs de références (une cellule dont le compteur est à zéro n'est pas référencée) ou

a2) marquage des cellules accessibles (les autres étant donc inaccessibles).

La phase b) se conçoit elle-aussi de deux manières :

b1) gestion d'une liste libre ou

b2) compactage des cellules utilisées en un "tas" poussé vers une extrémité de la zone-mémoire réservée aux structures dynamiques, l'autre extrémité constituant un réservoir de cellules. Il y a plusieurs types de compactage : la linéarisation rend généralement contiguës les cellules qui au départ s'enchaînaient; le glissement déplace les cellules vers la fin du tas sans en changer l'ordre; enfin, le compactage arbitraire ne s'aligne sur aucune politique particulière.

4.2. RÉCUPÉRATION DE CELLULES UNITAILLÉES

4.2.1. MARQUAGE

Les cellules LISP illustrent les problèmes rencontrés lors du marquage des cellules unitaillées d'une structure de liste (voir en annexe un aperçu de l'implémentation de la structure interne de ce langage). Chacune d'elles est divisée en deux champs LEFT (car) et RIGHT (cdr) qui contiennent des pointeurs vers d'autres cellules ou vers des atomes (cellules spéciales ne contenant pas de pointeur). A côté de ceux-ci se trouvent encore deux champs booléens (bits) permettant le premier, A, de distinguer les cellules atomiques (A = 1) des autres (A = 0) et le second, M, de marquer (M := 1) les cellules atteintes lors du parcours des différentes structures de liste.

La représentation interne d'une structure de liste de

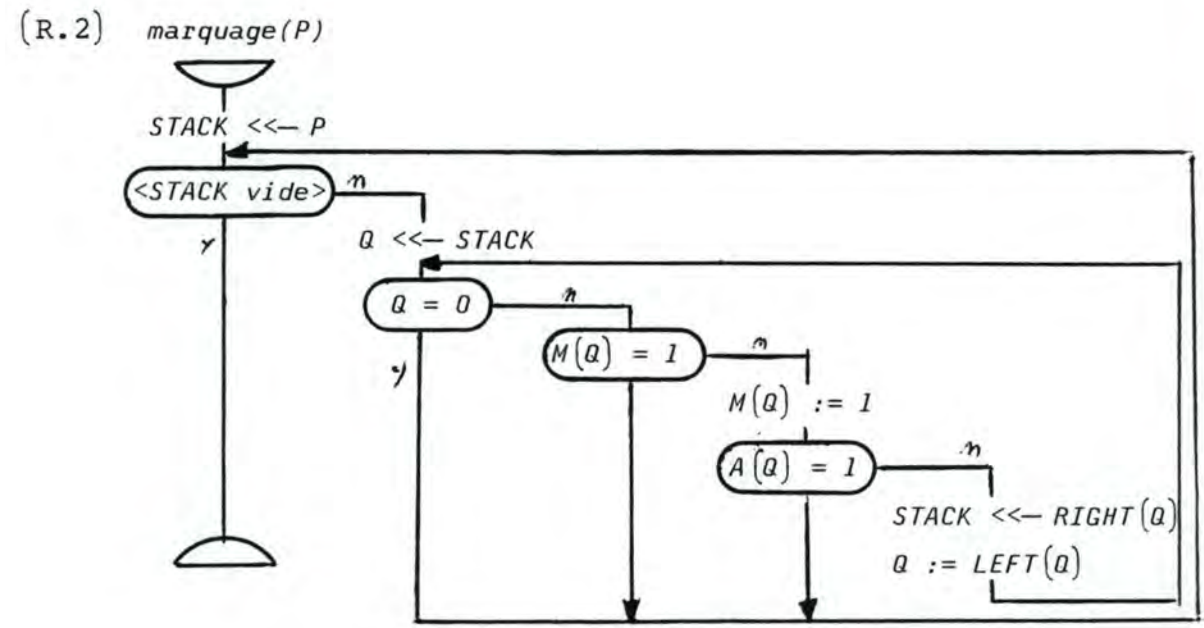
ce type diffère donc quelque peu de celle donnée dans le chapitre II.

Le premier algorithme qui vient à l'esprit pour le marquage des listes LISP est une version légèrement modifiée de l'algorithme (A.25) du chapitre II. P désigne une adresse et initialement celle-ci est celle de la première cellule de la liste :

(R.1)
```
procedure MARQUAGE(P);

if P ≠ 0
then if M(P) = 0
     then begin
          M(P) := 1;
          if A(P) = 0
          then begin
               marquage(LEFT(P));
               marquage(RIGHT(P));
               end;
          end;
```

Cet algorithme récursif est à la base du parcours préfixé des arbres binaires; il s'applique à toutes les sortes de listes, y compris les circulaires et les récursives. Malheureusement, son caractère récursif, justement, exige l'utilisation implicite d'une pile. La levée de la récursivité fait apparaître explicitement cette pile, qui ici est baptisée STACK. Initialement, celle-ci est vide.

(R.2) *marquage(P)*



Remarque : STACK est une pile destinée à contenir des pointeurs,non des cellules : la notation STACK <<— P et son inverse ont donc une autre signification que la notation STACK <== P du chapitre II.

Cette solution est sensée être utilisée pour une récupération de cellules en mémoire centrale. Or l'implémentation d'une pile requiert assez bien de place et si on pouvait trouver le moyen de se passer d'une telle pile, il est fort probable que l'on réduirait le nombre d'appels au récupérateur.

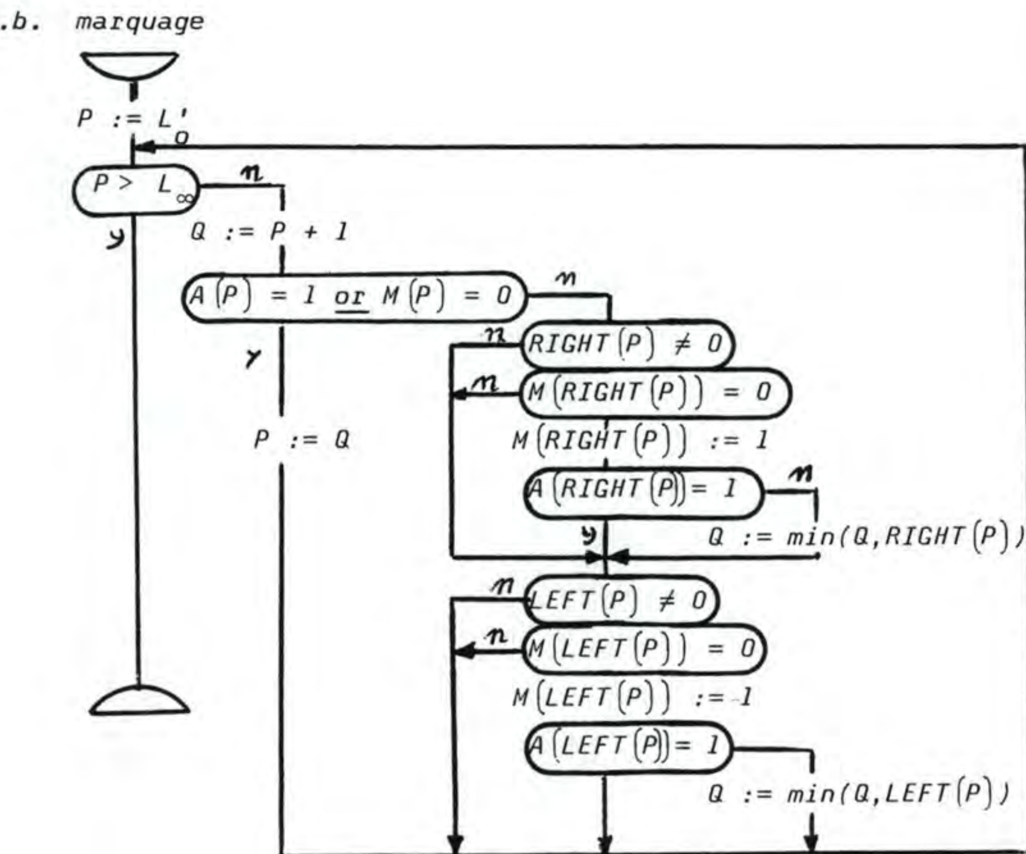
Plusieurs auteurs se sont penchés sur cette question et les algorithmes qu'ils proposent éliminent le problème de la place utilisée. Mais, revers de la médaille, ils sont aussi plus lents.

L'algorithme que voici s'appuye sur les hypothèses suivantes : les cellules sont disposées de façon contiguë dans la mémoire, de sorte que si P est l'adresse de l'une d'entre elles et que leur taille est de *c* mots-mémoire, P + *c* est l'adresse de la suivante. Mais cette disposition ne reflète pas la structure des listes ! Pour simplifier, nous prendrons *c* = 1. Le but de cet algorithme est de marquer toutes les cellules accessibles de toutes les listes. Le tas

est supposé s'étendre de l'adresse L'_o à l'adresse L_∞.

(R.3) .a. *Marquer chacune des cellules directement accessibles (têtes* **de** *listes);*

.b. *marquage*



En gros, cet algorithme réalise la tâche suivante : une fois que chacune des cellules de tête est marquée :

- rechercher le premier article non atomique marqué
- marquer ses deux successeurs s'ils existent et s'ils ne le sont pas encore
- recommencer... avec un autre article.

Il est extrêmement lent, car certaines cellules sont visitées très souvent, surtout dans certaines configurations défavorables.

Puisque les algorithmes (R.2) et (R.3) prennent l'un trop de place, l'autre trop de temps, ne serait-il pas possible de les combiner pour remédier aux défauts de l'un par

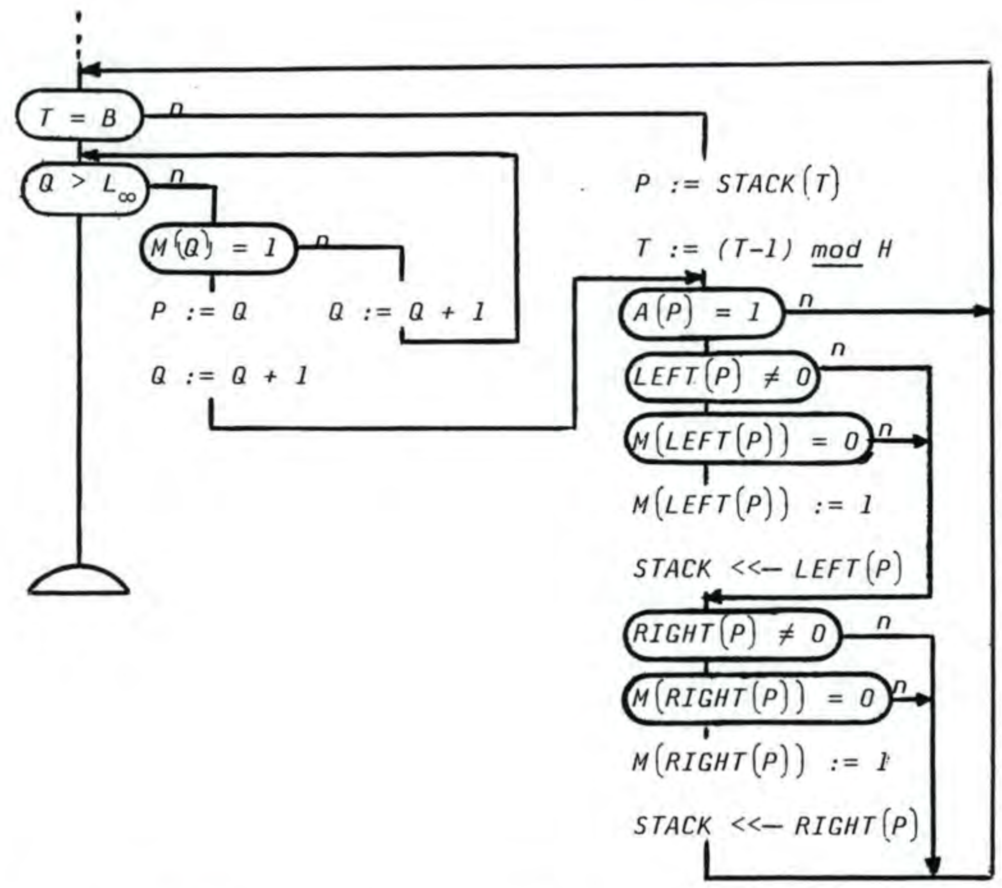
les qualités de l'autre ? La réponse à cette question est positive et il en découle l'algorithme suivant, qui fait usage d'un tableau d'adresses de taille fixe, STACK(0..H-1).

Remarque préalable : l'opération STACK <<— P cache cette fois-ci plusieurs instructions (Q est une adresse et T et B des indices relatifs au tableau STACK) :

```
T := (T+1) mod H;
STACK(T) := P;
if T = B
then begin
     B := (B+1) mod H;
     Q := min(Q,STACK(B))
     end;
```

Le tableau STACK fonctionne comme une liste libre en sortie; T est l'indice de l'élément du sommet et B + 1 celui de l'élément du fond. Nous travaillons toujours avec l'hypothèse $c = 1$.

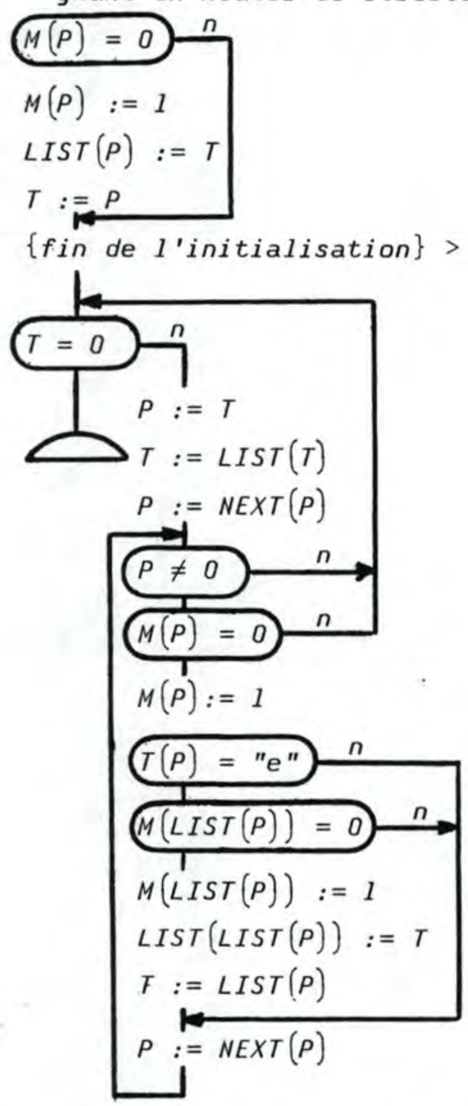
```
(R.4)  marquage
       T := -1; B := -1
       Q := L∞ + 1
       <marquer tous les articles directement accessibles et
        en mettre les adresses dans le tableau STACK, comme
        indiqué ci-dessus >
```



La caractéristique essentielle de cet algorithme est l'utilisation d'une pile dont l'implémentation est séquentielle. Or nous avons vu que les piles pouvaient être implémentéessous forme chaînée, ce qui permettait d'en avoir les éléments dans la mémoire, aux endroits disponibles. C'est cette propriété qui est mise en valeur dans l'algorithme R.5 ci-dessous. Il s'applique aux structures de liste telles que décrites dans le chapitre II (2.7.), c'est-à-dire où on distingue des éléments "header", "étiquette" et "atome" (via un champ T) et où les champs pointeurs s'appellent LIST et NEXT. On adjoint évidemment le champ M à cette structure.

La pile va maintenant être gérée via les champs LIST des cellules de type "header" (cet espace était jusqu'à présent perdu).

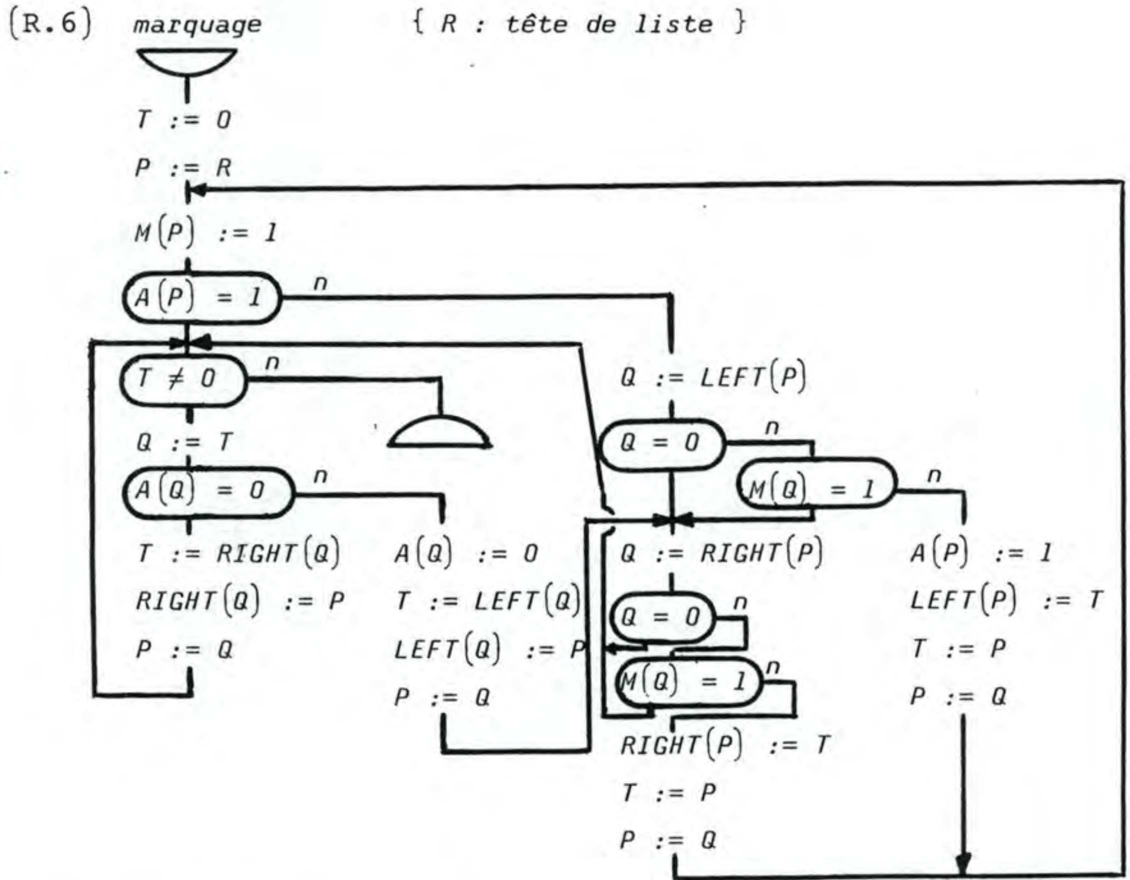
gnant un header de structure de liste, faire :



Il ne faut modifier qu'un détail pour que cet algorithme soit applicable aux structures de liste circulaires. Mais comme on l'a dit, l'application de cet algorithme est néanmoins restreinte aux structures de liste telles que proposées en 2.7., qui ne sont pas les plus générales (les cellules LISP par exemple ne s'organisent pas strictement selon ce format).

Deutsch d'une part, Schorr et Waite d'autre part,

ont développé un algorithme adapté aux structures de liste de type LISP. Nous retrouvons les quatre champs A, M, LEFT et RIGHT. Etant donné un pointeur R (non 0), la cellule ainsi référencée est marquée ainsi que toutes celles qui peuvent être atteintes à partir de celle-ci par une chaîne de pointeurs LEFT ou RIGHT. Trois autres pointeurs sont nécessaires à la réalisation de ce parcours, P, T et Q, parcours dont l'originalité vient de ce qu'il "renverse" certains pointeurs de façon à retrouver les cellules d'où on est parti. Avant que l'algorithme ne se termine, il faut bien sûr les restaurer.



Que cet algorithme donne les résultats escomptés peut être prouvé par récurrence sur le nombre de cellules à marquer.

Si seule la cellule de tête (référencée par R) doit l'être, alors, évidemment, qu'elle soit atomique ou non,

l'algorithme s'exécute correctement; de plus, en fin d'exécution, P = R.

Supposons maintenant qu'il faille marquer n (> 1) cellules : on a bien sûr A(R) = 0. Si LEFT(R) vaut 0 ou R, on passe directement à l'instruction "Q := RIGHT(P)". Sinon, on passe par l'instruction "A(P) := 1" et les suivantes (extrême droite du schéma) de sorte que LEFT(R) devient 0 et on recommence l'algorithme avec R remplacé par LEFT(R) et T remplacé par R; par induction, on peut voir que l'on marquera toutes les cellules qu'on peut atteindre à partir de celle référencée par LEFT(R)(à condition que les chemins suivis ne passent pas par la cellule initiale qui est marquée); on se retrouve alors au test T ≠ 0 avec T valant R et P valant LEFT(R). Puisque A(T) = 1 (modification faite au premier passage, dans la partie "droite" de l'algorithme selon le schéma, instruction A(P) := 1), l'étape présente (instructions A(Q) := 0 et suivantes) restaure LEFT(R) et nous arrivons à l'instruction déjà mentionnée "Q := RIGHT(P)". On développe exactement le même argument que ci-dessus (avec RIGHT(R) cette fois), et on arrive à la dernière restauration (partie gauche du schéma) qui force T à 0 et P à R.

On pourrait adapter cet algorithme à une situation où les atomes sont repérés par un bit placé dans la cellule qui contient une référence vers cet atome. Ce bit est à ajouter aux autres champs de la cellule. Tout comme pour (R.1), le temps d'exécution de (R.6) est proportionnel au nombre de cellules à marquer, mais la constante de proportionnalité est plus importante pour le second. C'est pourquoi on opte parfois pour un mélange de ces algorithmes, tel que (R.4). De nombreux auteurs se sont penchés sur ces algorithmes, cherchant à diminuer le nombre de bits nécessaires au stockage des différentes informations ou à améliorer les combinaisons de différentes techniques. On a aussi trouvé des variantes de ces algorithmes qui les rendent applicables au marquage de cellules unitaillées avec

plus de deux champs pointeurs.

4.2.2. RÉORGANISATION DE LA MÉMOIRE

La réorganisation la plus simple, en cas de cellules unitaillées, est de balayer linéairement la zone de mémoire où évoluent les structures dynamiques (le tas) et d'incorporer à la liste libre chacune des cellules non marquées rencontrées. Evidemment, ceci ne mène à aucun compactage de la mémoire.

Si on préfère compacter le tas, voici une technique possible, basée sur l'utilisation de deux variables D et F indiquant respectivement le début et le fond du tas (si ce tas se trouve compris entre les adresses L'_0 et L_∞, D et F sont initialisés à ces valeurs). On supposera comme d'habitude que les cellules ont une taille de un mot-mémoire.

Il y a deux survols du tas. Le premier a pour but de transférer les articles "actifs" du fond du tas vers les cellules libres du début, en laissant comme indication dans les anciennes cellules les adresses des nouvelles et ce jusqu'à ce que le compactage soit réalisé. Le second sert à mettre à jour les pointeurs qui référencent des articles qui ont été déplacés : les nouvelles adresses sont contenues dans les cellules qu'ils continuent à référencer.

```
(R.7)  while D < F do                    { premier passage }
           begin
           while M(D) = 1 do D := D + 1;
           while M(F) = 0 do F := F - 1;
           if D < F
           then begin
                MM(D) ← MM(F);
                M(F) := 0;
                LEFT(F) := D
                end
           end;
```

```
for D := L'₀ to F do                { deuxième passage }
    begin
    if M(LEFT(D)) = 0
    then LEFT(D) := LEFT(LEFT(D));
    if M(RIGHT(D)) = 0
    then RIGHT(D) := LEFT(RIGHT(D));
    end;
```

4.2.3. RÉCUPÉRATION PAR ÉVACUATION

Une idée qui vient rapidement à l'esprit quand on réfléchit au problème de la récupération est d'évacuer en mémoire secondaire ou dans une autre zone en mémoire centrale toutes les données utiles et ensuite de les ramener dans la zone principale (à moins qu'on intervertisse les rôles des zones principale et secondaire si on n'utilise pas de support externe). Ceci pose néanmoins divers problèmes :

- cela peut nécessiter beaucoup de place en mémoire
- les entrées-sorties ou déplacements internes ralentissent considérablement le processus
- sans précaution spéciale, on pourrait sauver plusieurs fois les mêmes cellules... et la restitution de ces différentes images ferait croître le nombre de cellules utilisées plutôt que de le diminuer !

Cette technique, abordée semblerait-il par Minsky, est évidemment applicable dans d'autres contextes que la récupération. De nombreux chercheurs ont imaginé des solutions plus ou moins performantes aux problèmes de **copie** ou de déplacement de liste. Cette première opération, contrairement à la seconde, nous interdit de modifier la liste originale (en tout cas, elle doit être restaurée si elle est dérangée pendant le copiage). Que l'on copie ou que l'on déplace, qu'il y ait ou non marquage de cellules, piles, récursivité, etc, on observe toujours

a) qu'une adresse est laissée dans l'ancienne cellule, qui servira plus loin à l'ajustement des pointeurs

b) que la liste déplacée ou la copie sont compactées.

Le paragraphe 4.5. traite de la récupération en mémoire virtuelle. On y trouvera un élégant algorithme de déplacement de listes LISP.

4.3. RÉCUPÉRATION DE CELLULES DE TAILLE VARIABLE

4.3.1. MARQUAGE

On conviendra que les éléments intervenant dans la structure de liste à marquer contiennent tous les champs suivants, dans le premier mot-mémoire : M, bit de marquage; N, nombre de champs pointeurs se trouvant dans la cellule; T, taille de l'article. Les N pointeurs de la cellule sont dans les mots-mémoire suivant le premier; considérons pour simplifier qu'ils sont les éléments d'un tableau NEXT[1..N] (à bornes dynamiques).

L'algorithme de départ est récursif (et donc inutilisable en pratique) :

```
(R.8)   procedure marquage(P);      { P : pointeur vers une cellule
                                                   de tête }
        if P ≠ 0
        then if M[P] = 0
             then begin
                    M[P] := 1
                    for i := 1 to N[P] do
                         marquage(NEXT[i])
                  end;
```

Pour lever la difficulté inhérente à l'utilisation d'une pile implicite, on a proposé plusieurs solutions calquées sur celles qui permettent la récupération de cellules unitaillées : ajouter deux champs par cellule pour y gérer la pile, implémenter une pile de taille fixe, renverser les pointeurs (Deutsch, Schorr et Waite)...

Nous ne nous étendrons pas sur ce point, tout en insistant bien sûr sur les difficultés rencontrées dans ce problème, d'une nature beaucoup plus complexe que le précédent car les cas de figure sont beaucoup plus nombreux (et les hypothèses faites ici rarement remplies).

4.3.2. RÉORGANISATION DE LA MÉMOIRE

Les cellules marquées et non marquées étant de taille variable, il n'est plus possible de les déplacer à volonté comme cela se faisait en [R.7]. Un compactage de type glissement a été un des premiers réalisés. Il se déroule en deux phases : un premier examen de la mémoire sert au compactage proprement dit ainsi qu'à la création d'une table de rupture utilisée lors d'un deuxième passage qui, lui, réajuste les pointeurs. La table de rupture contient pour chaque "trou" (**suite** de cellules non marquées) une entrée à deux champs A et B : A est son adresse et B la somme des tailles des trous dont les adresses sont inférieures ou égales à A. Cette table ne requiert pas d'espace supplémentaire, car elle est gérée dans les trous eux-mêmes. De temps en temps, par contre, il s'avère indispensable de transporter cette table dans un trou plus grand, laissé par le compactage. A la fin du premier examen, la table de rupture occupe la partie libérée de la mémoire; on la trie pour accélérer le déroulement du réajustement des pointeurs. Cette mise-à-jour se fait très simplement : étant donné un pointeur X, on cherche dans la table l'entrée correspondant à l'adresse A la plus proche de X ($A < X$) et on retire B de X.

Le compactage se fait vers le début de la zone-mémoire (petites adresses). La difficulté principale de cette méthode est la gestion de la table car, même si on est sûr d'avoir suffisament de place pour l'implémenter, la place maximale n'est pas disponible tout de suite. De nombreux **transferts** peuvent devoir être réalisés dans des cas de

figure variés. Ceci prend tellement de temps qu'une autre idée est d'effectuer un premier passage pour réaliser cette table sous forme de liste chaînée (enchaînement des trous), un deuxième pour ajuster les pointeurs, un troisième pour réaliser le glissement.

D'autres méthodes proposées ont pour différences essentielles avec la précédente, la façon d'implémenter et de calculer la table de rupture.

Certains auteurs se sont passés de cette table, en ajoutant un bit par pointeur, ou en imposant des restrictions sur les pointeurs, ou encore en tentant de retarder au maximum le compactage.

La seule conclusion qui s'impose, c'est que la méthode utilisée dépend fortement de la façon dont les cellules sont organisées, c'est-à-dire finalement de l'implémentation du langage. D'autre part, soulignons encore le fait bien connu que d'un cas particulier à un autre, une même technique peut donner des résultats tout à fait discordants.

4.4. COMPTEURS DE RÉFÉRENCES

Proposés voici plus de 20 ans, les compteurs de références reviennent à la une. L'avantage qu'ils ont, c'est évidemment de rendre inutile le marquage des cellules actives. De plus, ils se prêtent assez bien aux structures de taille variable. Cette technique requiert un champ supplémentaire par cellule, appelons-le COUNT. Il contient le nombre de fois que la cellule est référencée. Ce champ est à mettre à jour chaque fois qu'un pointeur référençant la cellule est créé ou détruit. Ce champ COUNT, qui doit être suffisament grand, et la mise-à-jour continuelle, sont des facteurs consommateurs d'espace et de temps; de plus, ils sont inadaptés aux listes circulaires. En effet, si une telle liste devient inaccessible, chacun des éléments de la liste n'en est pas moins référencé par le précédent...

Pour ce qui est de l'aspect "perte de temps", on peut bien sûr considérer qu'il est regagné par rapport au temps perdu dans la récupération par marquage.

Chaque fois qu'un compteur retombe à zéro, la cellule devenue inutile est placée sur une pile (plus exactement, son adresse). Lorsqu'on a besoin d'une cellule, on la prend sur la pile et alors (et seulement alors) les compteurs de ses descendants sont décrémentés. La pile est gérable comme une liste libre, via le champ COUNT.

Une combinaison des compteurs de référence et du garbage collector a été imaginée, sous diverses formes, par plusieurs auteurs. La plupart du temps, on ne manipule que les compteurs de références. La récupération par marquage, plus onéreuse, intervient en dernier recours. On considère que lorsqu'un compteur de références atteint sa valeur maximale, il ne peut plus être modifié. Quand il n'y a plus de cellules disponibles, la récupération classique démarre par une remise à zéro de tous les compteurs. Ils sont restaurés durant le marquage (quand on visite une cellule, on ajoute 1 à son champ COUNT) qui permet de restituer à l'allocateur les structures de listes circulaires inactives et des cellules dont le compteur de références avait la valeur maximale (et n'avait donc pas été décrémenté).

L'approche hybride de Deutsch et Bobrow est foudroyante dans le cas du LISP, basée qu'elle est sur un fait que la statistique a mis en évidence : dans la plupart des programmes LISP, la plupart des compteurs de références valent 1 (97 % !).

Les auteurs proposent trois tables :

a) *MRT* (multiple reference table).
   On y accède par l'adresse d'une cellule et la valeur associée est celle du compteur de référence de la cellule. Seules les cellules dont le compteur indique deux ou plus sont représentées dans cette table.

b) *ZCT* (zero count table).

Se trouvent dans cette table les adresses des cellules dont le compteur de références est à 0. Ces cellules peuvent être de deux types : celles qui sont référencées par les variables du programme (toujours actives) et celles qui ne le sont pas (et donc peuvent être rendues à l'allocateur).

c) *VRT* (variable reference table).
Elle contient les adresses des cellules référencées par les variables du programme (y compris celles des variables se trouvant dans les piles créées lors des appels récursifs).

Si une cellule n'est reprise ni en MRT ni en ZCT, c'est que son compteur de référence est à 1.

On distingue trois types d'opérations ou transactions qui affectent l'accessibilité des données :

- allocation d'une nouvelle cellule
- création d'un pointeur
- destruction d'un pointeur

Mais plutôt que de mettre à jour les tables décrites ci-dessus dès que survient une transaction, on préfère les enregistrer dans un fichier séquentiel qui est examiné de temps en temps, et c'est seulement à ce moment là que les tables sont réactualisées. Ceci minimise le nombre de défauts de pages.

Lors de l'allocation d'une nouvelle cellule, l'adresse de celle-ci devrait être placée en ZCT. Mais puisque cette opération est normalement suivie de la création d'un pointeur vers la nouvelle cellule, ce qui implique son retrait de la ZCT, ces deux transactions sont ignorées.

Trois cas peuvent se présenter lors de la création d'un pointeur :

a) le pointeur désigne une cellule de la MRT : le compteur de références correspondant est incrémenté (s'il ne valait pas le maximum)
b) le pointeur désignait une cellule de la ZCT : elle en est retirée (COUNT = 1)

c) si la cellule désignée n'est ni dans la MRT ni dans la ZCT, c'est que son compteur de références était à 1 et elle est ajoutée à la MRT (COUNT = 2).

Lors de la destruction d'un pointeur, il y a deux possibilités :

a) le pointeur référençait une cellule de la MRT; on en décrémente alors le compteur COUNT (s'il n'est pas maximal) et si il arrive à un, la cellule est retirée de la MRT.
b) la cellule référencée n'est pas en MRT, donc son compteur passe de 1 à 0 et la cellule est placée en ZCT.

On se sert de la VRT quand de nouvelles cellules sont ajoutées à la liste libre. Une cellule est restituée lorsqu'elle figure dans la ZCT et pas dans la VRT. Puisque la pile est constament mise à jour, la VRT n'est calculée qu'épisodiquement. On met à jour la ZCT en éliminant les entrées correspondant à des cellules restituées qui ne sont pas référencées par des variables du programme.

Cette technique est destinée à un environnement paginé. Pour la récupération classique, un algorithme dérivé de [R.9] (voir paragraphe 4.5.) est préconisé par les auteurs. Une dernière de leurs constatations est qu'un processeur auxiliaire, dont la tâche serait d'examiner les tables ZCT et VRT afin de déterminer les cellules à incorporer à la liste libre, permettrait d'accélérer la récupération.

4.5. RÉCUPÉRATION ET MÉMOIRE VIRTUELLE

Si on dispose d'une mémoire virtuelle (bien plus importante que la mémoire principale et soumise à la pagination), il y a d'importants changements quant à l'implémentation d'un récupérateur. On a par exemple la possibilité d'utiliser une pile car la place n'est plus comptée. Par contre, il est essentiel de réduire le nombre de défauts de page. Pour ce faire, un compactage linéaire est très important puisqu'il évite la dispersion d'une structure de liste sur

plusieurs pages.

Prenons le cas de la fonction LISP "<u>cons</u>" qui contient implicitement un appel à l'allocateur : une politique intelligente pour réaliser <u>cons</u>(x,y) est de choisir une cellule selon ces critères (du plus intéressant au moins intéressant) :

- dans la page contenant la cellule référencée par y;
- dans la page contenant la cellule référencée par x;
- dans la page contenant la cellule la plus récemment allouée;
- dans une page contenant un bon nombre de cellules libres.

Naturellement, ceci suppose l'existence d'une liste de cellules libres par page.

Un autre changement concerne le moment d'appeler le récupérateur : si on dispose de beaucoup de place, ce moment devient celui où le nombre de défauts de page ralentit considérablement le déroulement d'un programme.

Regardons d'un peu plus près un algorithme donné pour le LISP mais applicable aux structures multitaillées. On divise en deux demi-espaces la mémoire disponible; à un moment donné, un seul est utilisé par l'allocateur. La récupération déplace les listes utiles d'un demi-espace à l'autre sous forme compactée. Nous emploierons les notations suivantes : P_o est l'adresse de la tête de la structure à déplacer; B et S sont deux adresses et B est initialisé à la première adresse du second demi-espace, soit L_o. Nous supposerons en outre que les adresses dans le premier demi-espace sont toutes inférieures à L_o.

Le coeur de l'algorithme, ce sont deux fonctions MOVE et COPY, dont les arguments et valeurs sont de type pointeur et dont voici le texte :

```
function copy(P) : pointer;

LEFT(B) := LEFT(P);
RIGHT(B) := RIGHT(P);
copy := B;
B := B + 1 ;
```

```
function move(P) : pointer;

if (P = 0 or P ≥ L0)
then move := P
else begin
     if LEFT(P) < L0
     then LEFT(P) := copy(P);
     move := LEFT(P);
     end;
```

COPY a donc pour but de simplement copier la cellule référencée par P dans une nouvelle cellule; MOVE, quant à elle, est principalement utilisée pour mettre à jour les pointeurs, ce qui peut éventuellement amener un appel à COPY. Lorsqu'un article est déplacé, le champ LEFT de son ancienne cellule contient l'adresse de sa nouvelle localisation.

L'algorithme en lui-même est tout simplement

```
(R.9)   S := move(P0);
        if S ≠ 0
        then while S < B do
                  begin
                  LEFT(S) := move(LEFT(S));
                  RIGHT(S) := move(RIGHT(S));
                  S := S + 1
                  end;
```

ANNEXE I

APPERÇU DE L'IMPLÉMENTATION DU LISP

5.1. DESCRIPTION GÉNÉRALE

5.1.1. LE LANGAGE EXTERNE ET LES S-EXPRESSIONS

Nous décrivons brièvement dans ce paragraphe le langage dans lequel les informations doivent être transmises à l'ordinateur, c'est-à-dire le S-langage ou langage externe, et les mots de celui-ci (les informations) qui sont appelés S-expressions. Notre but n'est pas d'étudier la sémantique du langage LISP, ni même sa syntaxe en totalité. C'est l'implémentation du langage qui est l'objet de cette annexe. Ceci requiert néanmoins un minimum d'introduction, mais on fera peu cas de la signification des S-expressions mentionnées à titre exemplatif.

Exemples de S-expressions

forme courante	S-expression LISP
x + y + z	(PLUS X Y Z)
x * y	(TIMES X Y)
x ** n	(EXPT X N)
-x	(MINUS X)
x_1	(X1)

Il apparaît donc que l'on utilise en LISP une notation préfixée; seules les majuscules sont admises; l'écriture des nombres, elle, est normale. Une séquence de majuscules et de chiffres telle que TIMES ou X1 ou ... est une entité indécomposable (sous peine de perdre toute signification) appelée <u>symbole atomique</u>. Les symboles atomiques, les nombres, les expressions LISP ci-dessus sont autant d'exemples de S-expressions à partir desquelles on peut construire des S-expressions de plus en plus complexes par composition :

sin(x + cos(y))   devient   (SIN (PLUS X (COS Y)))

On appelle <u>liste</u> toute S-expression qui débute par une parenthèse gauche, suivie d'une séquence (éventuellement vide) de S-expressions séparées par des intervalles blancs, et qui se termine par une parenthèse droite. Les nombres et les symboles atomiques ne sont donc pas des listes.

Une <u>paire pointée</u> est composée de deux S-expressions séparées par un point et encadrées par des parenthèses. Une paire pointée est une S-expression.

Exemples de listes et paires pointées :

(A B C) () (A.B) (((A B).X2).X3) (X Y (Z.W))

Nous verrons qu'en fait, du point de vue des représentations internes, toute liste est équivalente à une paire pointée.

5.1.2. SYNTAXE DU LANGAGE EXTERNE

*<SYMBOLE DE BASE> ::= <LETTRE> | <**chiffre**> | <**signe**> | <DELIMITEUR>*

*<LETTRE> ::= A | B | C | ... | Y | Z | **

<chiffre> ::= 0 | 1 | 2 | ... | 8 | 9

<signe> ::= + | -

<DELIMITEUR> ::= (|) | . | ˜

Notes : ~ est le caractère "espacement" et les trois points ... ne sont pas un symbole du langage externe mais indiquent une énumération fastidieuse.

<S-expression> ::= <symbole atomique> | <nombre> | <paire pointée>
<liste>

<symbole atomique> ::= <LETTRE> | <LETTRE><finale atomique>

<finale atomique> ::= <LETTRE> | <LETTRE><finale atomique> | <chiffre> |
<chiffre><finale atomique>

<nombre> ::= <nombre sans signe> | <signe><nombre sans signe>

<nombre sans signe> ::= <entier sans signe> |
<entier sans signe>.<entier sans signe>

<entier sans signe> ::= <chiffre> | <chiffre><entier sans signe>

<paire pointée> ::= (<S-expression>.<S-expression>)

<liste> ::= () | (<contenu de liste>)

<contenu de liste> ::= <S-expression> |
<élément de liste><espace vide><contenu de liste>

<élément de liste> ::= <S-expression>

<espace vide> ::= ~ | ~<espace vide>

5.1.3. LA TABLE DES PROPRIÉTÉS D'UN SYMBOLE ATOMIQUE

Chaque symbole atomique a une représentation interne particulière qui lui confère diverses propriétés; cette représentation est donc une sorte de <u>table de propriétés</u>. Chaque type de propriété dans cette table est repérée par un <u>indicateur</u> qui est un symbole atomique déterminé.

Exemples

ATOMES	INDICATEURS	PROPRIETES CORRESPONDANTES
FACT	EXPR	(LAMBDA (N) (COND ((ZEROP N) 1) (*T* (TIMES N (FACT (SUB1 N))))))
	PNAME	indication qui donne accès à la séquence des caractères du symbole atomique (F, A, C, T)

XP2Z	APVAL	(A B (C.D) B A)
	PNAME	
CONC	FEXPR	(LAMBDA (I J) (MAPCON I (FUNCTION (LAMBDA (I) (EVAL (CAR I))))))
	PNAME	
CAR	SUBR	indication qui donne accès à un sous-programme en langage machine
	PNAME	
LIST	FSUBR	indication qui donne accès à un sous-programme en langage machine

Chaque table ne peut contenir qu'un seul des cinq indicateurs EXPR, FEXPR, SUBR, FSUBR, APVAL; l'indicateur PNAME doit figurer dans chaque table de propriétés. Pour EXPR, FEXPR et APVAL, l'indication est toujours une S-expression qui sera traitée spécifiquement dans chacun des trois cas; sa sémantique n'est pas l'objet de cette étude.

5.2. REPRÉSENTATION INTERNE DES S-EXPRESSIONS ET STRUCTURES DE LISTE

5.2.1. NOTATIONS ET TERMINOLOGIE

Toute S-expression est représentée dans l'ordinateur, à chaque instant, par une adresse de mémoire.

Des adresses quelconques seront désignées par des lettres grecques; les contenus des mots-mémoire d'adresse α, β, γ seront respectivement notés (α), (β), (γ); ces contenus étant généralement eux aussi des adresses, le contenu du mot-mémoire d'adresse (α) sera noté $((\alpha))$ ou $C(\alpha)$.

Représentation graphique d'un mot-mémoire : rectangle précédé ou surmonté de son adresse. Une flèche émanant d'un mot-mémoire ainsi représenté désigne le mot-mémoire dont l'adresse est contenue dans le mot-mémoire d'origine. Ce mot-mémoire ne sera pas nécessairement représenté.

Exemples :

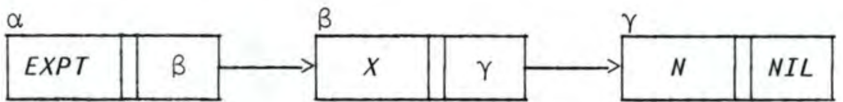
α [β]   δ[γ]——>γ[X]

$(\alpha) = \beta$ $((\delta)) = C(\delta) = (\gamma) = X$

5.2.2. Représentation interne des S-expressions

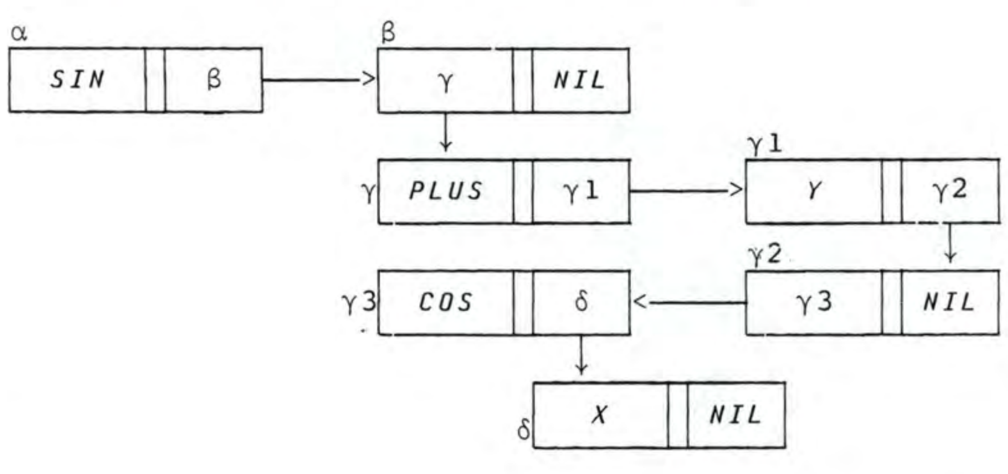
Les rangements en mots-mémoire consécutifs ne sont pas possibles lorsqu'il s'agit de représenter des listes ou S-expressions quelconques, puisqu'on ignore le nombre d'éléments d'une liste et vu le fait que ces éléments sont (généralement) des listes.

Dans les exemples qui suivent, NIL est un symbole atomique spécial qui sert à repérer le dernier mot-mémoire utile.



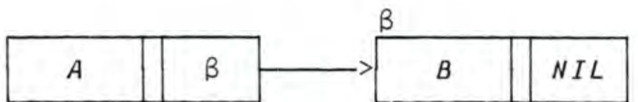
pour (EXPT X N).

(SIN (PLUS Y (COS X))) deviendra en mémoire :



(A.B) deviendra en mémoire : [A | B]

Regardons encore (A.(B)) :



La représentation interne d'une liste est appelée <u>structure de liste</u> ou simplement <u>liste</u>.

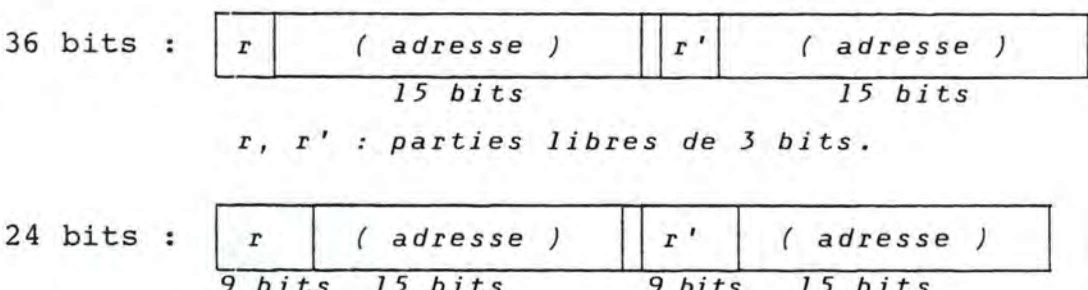
Remarque : (A B) et (A.(B)) ont des représentations internes identiques; il en va de même pour (B) et (B.NIL). On dira que deux S-expressions sont <u>équivalentes</u> si elles ont des représentations internes identiques. Toute liste est équivalente à un assemblage exclusivement composé de paires pointées; mais une paire pointée ne peut pas toujours être écrite sous forme de liste.

5.2.3. CELLULE DE LISTE

Suivant le nombre de mots-mémoire d'un ordinateur et la taille de ceux-ci, on appellera <u>cellule de liste</u>

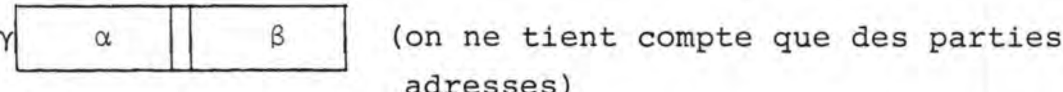
- soit un mot-mémoire pouvant contenir deux adresses et deux parties libres de quelques bits
- soit deux mots-mémoire d'adresses consécutives.

Ainsi, pour un ordinateur de 32 K de mémoire centrale dont les mots-mémoire sont de



Le <u>premier</u> (resp. le <u>second</u>) <u>élément</u> d'une cellule est la partie qui se trouve à gauche (resp. à droite) du double trait vertical. Si on ne tient pas compte du

contenu des parties libres, on adoptera une représentation simplifiée :



(on ne tient compte que des parties adresses)

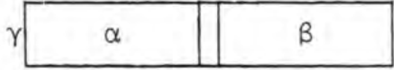
où γ est l'adresse du premier élément de la cellule (c'est-à-dire l'adresse de la cellule elle-même ou celle du premier mot-mémoire selon le cas).

5.2.4. REPRÉSENTATION INTERNE DES PAIRES POINTÉES ET LISTES

La représentation interne de chaque symbole atomique (ou nombre) est une adresse qui sera notée (sur papier) comme ce même symbole (ou nombre). En fait, à chaque symbole atomique est associée une table de propriétés qu'on appellera P-liste lorsque l'on parle de représentation interne.

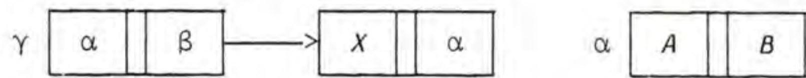
En dehors de ces cas, toute S-expression est une liste ou une paire pointée et sa représentation interne est un ensemble de cellules de liste arrangées selon les règles suivantes :

- toute S-expression a un nom interne qui est une adresse (qui désigne la S-expression elle-même)
- si α et β sont les noms internes des deux éléments d'une paire pointée (dans l'ordre), alors la représentation interne de cette paire est une cellule de liste de la forme

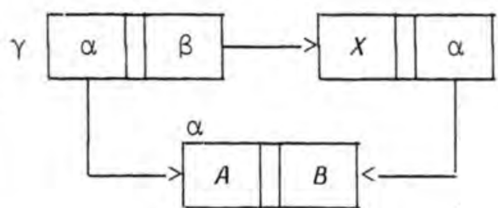


  et γ est le nom interne de la paire pointée elle-même.
- la représentation interne d'une liste non vide s'obtient comme cas particulier d'un paire pointée : la liste (X) est équivalente à la paire pointée (X.NIL), la liste (A B C ... R) à la paire pointée (A.(B C ... R)).
- la représentation interne de la liste vide () est celle du symbole atomique NIL.

Remarque : si on a



alors on simplifiera la représentation de ((A.B).(X.(A.B))) en



5.2.5. Représentation interne des symboles atomiques

Nous avons dit que la représentation interne d'un symbole atomique était l'adresse désignée par ce symbole. En fait, il s'agit de l'adresse de la première cellule de la P-liste du symbole atomique considéré. Une P-liste porte mal son nom, puisqu'il ne s'agit ni d'une liste, ni même d'une S-expression. Néanmoins, la structure interne d'une liste de propriétés est analogue aux représentations internes que nous avons déjà évoquées.

5.2.5.1. CAS DES CELLULES DE LISTE COMPOSEES DE DEUX MOTS-MEMOIRE. Si on se reporte aux schémas de la page 77, on constate que chacune des parties libres r et r' comporte 9 bits. On subdivisera ces deux champs en trois zones de 3 bits (pouvant donc contenir les représentations binaires des chiffres 0 à 7) numérotés de 1 à 3 de gauche à droite.

Ceci dit, toute P-liste est ainsi structurée :

1) Critère de reconnaissance d'une P-liste : la première cellule d'une P-liste est toujours caractérisée par sa première zone, où seul le premier bit est positionné (chiffre 4 dans la première zone).
2) La première cellule contient un des indicateurs SUBR, FSUBR, EXPR, FEXPR, APVAL ou PNAME (celui-ci par défaut). Ces indicateurs, symboles atomiques, sont donc

des adresses et par conséquent le premier élément de cette cellule initiale est une adresse.

3) Toute cellule d'indicateur est directement connectée à une cellule qui donne accès à l'indication correspondante. Cette cellule (que l'on pourrait qualifier de <u>cellule d'accès</u>) contient l'adresse des <u>cellules d'indication</u>. La table des propriétés d'un symbole atomique, donc sa P-liste, peut contenir deux indicateurs (voir paragraphe 5.1.3.). Dans ce cas, le second élément de la cellule d'accès contient l'adresse d'une deuxième cellule d'indication.

4) Si l'indicateur est SUBR ou FSUBR, l'indication utilise une seule cellule de liste : le premier élément contient l'adresse d'un sous-programme en langage machine et, si nécessaire, le second élément contient le nombre d'arguments. La première zone d'une telle cellule contient le chiffre 1.

5) Si l'indicateur est EXPR ou FEXPR, l'indicateur est une S-expression dont le **nom interne** est l'adresse α contenue dans la cellule d'accès; la cellule d'indication est remplacée par une structure de liste qui est la représentation interne de la S-expression.

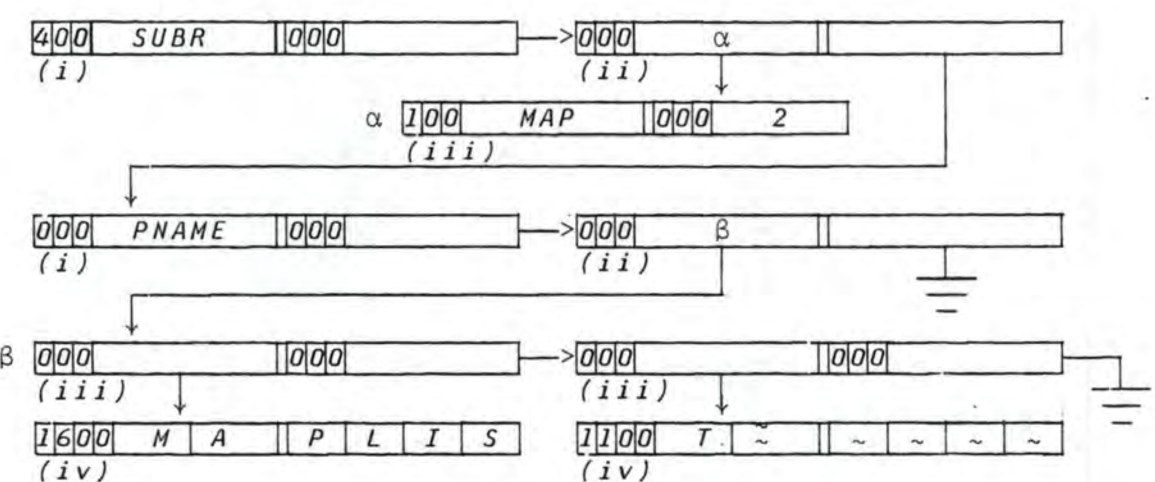
6) Si l'indicateur est APVAL, l'indication est également une S-expression dont l'adresse est contenue dans le premier élément de la cellule d'indication. Le second élément contient NIL. On utilise dans ce cas la cellule d'accès.

7) Toute P-liste contient l'indicateur PNAME (contraction de l'anglais Print Name), l'indication étant l'ensemble des caractères du nom externe du symbole atomique considéré. Le code employé (DCB) exige 6 bits par caractère. Une cellule de liste contient 2 fois 24 bits; les 3 zones du premier élément (9 bits) sont réservées mais pas celles du second. On pourrait donc placer les représentations binaires de 6 caractères. Mais si le nom externe comporte plus de 6 lettres, il faut adopter

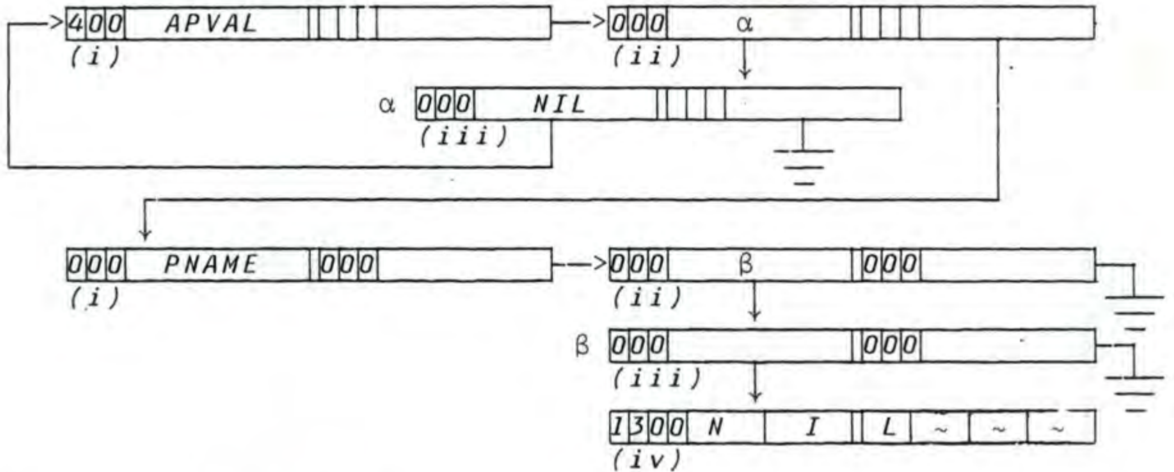
une disposition en niveaux. Pour généraliser, cette disposition en niveaux est toujours choisie. On arrive ainsi à la structure suivante : la cellule d'indicateur PNAME, la cellule d'accès (premier élément : une adresse; deuxième : NIL), la (ou les) cellule(s) d'indication, dont le premier élément contient l'adresse des 6 premiers caractères du nom externe (complété éventuellement par des blancs), et enfin les cellules contenant les caractères ou cellules pleines. La première zone d'une cellule pleine contient le chiffre 1, la deuxième, le nombre de caractères utiles.

Avant d'aborder les différences qui se présentent pour les cellules de liste formées d'un seul mot-mémoire, illustrons les règles 1 à 7 par deux exemples.

Représentation interne de MAPLIST :



Représentation interne du symbole atomique NIL :



(i) : cellule d'indicateur (SUBR et PNAME sont des adresses)

(ii) : cellule d'accès

(iii) : cellule d'indication (MAP est l'adresse d'un sous-programme en langage machine à deux arguments)

(iv) : cellule pleine

5.2.5.2. CAS DES CELLULES DE LISTE COMPOSEES D'UN SEUL MOT-MEMOIRE. Quelques petites modifications sont à signaler par rapport au premier cas :

- la première cellule d'une P-liste contient la valeur -1 dans la partie adresse.
- la seconde cellule contient un indicateur (SUBR, FSUBR, EXPR, FEXPR, APVAL ou PNAME).
- toute cellule d'indication liée à SUBR voit ses bits 1, 2 et 3 positionnés.
- Une cellule pleine (36 bits) peut contenir 6 caractères et par conséquent il n'y a pas d'indication du nombre de caractères utiles; une cellule garnie de moins de 6 caractères utiles est complétée par un caractère non imprimable par l'imprimante.

5.2.6. REPRÉSENTATION INTERNE DES NOMBRES

C'est de nouveau une adresse de mémoire qui représente tout nombre; cette adresse est celle du premier élément d'une

cellule de liste qui contient toues les informations relatives au nombre considéré. Nous ne nous intéressons pas aux conventions choisies pour caractériser les cellules de cette structure de liste.

5.3. La liste libre

5.3.1. Introduction

La mémoire contient

- les sous-programmes en langage machine nécessaires à la lecture des données, à leur interprétation et à l'impression des résultats d'un programme LISP;
- les représentations internes d'informations présentes en permanence dans le système.

Contentons-nous de supposer que la mémoire contient tout ce qui est nécessaire à l'exécution des programmes LISP (ces programmes sont une séquence de doublets). Avant même que les doublets ne soient introduits en mémoire, il existe un ensemble de mots-mémoire non utilisés et contigus. Soit FREE l'adresse du premier mot-mémoire de cette liste. Notons que ces mots-mémoire sont organisés en cellules de liste. FREE est une adresse fixée une fois pour toutes. On organise les cellules de liste non utilisées en structure de liste classique (chaînées par le second élément). La liste d'adresse FREE est appelée liste libre ou FREE LIST. Le contenu du premier élément de chaque cellule de la liste est sans signification.

On forme la liste libre en lançant un sous-programme spécial appelé FRLIS. Cette liste est un réservoir de cellules. Fréquemment, en effet, un programme LISP devra extraire une nouvelle cellule, et au moins

- à la création des représentations internes des S-expressions composant les doublets (base d'un programme LISP), notamment les P-listes des nouveaux symboles atomiques;

- lors de l'interprétation des doublets, lorsqu'il s'agit de ranger les représentations internes des résultats et des S-expressions intermédiaires ne pouvant momentanément pas être perdues.

5.3.2. EXTRACTION D'UNE CELLULE DE LISTE

La cellule extraite de la liste libre est la seconde, c'est-à-dire celle dont l'adresse est le contenu de la cellule FREE. Si cette adresse est NIL, il y a lieu de nettoyer la mémoire à la recherche de cellules de liste "inutiles"; le garbage collector (sous-programme RECLAIM) réalise cette opération. Si cette adresse n'est pas NIL, elle est mémorisée à un endroit spécial (cellule d'adresse MM0) tandis que la liste libre est restaurée.

Il est évident qu'après une seule extraction, les cellules de la liste libre ne sont plus contiguës.

5.3.3. RETOUR D'UNE CELLULE DANS LA LISTE LIBRE

Supposons que la cellule d'adresse α ne soit plus utilisée par le programme : on a tout intérêt à ce qu'elle réintègre la liste libre. Le sous-programme s'occupant de ces opérations s'appelle RTFREE et les réalise simplement en consultant MM0 (sensée contenir l'adresse de la cellule à rendre) et en insérant la cellule correspondante en deuxième position (derrière FREE). De nouveau, on perd rapidement le caractère contigu des cellules de la liste libre.

5.4. ORGANISATION GÉNÉRALE

5.4.1. PRÉSERVATION PAR PILES-LISTES

Pour la plupart des fonctions de type SUBR (le caractère "fonction" d'un symbole atomique découle de la sémanti-

que du langage LISP que nous continuons d'ignorer), le programme correspondant place l'adresse du résultat dans un mot-mémoire d'adresse A0. Etant donné que LISP est récursif et que de toutes façons un sous-programme peut en utiliser un autre, il va falloir gérer une pile de résultats intermédiaires. Etudions une façon de réaliser cette préservation qui s'appuye sur des piles à structure de liste ou piles-listes. Dans cette organisation, A0 devient l'adresse de tête d'une structure de liste de la forme suivante :

$$\varepsilon$$
$$A0 \; [\,\alpha n \mid \varepsilon\,] \to [\,\alpha n\text{-}1 \mid \phi\,] \to \ldots [\,\alpha 1 \mid NIL\,]$$

où αn = (A0) est le résultat du dernier sous-programme appelé et où $\alpha n\text{-}1$, $\alpha n\text{-}2$, ..., $\alpha 1$ sont les adresses des résultats successivement obtenus (en A0) par des sous-programmes antérieurs.

Préserver le contenu de A0 revient donc à remplacer la pile-liste par une autre qui comporte un élément de plus :

$$\psi$$
$$A0 \; [\,\alpha n \mid \psi\,] \to [\,\alpha n \mid \varepsilon\,] \to \ldots [\,\alpha 1 \mid NIL\,]$$

et de la sorte, l'adresse $\alpha n{+}1$ d'un nouveau résultat d'un sous-programme ultérieur pourra être sauvée à la place de αn dans A0 (cette dernière donnée ayant été recopiée dans la seconde (et nouvelle) cellule de la pile-liste).

On appellera par exemple PUSDA0 le programme chargé de réaliser le passage de la liste "avant" à la liste "après". Il comporte les étapes suivantes :

- appel à EXFREE qui donne (MM0) = β, adresse d'une cellule extraite de la liste libre;
- rangement de αn et ε dans les deux éléments de la nouvelle cellule;
- remplacement de ε par β dans le deuxième élément de la première cellule de la pile-liste (A0).

C'est donc de nouveau en seconde position que l'on ajoute un élément à la pile-liste.

Note : si A0 est | *NIL* | *NIL* | , alors PUSDA0 n'a aucun effet.

Lorsqu'on a récupéré une adresse contenue dans A0, il faut retrouver la pile-liste telle qu'elle était avant l'exécution du dernier PUSDA0. On appelle cette double opération "restitution" du contenu de A0 et elle est prise en charge par un programme qu'on pourrait appeler POPA0 et qui effectue les actions que voici, après saisie de l'adresse contenue en A0 :

- placer l'adresse du second mot de la pile-liste en MM0 pour son retour à la liste libre;
- ranger le contenu des deux éléments de cette cellule en A0;
- appeler RETFREE.

Note : si A0 est réduit à | α | *NIL* | , il faut arriver à | *NIL* | *NIL* |.

5.4.2. PILES-LISTES D'ARGUMENTS

Une fonction LISP (de type SUBR) a un certain nombre d'arguments, dont les adresses sont par exemple α, β, γ, ... Ces adresses sont en fait les données du sous-programme relatif à la fonction; elles doivent être rangées dans des mots-mémoire prédéterminés, par exemple A1, A2, A3, ... Ces adresses sont autant de têtes de piles-listes, pour les mêmes raisons que ci-dessus : appels imbriqués de sous-programmes ou programmes récursifs...

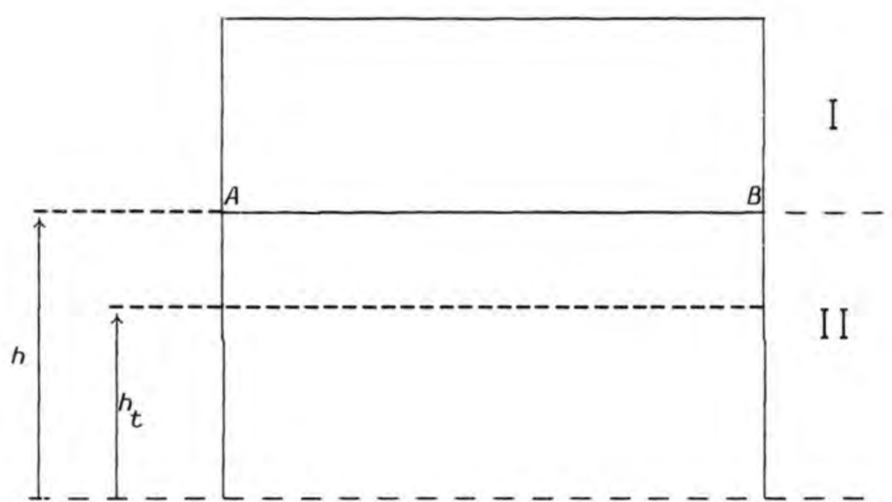
Tout à fait en parallèle avec les piles-listes déjà citées, notons encore la pile-liste RETAD, qui est celle qui permet de préserver les adresses de retour.

5.4.3. AUTRES MÉTHODES D'IMPLÉMENTATION

Nous allons dans ce paragraphe évoquer deux autres méthodes de préservation utilisant des mots-mémoire d'adresses consécutives. Elles sont comparées entre elles et avec la première aux points de vue

- de l'automatisme d'emploi
- de la rapidité d'exécution
- de l'utilisation judicieuse de la mémoire disponible.

En effet, étant donné un ordinateur, c'est ce triple critère qui devrait permettre de choisir sûrement une méthode déterminée. Quelle que soit la méthode adoptée, un "ramasse-miettes" ou récupérateur devra être disponible pour permettre de reconstituer une liste libre chaque fois qu'elle sera épuisée. Ce récupérateur sera plus ou moins rapide et plus ou moins volumineux selon la technique utilisée, et il sera appelé plus ou moins souvent pendant l'exécution d'un programme selon l'importance de la mémoire disponible. L'importance, la rapidité et la fréquence d'appel d'un récupérateur sont des points essentiels qui conditionnent le choix d'une méthode d'implémentation du langage.



5.4.3.1. PREMIERE METHODE. Deux zones totalement indépendantes se partagent la mémoire disponible (après rangement de l'interprêteur, du récupérateur et de tout le dispositif LISP). La zone II est organisée en structure de liste et accueille la liste libre dans son état initial; la zone I est réservée à l'évolution de la pile; sa hauteur à l'instant t est notée h_t. On devra toujours avoir, en tout instant de l'exécution d'un programme, $h_t \leq h$. La liste

libre occupe la partie de la zone II qui n'a pas fait l'objet de demande d'extraction. Lorsque cette partie est épuisée, il faut faire appel au récupérateur. Sa mission est de balayer la zone II à la recherche des cellules devenues inutiles pour les réorganiser en structure de liste. Il sera relativement simple et rapide (on sait bien que la grosse difficulté dans la mise au point d'un récupérateur est la phase de repérage...).

Cette méthode est quelque peu gênante dans la mesure où il se pose un choix : celui de h; la valeur idéale est le maximum de h_t , qui permet de réserver le maximum pour la zone II en évitant peut-être trop d'appels au récupérateur. Il n'y a pas de recette précise sinon l'essai et l'erreur.

5.4.3.2. DEUXIEME METHODE. Cette méthode vise une utilisation plus économique, commode et automatique du système LISP, mais au prix d'un récupérateur plus lourd et plus lent. Il s'agit de ne pas fixer la frontière AB entre les zones I et II mais de la faire évoluer avec le temps en la faisant coïncider à tout instant avec le sommet de la zone I. D'autre part, les cellules extraites de la liste libre sont prises consécutivement à partir du sommet, de sorte qu'en réalité il se crée une troisième zone au-dessus de la liste libre qui contient exactement les représentations internes des S-expressions traitées ou à traiter.

Toute la mémoire est donc utilisée (ou utilisable) à chaque instant. Le récupérateur est appelé lorsque les zones I et II se rejoignent. Mais il ne lui suffit plus de balayer la zone III, il doit encore recréer une configuration de la mémoire en trois zones, ce qui suppose les opérations suivantes :

- repérage et compactage des cellules utiles (fixer les dimensions de la zone III);
- déplacement des cellules utiles se trouvant hors de la future zone III dans cette zone;
- modification des adresses en conséquence.

Notons un avantage de la méthode par piles-listes : l'automaticité est totale puisqu'il n'y a qu'une seule zone et qu'elle est initialement organisée en structure de liste. Le récupérateur est assez simple et rapide, puisqu'il n'y a qu'une zone à balayer (comme dans la première méthode).

5.4.4. LE RÉCUPÉRATEUR

Le programme de récupération de cellules libres s'exécute automatiquement, sans intervention du programmeur, à partir du moment où la liste FREE est réduite à NIL.

Organisation générale : la distinction entre cellules utiles et cellules inaccessibles vient de la structure de liste. Toutes les informations sont organisées de cette manière dans l'ordinateur et toute cellule active fait partie d'une liste; son adresse s'obtient par parcours de la structure en question, d'autant plus facilement que le langage propose des primitives d'accès à ces adresses (fonctions CAR et CDR). Il suffit donc de connaître toutes les cellules de tête de toutes les listes qui se trouvent en mémoire; on les parcourt et on marque chaque cellule rencontrée. Les cellules non marquées sont inaccessibles et peuvent réintégrer la liste libre (voir chapitre IV).

La détermination des cellules de base se fait en étudiant l'implémentation des listes en mémoire centrale :

- avant la lecture des doublets composant le programme, il n'y a en mémoire centrale que les listes de propriétés (P-listes); les cellules de base sont toujours mises à une certaine place fixe;
- lors de la lecture, les doublets sont placés dans deux piles spéciales et les adresses de tête sont des cellules de base;
- lors de l'exécution d'un programme, les informations se trouvent en A1, A2, etc... si ce sont des arguments, en A0 si ce sont des résultats ou des résultats partiels:
- citons encore la cellule de tête de la pile-liste RETAD.

Les opérations du récupérateur sont donc les suivantes :

- préservation des mémoires de manoeuvre et des piles-listes en vue de sauvegarder les résultats partiels obtenus lors de l'appel au récupérateur;
- marquage des cellules actives;
- exploration de la zone de mémoire précédemment occupée par la liste libre initiale en vue d'enlever le bit de marquage et de replacer les cellules non marquées dans la liste libre;
- restitution des mémoires et des piles-listes préservées lors de la première opération.

Bibliographie

BERKLEY, E. et BOBROW, D. : The programming langage LISP, M.I.T., Cambridge, Mass. 1974.

BRANQUART, P. et LEWI, J. : "A scheme of storage allocation and garbage collection for Algol 68" in Algol 68 implementation, J.E.L. Peck (Ed.), North-Holland, Amsterdam, 1970.

COHEN, J. : "Garbage collection on linked data structures", Computing surveys, 13, 3, sept. 1981.

DEUTSCH, L. et BOBROW, D. : "An efficient incremental automatic garbage collector", Comm. ACM, 19, 9, sept. 1976.

DIJKSTRA, E. : A discipline of programming, Prentice-Hall, Engelwood, N.J. 1976.

KNUTH, D. : The art of computer programming, vol. 1 : Fundamental algorithms, Addison-Wesley, Reading, Mass. 1973.

LECHARLIER, B. : cours d'Introduction au langage LISP, FNDP Namur, 1985.

LEROY, H. : cours de Méthodologie de la programmation, FNDP Namur, 1984.

LEROY, H. : cours de Théorie des langages, FNDP Namur, 1984.

LEROY, H. : cours de Théorie des langages et questions spéciales de compilation, FNDP Namur, 1985.

RIBBENS, D. : Programmation non numérique LISP 1.5, Dunod, Paris, 1969.

TABLE DES MATIÈRES

Introduction

Chapitre I : Structures de données 1

1.1. Données 1
1.2. Description logique des structures de données .. 2
1.2.1. Juxtaposition 3
1.2.2. Enumération 3
1.2.3. Séparation des cas et partie variable ... 3
1.2.4. Récursion 4
1.3. Représentation interne 4
1.4. Remarques et conventions 6
1.4.1. Données et structures 6
1.4.2. Taille fixe et variable 6
1.4.3. Parties statiques et dynamiques 7
1.4.4. Pointeurs et adresses 7
1.4.4.1. Niveau de l'utilisateur 7
1.4.4.2. Niveau de l'implémentation 8

Chapitre II : Structures classiques 10

2.1. Différentes listes linéaires 10
2.2. Allocation séquentielle 11
2.2.1. Gestion séquentielle d'une liste 12
2.2.1.1. Piles 12
2.2.1.2. Files 13
2.2.1.3. Listes ouvertes 14
2.2.1.4. Remarques 15

2.2.2. Gestion séquentielle de plusieurs listes 15
2.2.2.1. Cas de deux piles 15
2.2.2.2. Cas de plusieurs piles 16
2.2.2.3. Algorithme de Garwick 18
2.3. Allocation chaînée 22
2.3.1. La liste libre 23
2.3.2. Implémentation des piles, files, listes ouvertes 26
2.3.2.1. Piles 26
2.3.2.2. Files 26
2.3.2.3. Listes ouvertes 28
2.4. Listes circulaires 28
2.5. Double chaînage 30
2.6. Structures arborescentes 33
2.7. Structures de liste 36

Chapitre III : Allocation dynamique 40

3.1. Introduction 40
3.2. La liste libre 41
3.2.1. Structure des blocs 41
3.2.2. Réservation 42
3.2.3. Libération 44
3.3. Le "buddy system" 46
3.4. Le théorème des 50 pourcents 49

Chapitre IV : Techniques de récupération 52

4.1. Introduction 52
4.2. Récupération de cellules unitaillées 53
4.2.1. Marquage 53
4.2.2. Réorganisation de la mémoire 62
4.3. Récupération de cellules de taille variable 64
4.3.1. Marquage 64
4.3.2. Réorganisation de la mémoire 65
4.4. Compteurs de références 66
4.5. Récupération et mémoire virtuelle 69

Annexe I : Aperçu de l'implémentation du LISP 72
5.1. Description générale 72
5.1.1. Le langage externe et les S-expressions . 72
5.1.2. Syntaxe du langage externe 73
5.1.3. La table des propriétés d'un symbole atomique 74
5.2. Représentation interne des S-expressions et structures de liste 75
5.2.1. Notations et terminologie 75
5.2.2. Représentation interne des S-expressions 76
5.2.3. Cellule de liste 77
5.2.4. Représentation interne des paires pointées et listes 78
5.2.5. Représentation interne des symboles atomiques 79
5.2.5.1. Cas des cellules de liste composées de deux mots-mémoire . 79
5.2.5.2. Cas des cellules de liste composées d'un seul mot-mémoire 82
5.2.6. Représentation interne des nombres 82
5.3. La liste libre 83
5.3.1. Introduction 83
5.3.2. Extraction d'une cellule de liste 84
5.3.3. Retour d'une cellule dans la liste libre 84
5.4. Organisation générale 84
5.4.1. Préservation par piles-listes 84
5.4.2. Piles-listes d'arguments 86
5.4.3. Autres méthodes d'implémentation 86
5.4.3.1. Première méthode 87
5.4.3.2. Deuxième méthode 88
5.4.4. Le récupérateur 89

Bibliographie 91

Table des matières 92