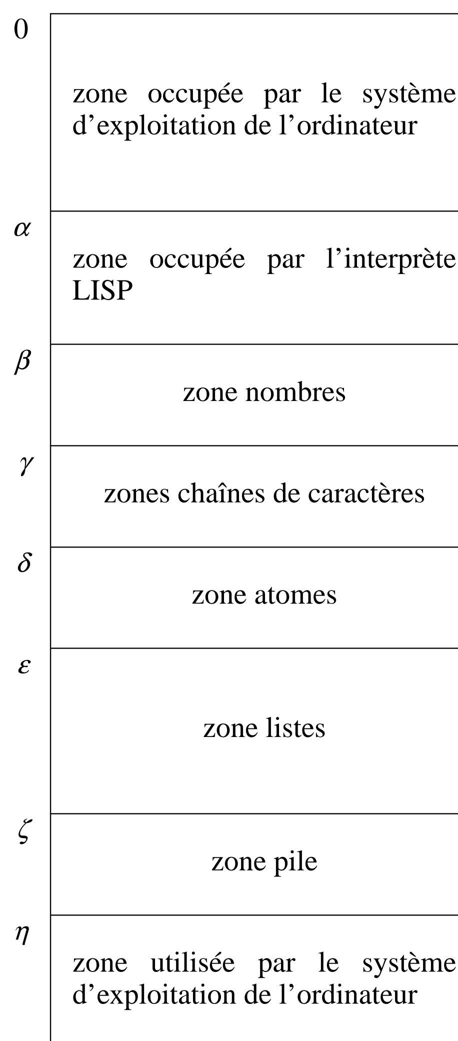
# 15. RETOUR VERS LES LISTES

## 15.1. L'ORGANISATION DE LA MEMOIRE LISP

Comment LISP utilise-t-il la mémoire de l'ordinateur ? Nous savons qu'en LISP nous avons toute sorte d'objets : les listes, les nombres, les chaînes de caractères, les atomes. Ils se trouvent dans des zones spéciales de la mémoire de votre ordinateur.

La figure de la page suivante montre graphiquement l'utilisation de l'espace mémoire de votre ordinateur sous LISP :[1]




Les numéros a' gauche du tableau représentent des adresses. Ainsi, dans notre figure, les premiers $\alpha$ mots

[1] L'organisation que nous proposons ici n'est pas la seule possible. D'autres systèmes LISP le font différemment. Nous proposons cette organisation puisqu'elle est, conceptuellement, la plus simple et expose néanmoins toutes les caractéristiques les plus importantes.

de la mémoire d'ordinateurs sont utilisés par le système d'exploitation de l'ordinateur. Naturellement, cette zone peut se situer ailleurs, son emplacement précis dépend de l'ordinateur particulier sur lequel votre LISP est implémenté. Ce qui compte, c'est que *quelque part* une zone (ou des zones) existe qui ne puisse pas eˆtre utilisée par LISP.

Regardons donc chacune des zones de LISP : la première, allant de l'adresse $\alpha$ a‘ l'adresse $\beta$-1, est occupée par l'interprète LISP meˆme. C'est ici qu'on trouvera les programmes d'entrée/sortie, la petite boucle *toplevel* que nous avons vu au chapitre 13, etc. Cette zone n'est pas accessible pour l'utilisateur : c'est la zone système de votre machine LISP, de meˆme que la première zone est la zone système de votre ordinateur.

Ensuite, il y a des zones pour chacun des objets existants en LISP : une zone pour stocker les nombres, une pour les chaînes de caractères, une pour les atomes et une pour les listes. Les listes ne se trouvent donc *que* dans la zone liste, les nombres que dans la zone nombre, etc.

La dernière zone de votre machine LISP est réservé pour la pile. C'est une zone de travail pour l'interprète : on y stocke, par exemple, des résultats intermédiares. Nous verrons l'utilité de cette zone plus tard. Concentrons-nous sur la zone liste, puisque c'est elle qui nous intéresse pour l'instant.

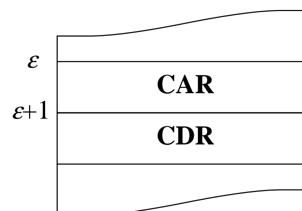
## 15.2. LA ZONE LISTE

Les listes ont été inventées pour pouvoir utiliser la mémoire sans contrainte de taille. Les idées de base sont :

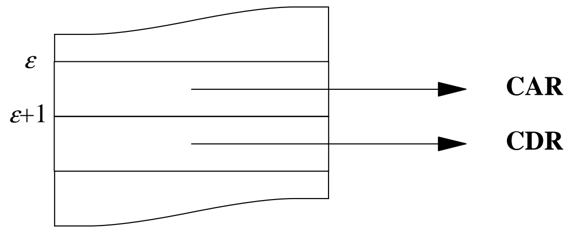
1. d'organiser cette mémoire de manière non continue : une liste ne doit pas obligatoirement occuper des mots mémoires contigus. Pour cela, les listes sont implémentées tel qu'il soit toujours possible a‘ partir d'*un* élément de la liste de trouver le suivant.
2. de disposer d'un algorithme de *récupération de mémoire*. Cet algorithme, qui s'appelle en anglais le *garbage collector*, doit pouvoir trouver des listes (ou des parties de listes) qui ne sont plus utilisées et il doit les inclure a‘ nouveau dans l'espace utilisable.

Tout cela a‘ l'air bien abstrait ! Examinons donc l'organisation de cette zone en plus de détail.

Une liste se constitue d'un **CAR** et d'un **CDR**, d'un premier élément et d'un reste. A l'intérieur de la machine, donc a‘ l'intérieur de la zone liste, ces deux parties se retrouvent en permanence, et le **CAR** et le **CDR** d'une liste occupent deux mots adjacent de cette zone :

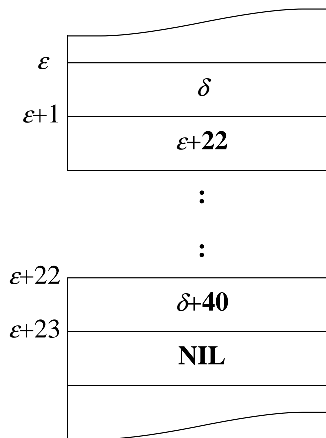



Chacune des boîtes représente un mot mémoire dans la zone liste. Les étiquettes **CAR** et **CDR** a‘ l'intérieur des boîtes sont des *pointeurs* vers les objets LISP constituant le **CAR** et le **CDR** de cette liste. Graphiquement :



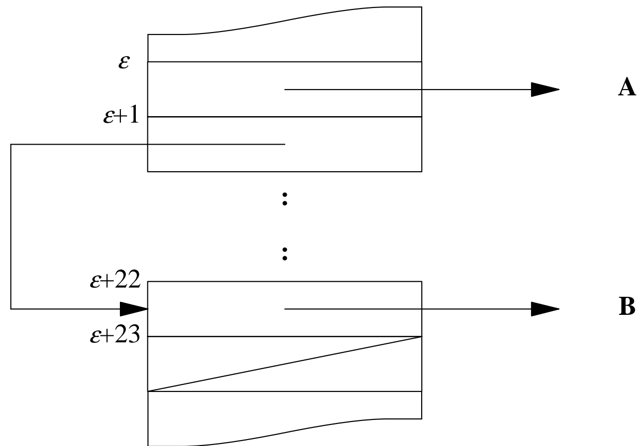

Le **CAR** ou le **CDR** d'une liste peuvent eˆtre de n'importe quel type : atome ou liste. De fait, les pointeurs CAR et CDR peuvent pointer dans n'importe quel *zone* des objets LISP. Ce sont donc, en réalité, des adresses de la mémoire.

Ainsi, disposant d'une liste **(A B)**, d'un atome **A** se trouvant en mémoire a‘ l'adresse $\delta$, et d'un atome **B** se trouvant a‘ l'adresse $\delta$+40, cette liste peut eˆtre implémentée en mémoire de la manière suivante :

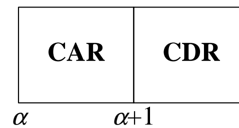



Les listes sont donc des couples (on dit : des *doublets*) de mots constituées d'un pointeur vers l'emplacement de la valeur du **CAR** et d'un pointeur vers l'emplacement de la valeur du **CDR**, c'est-a‘-dire : l'*adresse* du mot mémoire où se trouve la valeur correspondant. La fin d'une liste est indiquée par un pointeur vers l'atome *très* particulier **NIL**.

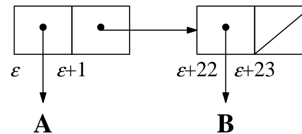
Afin de ne pas surcharger la représentation graphique par trop d'adresses, nous allons représenter les pointeurs vers des listes par des flèches vers les doublets correspondants, la fin d'une liste par une diagonale dans le **CDR** du dernier doublet et au lieu de marquer les adresses des atomes, nous allons juste indiquer des pointeurs vers les noms des atomes. Ce qui donne pour la meˆme liste **(A B)** :



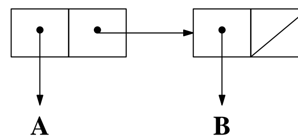

Bien évidemment, nous ne pouvons pas constamment prendre en considération les adresses réelles, c'est pourquoi nous utiliserons une représentation graphique des listes qui ne prend en considération que les *doublets* directement utilisés dans une liste. Une liste sera donc représentée comme une suite de *doublets* composée des **CAR**s et **CDR**s de cette liste :




La liste **(A B)** de ci-dessus sera donc représentée comme :

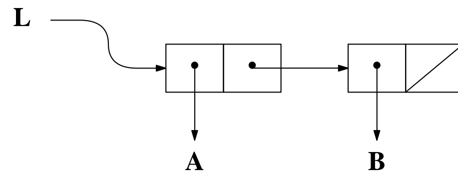



Normalement, quand nous allons donner une représentation graphique d'une liste, nous allons ignorer les adresses d'implémentation réelle et dessiner, par exemple, la liste **(A B)**, comme :



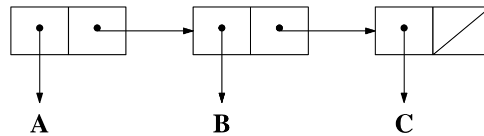

Mais, meˆme si nous prenons cette représentation (plutôt) abstraite, il ne faut jamais oublier qu'en réalité toute la représentation interne se fait en terme d'adresses réelles !

Si cette liste **(A B)** est la valeur de la variable **L**, au lieu de représenter graphiquement l'atome **L** avec comme C-valeur (cf §8.1) l'adresse de cette liste, nous allons représenter ce fait simplement par une flèche de **L** vers cette liste, comme ci-dessous :

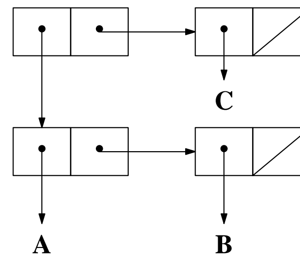



On peut aisément reconnaître ce que font la fonction **CAR** et **CDR** : elles ramènent respectivement le pointeur du champ **CAR** et **CDR**.

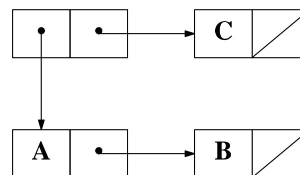
Regardons quelques exemples de cette représentation : La liste **(A B C)** sera donc implémentée comme :




et la liste **((A B) C)** sera, en terme de doublets, implémentée comme :

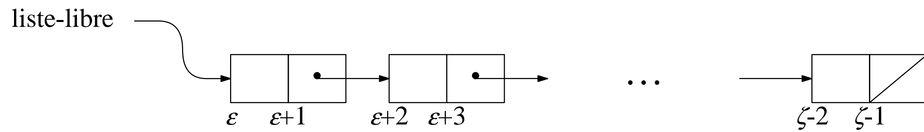



Nous pouvons encore simplifier en ne fléchant que les pointeurs vers des listes; ce qui donne pour cette dernière liste :




N'oublions jamais que tout pointeur et tout atome n'est - en réalité - qu'une adresse: l'adresse mémoire où se trouve l'objet correspondant.

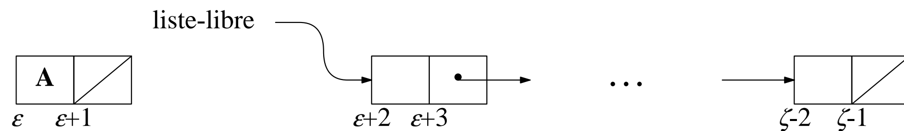
Au lancement de LISP toute la zone liste est organisée comme une *liste libre*, c'est-a'-dire : tous les doublets sont *chaînés* entre eux pour former une énorme liste :




Chaque appel de la fonction **CONS** enlève un doublet de cette liste libre. Ainsi, si le premier appel de **CONS** est

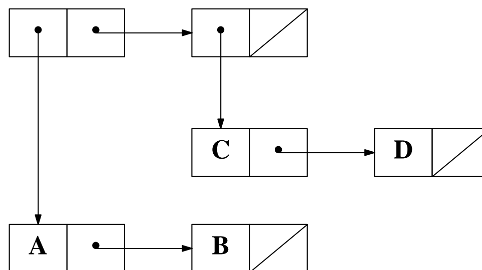
**(CONS ’A NIL)**

**CONS** cherche d’abord un doublet dans la liste libre, l’enlève de cette liste, insère dans le champ **CAR** de ce doublet un pointeur vers l’atome **A** et dans le champ **CDR** de ce meˆme doublet un pointeur vers l’atome **NIL**. ce qui donne:

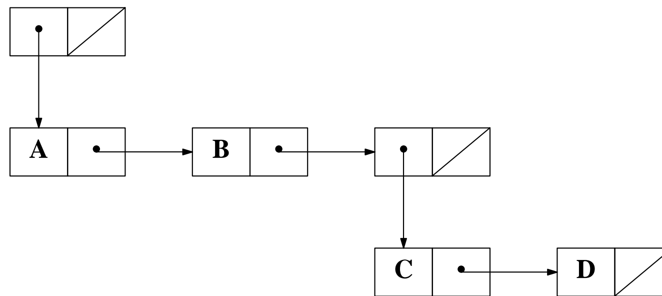



Quand la liste libre est vide, **CONS** met en marche un mécanisme de récupération de mémoire, qui construit une nouvelle liste libre a‘ partir des doublets qui ne sont plus nécessaires, c’est-a‘-dire : a‘ partir des doublets qui ne sont plus utilisés.

Ci-dessous encore deux exemples de représentation de listes sous forme de doublets : d’abord la liste **((A B) (C D))** :

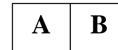



et ensuite la liste **((A B (C D)))** :




Si nous avons dit que le contenu du champ **CAR** ou du champ **CDR** d’un doublet peut eˆtre un pointeur vers n’importe quel objet LISP, il peut s’avérer que nous trouvions dans le champ **CDR** d’un doublet un pointeur non pas vers une liste mais un pointeur vers un atome, comme, par exemple dans le doublet

ci-dessus :




ou encore dans la liste suivante :

Actuellement, nous ne pouvons pas représenter ces deux listes de façon externe. C'est pourquoi, pour lire ou écrire de telles listes, nous avons besoin d'une notation supplémentaire : pour représenter des listes qui contiennent dans le champ **CDR** un pointeur vers un atome différent de **NIL**[2] nous utiliserons le caractère '**.**' (point) comme séparateur entre le champ **CAR** et le champ **CDR** de telles doublets. Ce qui donne, pour la première liste

**(A . B)**

et, pour la deuxième liste

**(A B . C)**

Naturellement, ceci change notre définition de la fonction **CONS** du chapitre 2 : plus aucune raison n'existe pour demander que le deuxième argument du **CONS** soit une liste. Bien au contraire, la liste **(A . B)** est le résultat de l'appel suivant :

**(CONS 'A 'B)**

et le **CDR** de cette liste est l'*atome* **B** !

Bien évidemment, nous avons besoin d'une représentation externe (comme suite de caractères) reflétant l'*implémentation* des listes sous formes de doublets. Pour cela, nous allons reprendre la définition classique des expressions symboliques (ou des S-expressions) de John McCarthy, l'inventeur de LISP :

Tout objet LISP est une S-expression. Le type le plus élémentaire d'une S-expression est un *atome*. Tout les autres S-expressions sont construites a‘ partir de symboles atomiques, les caractères "(", ")" et "**.**" (le point). L'opération de base pour former une S-expression consiste a‘ en combiner deux pour produire une S-expression plus grande. Par exemple, a‘ partir des deux symboles **A** et **B**, on peut former la S-expression **(A . B)**.

La règle exacte est : une S-expression est soit un symbole atomique soit une composition des éléments suivants (dans l'ordre) : une parenthèse ouvrante, une S-expression, un point, une S-expression et une parenthèse fermante.

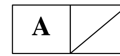
Voici, suivant cette définition, quelques exemples de S-expressions et leur représentation interne sous forme de doublets :

**A**

évidemment, c'est un atome et ne peut donc pas eˆtre représenté sous forme de doublets.

**(A . NIL)**

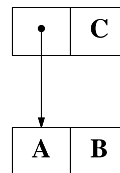
[2] rappelons qu'un pointeur vers l'atome **NIL** dans le champ **CDR** indique la fin d'une liste.



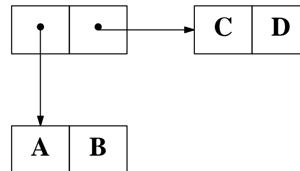

**(A . (B . NIL))**

**(A . B)**

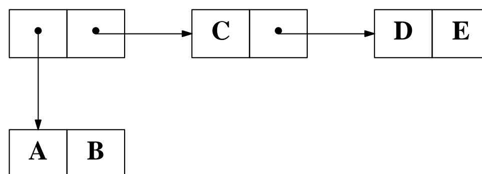
**((A . B) . C)**




**((A . B) . (C . D))**




**((A . B) . (C . (D . E)))**




Pour passer de cette notation des S-expressions a‘ la notation des listes a‘ laquelle nous sommes habituées, il suffit de

1. remplacer chaque occurrence d'un point suivi d'une parenthèse ouvrante par un caractère blanc et d'éliminer la parenthèse fermante correspondante, ce qui transforme, par exemple, **(A . (B . NIL))** en **(A B . NIL)**,
2. éliminer les occurrences d'un point suivi par **NIL**, ce qui transforme, par exemple, **(A B . NIL)** en **(A B)**.

Ainsi, pour les S-expressions de ci-dessus nous avons les correspondances que voici :

| | | |
|---|---|---|
| **A** | ≡ | **A** |
| **(A . NIL)** | ≡ | **(A)** |
| **(A . (B . NIL))** | ≡ | **(A B)** |
| **(A . B)** | ≡ | **(A . B)** |
| **((A . B) . C)** | ≡ | **((A . B) . C)** |
| **((A . B) . (C . D))** | ≡ | **((A . B) C . D)** |
| **((A . B) . (C . (D . E)))** | ≡ | **((A . B) C D . E)** |

Naturellement, les *paires pointées*, c'est-a'-dire : les doublets où le **CDR** est un pointeur vers un atome différents de **NIL**, restent représentés a' l'aide du caractère '**.**'.

## 15.3. LES FONCTIONS DE MODIFICATION DE LISTES

Les fonctions de traitement de listes que nous connaissons jusqu'a' maintenant, **CAR**, **CDR**, **REVERSE** etc, ne modifient pas les listes données en argument. Ainsi, si a' un instant donné la valeur de la variable **L** est la liste **(A B C)**, le calcul du **CDR** de **L** livre bien la liste **(B C)**, et la valeur de **L** est toujours l'ancienne liste **(A B C)**.

Dans ce paragraphe nous étudierons quelques fonctions de traitement de listes *destructives*, c'est-a'-dire, qui modifient après évaluation les listes passées en argument.

Les deux fonctions les plus importantes dans ce domaine sont les fonctions **RPLACA** et **RPLACD**. La première *modifie physiquement* le **CAR** de son premier argument, la deuxième *modifie physiquement* le **CDR** de son argument. Voici leur définition :

**(RPLACA** *liste expression***)** → *nouvelle-liste*
**RPLACA** remplace physiquement le **CAR** de la *liste* premier argument par la valeur de l'*expression* donnée en deuxième argument. La *liste* ainsi modifiée est ramenée en valeur.

**(RPLACD** *liste expression***)** → *nouvelle-liste*
**RPLACD** remplace physiquement le **CDR** de la *liste* premier argument par la valeur de l'*expression* donnée en deuxième argument. La *liste* ainsi modifiée est ramenée en valeur.

Voici quelques exemples :

| | | |
|---|---|---|
| **(RPLACA '(A B C) 1)** | → | **(1 A B)** |
| **(RPLACA '((A B) (C . D)) '(1 2 3))** | → | **((1 2 3) (C . D))** |
| **(RPLACD '(A B C) 1)** | → | **(A . 1)** |
| **(RPLACD '((A B) (C . D)) '(1 2 3))** | → | **((A B) 1 2 3)** |

Dans les exemples ci-dessus nous utilisons ces fonctions pour calculer une valeur. Mais pensons aussi au fait que ces deux fonctions *modifient* les listes données en argument. Pour cela, regardons donc la fonction suivante :

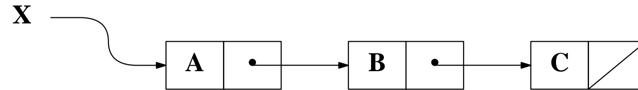
```
(DE FOO (X)
    (CONS (RPLACA X 1) X))
```

Que fait cette fonction ? Est-ce que le résultat est le meˆme si nous remplaçons le corps par **(CONS (CONS 1 (CDR X)) X)** ?

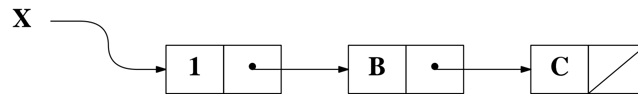
Regardons donc en détail. Si nous appelons cette fonction comme :

**(FOO '(A B C))**

la variable **X** sera liée à la liste **(A B C)**. Avant de commencer l'évaluation du corps de la fonction nous avons donc :




L'évaluation du corps de la fonction se réduit à l'évaluation du **CONS**. Il faut donc d'abord évaluer les deux arguments **(RPLACA X 1)** et **X**. **RPLACA** modifie la liste **X**, ce qui donne :




et c'est cette nouvelle liste qui sera le premier argument pour **CONS**.

L'évaluation de **X** (deuxième argument) produira également cette nouvelle liste **(1 B C)**, puisque maintenant cette liste est la valeur de **X**. Le résultat de l'appel **(FOO '(A B C))** est donc la liste

**((1 B C) 1 B C)**

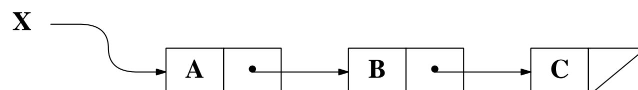
et non point la liste **((1 B C) A B C)**.

Quand vous utilisez ces fonctions, faites attention, leur usage est très dangéreux, soyez sûrs que vous voulez vraiment *modifier* la structure interne des listes !

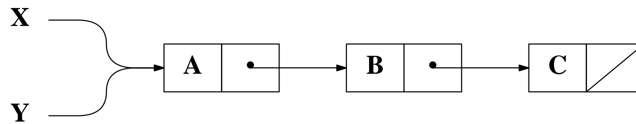
Imaginons que la personne qui a construit le programme **FOO** ci-dessus veuille, en réalité, construire une fonction qui produise une liste contenant en **CAR** la même liste que le **CDR** avec le premier élément remplacé par l'atome **1**, qu'il voulait donc que **(FOO '(A B C))** livre **((1 A B) A B C)**. Constatant l'erreur, elle construit alors la nouvelle version que voici :

```
(DE FOO (X)
   (LET ((X X) (Y X))
        (CONS (RPLACA X 1) Y)))
```

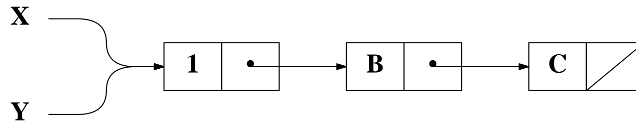
Est-ce que cette solution est meilleure ? L'idée est claire : puisque **RPLACA** modifie la liste, on sauvegarde, avant de faire le **RPLACA**, la valeur de **X** dans la variable **Y**, et cette variable sera ensuite le deuxième argument du **CONS**. Regardons donc ce qui se passe, et appelons à nouveau **(FOO '(A B C))**. Comme préalablement, la variable **X** sera liée à la liste **(A B C)** :




ensuite, dans le **LET**, **X** sera reliée à cette même liste, et **Y** sera liée à la valeur de **X** ce qui donne :

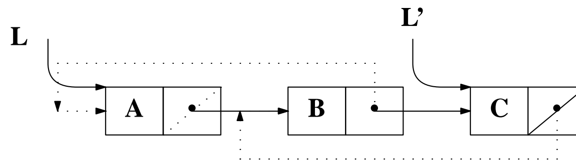



ensuite, comme préalablement, l’évaluation du premier argument du **CONS** livre la liste **(1 B C)** et modifie la valeur de **X**, ce qui donne :

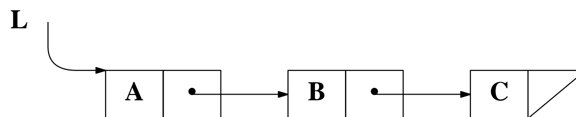



Etant donnée que la variable **X** *et* la variable **Y** sont liées a‘ la *meˆme* liste (c’est-a‘-dire : la C-valeur de **X** et la C-valeur de **Y** contiennent la meˆme adresse de la zone liste), la fonction **RPLACA** modifie les valeurs de tous les deux variables ! Ainsi, meˆme avec cette sauvegarde initiale de la valeur de **X**, rien n’est changé : on se trouve dans le meˆme cas que précédemment, c’est-a‘-dire : le résultat sera la liste **((1 B C) 1 B C)**.

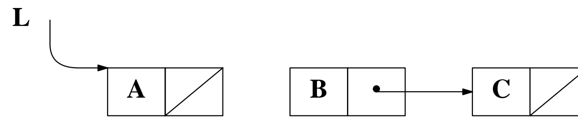
Faisons un petit exercice et écrivons une fonction **FREVERSE** qui doit inverser une liste *sans* construire une nouvelle liste, c’est-a‘-dire : sans faire un seul appel a‘ la fonction **CONS**. Pour résoudre ce problème nous changerons les pointeurs de la liste originale, telle que les *meˆmes* doublets seront, après exécution de la fonction, chaînés a‘ l’envers du chaînage original. Ci-dessous, la liste **L** aux trois éléments **A**, **B** et **C**, où j’ai dessiné en pointillé les pointeurs tels qu’ils doivent eˆtre après inversement de la liste et où **L’** est la valeur de **L** a‘ la fin :



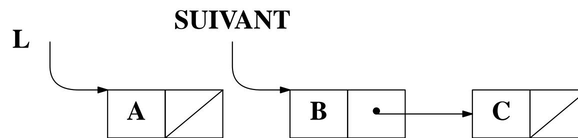

Regardons, sur une trace graphique comment nous devons procéder. Au début nous avons un pointeur, nommons le **L**, vers la liste initial :




Evidemment, la première chose a‘ faire est de remplacer le **CDR** du premier doublet par **NIL**, ce qui nous donne :

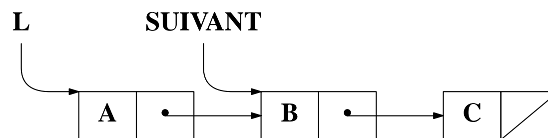


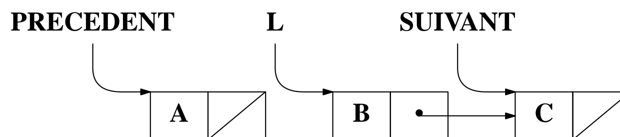

Le pas suivant consiste a‘ remplacer le pointeur du **CDR** du deuxième doublet par un pointeur vers ce premier doublet juste modifié. Mais, après cette opération du *remplacement* du **CDR** du premier doublet par **NIL**, nous n'avons plus aucun accès au deuxième doublet. Donc : avant de faire cette opération nous devons garder quelque part un pointeur vers la suite de la liste. Appelons ce pointeur **SUIVANT**. Nous devons donc avoir la configuration suivante :

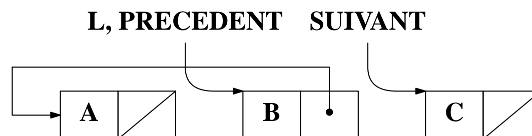


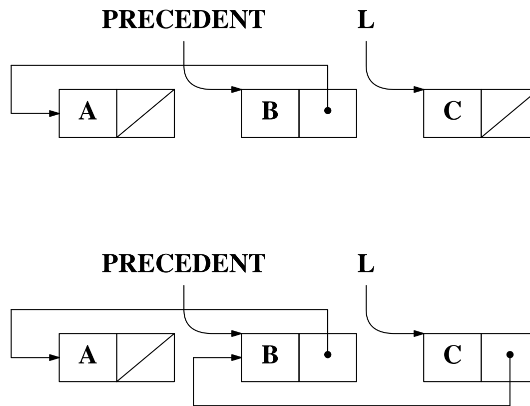

Si maintenant nous remplaçons le **CDR** du deuxième doublet par un pointeur vers la valeur de **L**, nous perdons a‘ nouveau la suite de la liste. Nous devons donc d'abord avancer le pointeur **SUIVANT**, mais sans perdre sa valeur actuelle, puisque, dans la suite nous devons établir un pointeur vers ce doublet pointé actuellement par **SUIVANT**.

La solution qui s'impose sera donc d'introduire un pointeur supplémentaire, nommons-le **PRECEDENT**, et que **L** pointera a‘ tout instant sur la nouvelle teˆte de liste, **PRECEDENT** pointera vers le doublet qui précédait dans la liste originale l'élément pointé par **L**. Le pointeur **SUIVANT** indiquera en permanence les éléments suivants, pas encore modifiés. Ce qui nous donne la suite d'opérations ci-après :






Voici donc la définition LISP de **FRERVERSE** :

```
(DE FREVERSE (L)
  (LET ((L L) (PRECEDENT))
       (IF L (SELF (CDR L) (RPLACD L PRECEDENT))
             PRECEDENT)))
```

Dans cette version nous n’avons pas besoin d’un pointeur **SUIVANT** explicite puisque la liaison de **L** au **CDR** de **L**, donc a‘ l’adresse du doublet suivant se fait en parallèle a‘ la destruction de ce meˆme champ **CDR** !

Voila‘ donc une fonction qui inverse une liste sans faire le moindre **CONS**, c’est-a‘-dire : une fonction qui n’utilise aucun doublet supplémentaire. **FREVERSE** sera donc sensiblement plus rapide que la fonction **REVERSE** que nous avons vu au chapitre 6.

Mais n’oublions jamais que **FREVERSE** est *destructive*, et qu’il faut prendre les meˆmes précautions qu’avec la fonction **FOO** ci-dessus. Cela signifie que si nous avons, par exemple, la fonction :

```
(DE BAR (L)
   (APPEND (FREVERSE L) L))
```

l’appel **(BAR ’(A B C))** ne nous livre pas la liste **(C B A A B C)**, mais la liste **(C B A A)**. Vous voyez pourquoi ?

Les deux fonctions **RPLACA** et **RPLACD** sont suffisantes pour faire n’importe quelle opération (destructive) sur des listes. Ainsi nous pouvons définir une fonction de concaténation de deux listes :

```
(DE LAST (L) (IF (ATOM (CDR L)) L (LAST (CDR L))))

(DE NCONC (LISTE1 LISTE2)
   (RPLACD (LAST LISTE1) LISTE2) LISTE1)
```

Nous pouvons construire une fonction **ATTACH** qui insère un nouvel élément en teˆte d’une liste, de manière telle que le premier doublet de la liste originale soit également le premier doublet de la liste résultat :

```
(DE ATTACH (ELE LISTE)
   (LET ((AUX (LIST (CAR LISTE))))
        (RPLACD (RPLACA LISTE ELE) (RPLACD AUX (CDR LISTE)))))
```

Nous pouvons construire la fonction inverse de **ATTACH** : **SMASH**, qui enlève physiquement le premier élément d'une liste, de manière telle que le premier doublet de la liste originale soit également le premier doublet de la liste résultat :

```
(DE SMASH (LISTE)
    (RPLACD (RPLACA LISTE (CADR LISTE)) (CDDR LISTE)))
```

Toutes ces fonctions ont en commun le fait de modifier physiquement les structures des listes données en arguments, et donc que les valeurs de *toutes* les variables qui pointent vers une telle liste se trouvent modifiées.

## 15.4. LES FONCTIONS EQ ET EQUAL

Au chapitre 6 nous avons construit la fonction **EQUAL**, qui teste l'égalité de deux listes en traversant les deux listes en parallèle et en comparant l'égalité des atomes avec la fonction **EQ**. Cette fonction teste donc si les deux listes ont la même structure et si elles sont composées des mêmes atomes dans le même ordre.

La fonction **EQ**, nous disions, ne peut tester que l'égalité des atomes. Précisons : chaque atome LISP n'existe qu'une seule fois. Ainsi, si nous avons l'expression :

**(CONS 'A '(A B C))**

les deux occurrences de l'atome **A** sont deux occurrences d'un pointeur vers le *même* atome. Pour prendre un autre exemple, chaque occurrence de l'atome **CONS** à l'intérieur d'un programme se réfère précisément au même atome ! De ce fait, le test d'égalité de la fonction **EQ** est en réalité un test d'égalité de deux adresses (et non point un test d'égalité d'une suite de caractères). C'est donc un test d'identité : effectivement, les deux atomes **A** dans l'expression ci-dessus sont des références à un seul et unique objet.

Sachant que **EQ** se réduit à une comparaison d'adresses, et sachant que des listes sont également accessibles à travers leurs adresses, nous pouvons utiliser cette fonction pour tester l'*identité* de deux listes.

Afin d'expliciter construisons deux listes (que nous nommerons **X** et **Y**) : la première, **X**, sera construite par l'appel

**(LIST '(A B) '(A B))**

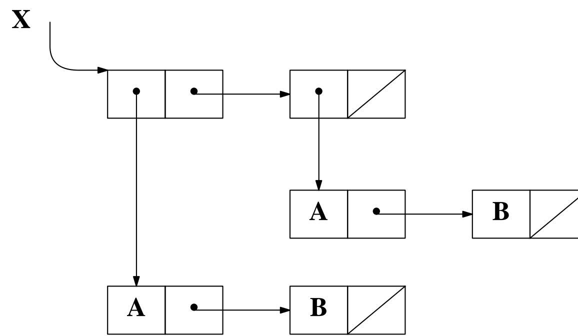
et la deuxième, **Y**, par l'appel :

**(LET ((X '(A B))) (LIST X X))**

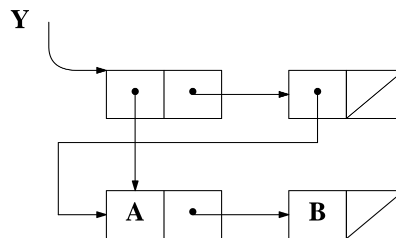
Regardez bien la différence entre ces deux listes ! Les deux listes s'écrivent de façon externe :

**((A B) (A B))**

mais, de façon interne, la première est représentée par :




et la deuxième par :




Bien que ces deux dernières listes soient - de façon interne - très différentes, elles ont toutes les deux une écriture externe identique !

La différence entre la liste **X** et la liste **Y** devient apparente si nous testons l'identité des deux sous-listes :

| | | |
|---|---|---|
| **(EQUAL (CAR X) (CADR X))** | → | **T** |
| **(EQUAL (CAR Y) (CADR Y))** | → | **T** |
| **(EQ (CAR X) (CADR X))** | → | **NIL** |
| **(EQ (CAR Y) (CADR Y))** | → | **T** |

Evidemment, le premier et le deuxième élément de **Y** sont des pointeurs vers la meˆme liste : la fonction **EQ** livre donc **T**.

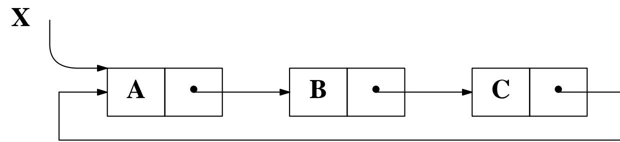
On dit que de telles listes sont *partagées*. Au niveau de l'impression standard, aucun moyen n'existe pour distinguer des listes partagées des listes non-partagées. Naturellement, des listes partagées sont toujours issues d'opérations sur des listes (c.a‘.d.: elles ne peuvent pas eˆtre données tel quel au terminal).

## 15.5. LES LISTES CIRCULAIRES

Regardons encore quelques listes partagées : quel est le résultat de l'opération suivante ?

**(LET ((X '(A B C))) (NCONC X X))**

**NCONC** concatène deux listes : nous concaténons donc a‘ la liste **X**, c'est-a‘-dire : **(A B C)**, la liste **X** elle-meˆme, ce qui donne :




Il s’agit d’une liste *circulaire* : une liste sans fin. Son impression donne quelque chose comme :

**(A B C A B C A B C A B C . . . )**

Donc, a‘ priori, il n’est pas possible d’imprimer de telles listes. Remarquons que ces listes peuvent eˆtre très utiles pour représenter des structures de données infinies, mais qu’elles sont très dangereuses : beaucoup de fonctions de recherche risquent de ne jamais se terminer sur de telles listes. Par exemple, si nous cherchons l’atome **D** a‘ l’intérieur de cette liste **X**, l’appel

**(MEMQ ’D X)**

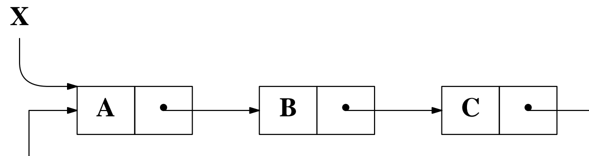
va boucler éternellement, puisque la fonction **MEMQ** compare le premier élément de la liste avec l’argument cherché, et si ce n’est pas le meˆme, elle recommence l’opération sur le **CDR** de la liste, ceci jusqu’a‘ ce qu’elle arrive soit a‘ une occurrence de l’élément cherché soit a‘ la fin de la liste. Etant donné que notre liste **X** n’a pas de fin, la recherche de l’élément qui effectivement ne se trouve pas dans la liste, n’a pas de fin également.

Pour écrire une fonction qui cherche l’occurrence d’un élément donné dans une liste circulaire, nous avons donc besoin de garder en permanence un pointeur vers le début de la liste, afin de pouvoir tester, avec la fonction **EQ**, si le **CDR** de la liste est ’**EQ**ual’ a‘ la liste entière et ainsi d’éviter de répeter la recherche a‘ l’infinie.

Ci-après la définition du prédicat C-MEMQ testant l’occurrence d’un élément a‘ l’intérieur d’une liste circulaire :

```
(DE C-MEMQ (OBJ L)
    (LET ((LL L)) (COND
         ((ATOM LL) (EQ LL OBJ))    ; le test d’égalité ;
         ((EQ L (CDR LL)) ())       ; le test d’arreˆt !! ;
         (T (OR (SELF (CAR LL)) (SELF (CDR LL)))))))
```

Si on appelle cette fonction avec l’atome **D** comme premier argument, et comme deuxième argument la liste circulaire :



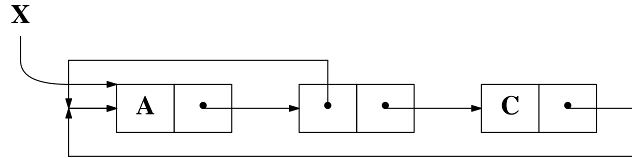

le résultat est **NIL**.

Naturellement, si le premier argument se trouve a‘ l’intérieur de la liste le résultat est **T**.

Modifions-donc notre liste **X** avec l’opération suivante :

**(RPLACA (CDR X) X)**

Cette opération nous livre une liste doublement circulaire : une boucle sur un **CAR**, et une boucle sur un **CDR** :

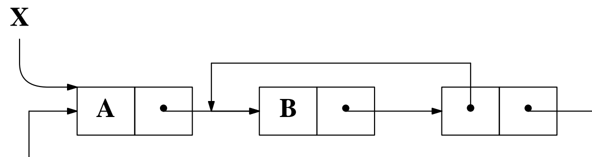



**C-MEMQ**, la fonction de recherche d'un élément dans une liste circulaire ne marche plus sur cette liste, car aucun test n'est effectué vérifiant la circularité sur le **CAR** de la liste.

Pour généraliser notre fonction **C-MEMQ** nous devons tester aussi bien le **CAR** que le **CDR** de la liste en vue d'une circularité vers le début de la liste. Voici alors la nouvelle fonction **C-MEMQ** :

```
(DE C-MEMQ (OBJ L)
   (IF (EQ (CAR L) OBJ) T
      (LET ((LL (CDR L))) (COND
            ((ATOM LL) (EQ LL OBJ))
            ((EQ L LL) ())
            (T (OR (SELF (CAR LL)) (SELF (CDR LL))))))))
```

Finalement, nous pouvons généraliser notre approche a‘ des listes a‘ circularités arbitraires, c'est-a‘-dire a‘ des circularités qui ne portent que sur des sous-parties de la liste, comme par exemple la liste **X** ci-dessous :




Dans ces cas généraux, en plus des tests de circularités 'globales' a‘ partir des champs **CAR** et **CDR**, il faut garder une pile de toutes les sous-listes rencontrées, afin de reconnaître une circularité sur des sous-structures arbitraires. Ce qui donne :[3]

```
(DE C-MEMQ (OBJ L)
   (LET ((L L) (AUX NIL)) (COND
         ((ATOM L) (EQ OBJ L))
         ((MEMQ L AUX) ())
         (T (OR (SELF (CAR L) (CONS L AUX))
                (SELF (CDR L) (CONS L AUX)))))))
```

Evidemment, la fonction **MEMQ** utilisée dans la ligne

**((MEMQ L AUX) ())**

est la fonction standard **MEMQ**, dont la définition se trouve au chapitre 6 dans l'exercice 4 : c'est une fonction qui procède au test d'égalité avec la fonction **EQ** !

[3] Cette version de la fonction **C-MEMQ** est inspirée de la fonction **SKE** de Patrick Greussay.

## 15.6. LES AFFECTATIONS

Après tant de fonctions de modifications terminons ce chapitre par un bref regard sur les fonctions de modifications des valeurs de variables.

Jusqu'a‘ maintenant, l'unique moyen que nous connaissons pour donner des valeurs a‘ des variables est la *liaison dynamique*, c'est-a‘-dire : la liaison des valeurs a‘ des variables pendant les appels de fonctions utilisateurs. Cette liaison dynamique se comporte de manière telle, qu'a‘ l'appel d'une fonction les variables prennent les nouvelles valeurs, déterminées par les arguments de l'appel, et qu'a‘ la sortie de la fonction les anciennes valeurs de ces variables soient restituées.

En LISP, il existe deux autres manières pour donner des valeurs a‘ des variables en utilisant les fonctions d'*affectation* **SETQ** et **SET**. La fonction **SETQ** prend - au minimum - deux arguments, le premier doit eˆtre le nom d'une variable, le second une expression LISP quelconque. L'*effet* de l'évaluation d'un appel de la fonction **SETQ** est de donner a‘ la variable premier argument comme nouvelle valeur le résultat de l'évaluation de l'expression deuxième argument. C'est donc une fonction de modification de la C-valeur des atomes (sans possibilité de restauration automatique ultérieure). La *valeur* de l'évaluation d'un appel de la fonction **SETQ** est le résultat de l'évaluation de l'expression.

Voici un exemple d'une interaction avec LISP utilisant cette fonction :

```
? (SETQ X 1)          ; X prend la valeur numérique 1
= 1
? X                   ; examinons la valeurs de X
= 1
? (SETQ X (+ X 3))
= 4
? X                   ; examinons la nouvelle valeur de X
= 4
```

et voici une manière élégante d'échanger les valeurs numérique de **X** et **Y** sans passer par une variable intermédiare :

```
? (SETQ X 10)         ; la valeur initiale de X
= 10
? (SETQ Y 100)        ; la valeur initiale de Y
= 100
? (SETQ X (- X Y))
= -90
? (SETQ Y (+ X Y))
= 10
? (SETQ X (- Y X))
= 100
? X                   ; la valeur de X est bien
= 100                 ; l'ancienne valeur de Y
? Y                   ; et la valeur de Y est bien
= 10                  ; l'ancienne valeur de X
```

La fonction **SETQ** est principalement utilisée dans la construction et la modification de bases de données globales a‘ un ensemble de fonctions, elle permet de donner a‘ l'extérieur de toute fonction des valeurs a‘ des variables.

Notons finalement qu'en général la fonction **SETQ** admet 2*$n$ arguments. Voici alors la définition formelle de **SETQ** :

**(SETQ** $var_1$ $val_1$ . . . $var_n$ $val_n$**)** → *valeur(*$val_n$*)*
et la C-valeur de $var_1$ est le résultat de l'évaluation de $val_1$, . . ., la C-valeur de $var_n$ est le résultat de l'évaluation de $val_n$.

La fonction **SET** est identique a‘ **SETQ**, mise a‘ part le fait qu'elle évalue son premier argument. L'évaluation du premier argument doit donc livrer le nom d'une variable. En guise de définition de la fonction **SET** disons que les expressions

**(SETQ** *variable expression***)**

et

**(SET (QUOTE** *variable***)** *expression***)**

sont équivalentes. La fonction **SET** permet des *indirections*. Par exemple, la suite d'instructions :

```
(SETQ A 'B)
(SET A 3)
```

donne a‘ la variable **B** la valeur numérique **3**.

A titre d'exemple, voici une version supplémentaire de la fonction **REVERSE**. Sa particularité est de modifier la valeur de la variable donnée en argument; la voici :

```
(DF PREVERSE (L RES)
    (LET ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES (CONS (CAR LISTE) RES)
                LISTE (CDR LISTE))
            (SELF LISTE))))
```

Voici une utilisation de cette fonction :

```
? (SETQ K '(A B C))     ; pour donner une valeur a‘ la variable K
= (A B C)
? (REVERSE K)           ; appelons la fonction REVERSE standard
= (C B A)               ; le résultat
? K                     ; examinons la valeur de K
= (A B C)               ; ça doit bien e^tre la me^me
? (PREVERSE K)          ; appelons maintenant la nouvelle fonction
= (C B A)               ; le résultat
? K                     ; regardons la valeur de K
= (C B A)               ; elle a bien été modifiée
                        ; voila toute la différence
```

D'ailleurs, la fonction **SETQ** peut e^tre définie avec la fonction **SET** de la manière suivante :

```
(DF SETQ (X)
    (LET ((VARIABLE (CAR X)) (VALEUR (EVAL (CADR X))) (REST (CDDR X)))
        (SET VARIABLE VALEUR)
        (IF REST (EVAL (CONS 'SETQ REST)) VALEUR)))
```

Bien évidemment, en LE_LISP cette définition s'ecrit :

```
(DF SETQ X
    (LET ((VARIABLE (CAR X)) (VALEUR (EVAL (CADR X))) (REST (CDDR X)))
        (SET VARIABLE VALEUR)
        (IF REST (EVAL (CONS 'SETQ REST)) VALEUR)))
```

## 15.7. EXERCICES

**1.** Montrez graphiquement l'effet des instructions suivantes :

a. **(SET 'X '(A B C)) (ATTACH 1 X)**

b. **(LET ((X '(A B C))) (SMASH X))**

c. **(SETQ X '(A B C)) (RPLACA (RPLACD X X) X)**

2. Ecrivez la fonction **FDELQ** a' deux arguments **ELE** et **LISTE**, qui enlève physiquement toutes les occurrences de l'élément **ELE** a' l'intérieur de la liste **LISTE**. (Attention au cas où la liste commence par l'élément **ELE**.)

3. Ecrivez une fonction qui inverse physiquement une liste circulaire simple.

4. Modifiez la fonction **FREVERSE** de manière à inverser une liste sur tous ses niveaux.

5. La suite d'instructions

**(SETQ** *variable* **(CAR** *la-liste***))**
**(SETQ** *la-liste* **(CDR** *la-liste***))**

est très courante. C'est pourquoi beaucoup de systèmes LISP ont une fonction **NEXTL** définit comme suit :

**(NEXTL** *variable***)** → **(CAR** *(valeur (variable))***)** et *variable* prend comme nouvelle valeur **(CDR** *variable***)**.

Avec cette fonction, la fonction **PREVERSE** peut e^tre redéfinie comme suit :

```
(DF PREVERSE (L RES)
    (LET ((LISTE (EVAL (CAR L))))
        (IF (NULL LISTE) (SET (CAR L) RES)
            (SETQ RES (CONS (NEXTL LISTE) RES))
            (SELF LISTE))))
```

Definissez alors cette fonction **NEXTL**.

6. Definissez la fonction **RPLACB** qui combine les fonctions **RPLACA** et **RPLACD** en remplaçant le **CAR** de la liste premier argument par le **CAR** de la liste deuxième argument et le **CDR** de la liste premier argument par le **CDR** de la liste deuxième argument. Ainsi si la variable **X** a la valeur **(X Y Z)**, le résultat de l'appel

**(RPLACB X '(1 2 3))**

affecte la valeur **(1 2 3)** à la variable **X**. De plus le premier doublet de la liste originale se trouve à la même adresse que le premier doublet de la liste résultat.

Dans LE_LISP, cette fonction s'appelle **DISPLACE**.