13. LES ENTREES / SORTIES (DEUXIEME PARTIE)

Il est indispensable de disposer de fonctions permettant la lecture des objets sur lesquels nous voulons travailler. Tant que tous les objets de travail sont soit déjà à l'intérieur du programme, soit donnés en argument aux appels des fonctions, le système LISP se charge, de manière implicite, de lire les données. Par contre, si nous voulons au fur et à mesure de l'exécution d'un programme entrer des données supplémentaires, nous devons utiliser des appels explicites à des fonctions de lecture. Tout système LISP a au moins une telle fonction : c'est la fonction **READ**. Elle n'a pas d'arguments, et ramène en valeur l'expression LISP lue dans le flot d'entrée.

Avant de continuer avec cette fonction de lecture, esquissons les problèmes liés aux entrées/sorties : dans l'interaction normale avec votre système LISP, vous entrez les fonctions et les données sur le clavier de votre terminal et vous recevez les résultats imprimés sur l'écran de votre terminal. Ceci veut dire : dans le cas standard, le *fichier-d'entrée* de LISP est le clavier, et le *fichier-de-sortie* de LISP est l'écran. Ce n'est pas obligatoire. Eventuellement vous voulez écrire les résultats du calcul dans un fichier disque, pour le garder afin de l'utiliser ultérieurement, ou vous voulez les écrire sur une bande magnétique, une cassette ou même un ruban perforé (si ça existe encore !). De même, vos entrées peuvent venir non seulement du clavier, mais également d'un fichier disque, d'un lecteur de cartes, d'un lecteur de ruban, d'une bande magnétique ou d'une cassette. Dans ces cas, vous dites préalablement à LISP, que vous voulez changer de *fichier d'entrée* ou de *fichier de sortie* (Nous verrons dans la suite comment le faire.) Rappelons-nous, pour l'instant, que les entrées/sorties sont *dirigées* vers des *fichiers d'entrées/sorties* qu'on peut déterminer par programme, et qui sont par défaut (et jusqu'à nouvel ordre) le clavier, pour l'entrée de données, et l'écran de visualisation pour la sortie des résultats de calcul.

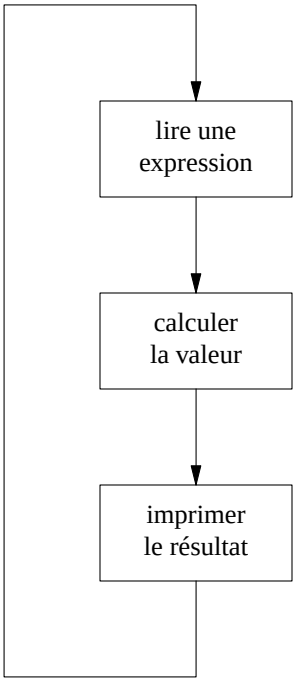
13.1. LA FONCTION GENERALE DE LECTURE : READ

Revenons pour l'instant à la fonction **READ**. Voici sa définition :

(READ) → *expression lue*
READ est une fonction sans argument qui lit une expression LISP (soit une liste, soit un atome) dans le flux d'entrée et qui ramène cette expression en valeur.

La fonction **READ** n'évalue donc pas l'expression qu'elle lit.

Cette fonction est primordiale dans l'interprète LISP même. Un interprète est un programme qui permet à la machine de comprendre un langage de programmation. Un interprète LISP est un programme qui permet à l'ordinateur de comprendre le langage LISP. Il se compose (au minimum) de trois parties : une partie qui lit ce que l'utilisateur (le programmeur) entre dans la machine, une partie qui calcule la valeur de ce qui a été lu, et une troisième partie qui imprime le résultat du calcul. Normalement, un interprète répète infiniment ces trois opérations :



Voici un petit interprète LISP, écrit en LISP :

```
(DE LISP ()
    (PRINT "-->" (EVAL (READ)))
    (LISP))
```

C'est un programme qui ne se termine jamais ! Voici une petite trace de l'utilisation de cette fonction :

? (LISP)	*le point d'interrogation est imprimé par LISP, indiquant que LISP attend une expression à évaluer. On lance la fonction* **LISP** *définie ci-dessus.*
? 1	*c'est toujours LISP qui imprime le point d'interrogation. Plus précisément : la fonction* **READ** *imprime un point d'interrogation chaque fois qu'elle attend une entrée du terminal. On donne l'expression numérique* **1** *à évaluer.*
--> 1	*notre fonction* **LISP** *a évalué l'expression et imprime sa valeur, précédée du signe '-->'*
? (CAR '(A B C))	**LISP** *s'appelle récursivement et redemande donc une expression à évaluer. On demande le* **CAR** *de la liste* **(A B C)**.
--> A	*et, encore,* **LISP** *nous donne le résultat de l'évaluation (l'atome* **A***), précédée du signe '-->'*
?	*et ainsi de suite. Visiblement, tout se passe exactement de la même manière que pendant une interaction avec l'interprète LISP ordinaire, excepté que le résultat est précédé du signe '-->' et non pas du signe '='.*

Pour donner un autre exemple d'utilisation de la fonction **READ**, reprenons le petit calculateur que nous avons décrit au chapitre précédent. La fonction principale s'appelait **CALCUL**. Nous allons construire un programme qui va lire une expression arithmétique après l'autre, l'envoyer à la fonction **CALCUL** pour calculer la valeur de cette expression, imprimer cette valeur, et redemander une nouvelle expression. Ainsi de suite jusqu'à ce qu'on donne l'atome **FIN** : on sort alors de cette fonction calculateur et on se remet à l'état normal. C'est encore un interprète, mais cette fois un interprète d'expressions arithmétiques :

```
(DE CALCULATEUR ()
   (PRINT "donnez une expression arithmétique s.v.p.")
   (LET ((X (READ)))
         (IF (EQ X 'FIN) 'fini
             (PRINT (CALCUL X))
             (PRINT "une autre expression arithmétique s.v.p.")
             (SELF (READ)))))
```

Ci-dessous un instantané de l'utilisation de cette fonction :

? (CALCULATEUR)	*on la lance*
donnez une expression arithmétique s.v.p.	*l'invitation*
? (1 + 2 + 3)	*la première expression arithmétique*
6	*le résultat*
une autre expression arithmétique s.v.p.	*une nouvelle invitation*
? (1 + 2 + (3 ∗ 4) + (80 / 10))	
23	
une autre expression arithmétique s.v.p.	
? (1 + 2 + 3 ∗ 4 + 80 / 10)	
39	
une autre expression arithmétique s.v.p.	
? FIN	*l'entrée indiquant que nous voulons terminer la session de calcul*
= fini	*on sort en ramenant l'atome* **fini** en valeur
?	*voilà, on est retourné vers LISP*

13.2. LES AUTRES FONCTIONS D'ENTREE / SORTIE

Souvent nous avons envie d'imprimer un message sans aller à la ligne. Dans l'exemple ci-dessus, on aimerait imprimer le message de manière à pouvoir entrer l'expression sur la même ligne que le message. A cette fin, LISP met à votre disposition la fonction **PRINC** en VLISP ou la fonction **PRINFLUSH** en LE_LISP. Elle est tout à fait identique à la fonction **PRINT** *sauf* qu'elle ne fait pas de passage à la ligne en fin d'impression. Ainsi, si nous changeons les deux occurrences de **PRINT** (dans les deux impressions de messages) en **PRINC**, l'interaction ci-dessus apparaiˆt sur votre terminal comme :

```
? (CALCULATEUR)
 donnez une expression arithmétique s.v.p.? (1 + 2 + 3)
 6
 une autre expression arithmétique s.v.p.? (1 + 2 + (3 ∗ 4) + (80 / 10))
 23
 une autre expression arithmétique s.v.p.? (1 + 2 + 3 ∗ 4 + 80 / 10)
 39
 une autre expression arithmétique s.v.p.? FIN
= fini
?
```

Revenons vers les fonctions de lecture. Les entrées se font - comme les sorties - en deux temps : d'abord le remplissage d'un tampon (le buffer d'entrée) à partir d'un périphérique d'entrée, ensuite l'analyse - par LISP - de ce tampon. L'analyse ne débute normalement qu'après lecture d'une ligne complète.

La fonction **READ** réalise donc deux opérations : si le buffer d'entrée est vide, elle lance le programme auxiliaire qui est chargé des opérations de lectures physiques (lire physiquement l'enregistrement suivant dans le flux d'entrée), ensuite elle lit une expression LISP à partir de ce buffer (l'expression LISP pouvant être un atome ou une liste). Si l'expression n'est pas complète à la fin de l'enregistrement (un enregistrement est une ligne de texte si l'entrée se fait à partir du terminal), **READ** lance une nouvelle lecture physique.

Imaginez que vous êtes en train d'exécuter un appel de la fonction **READ** et que vous tapez alors la ligne suivante :

(CAR '(A B C)) (CDR '(A B C)) ® [1]

Cette ligne sera mise - par le programme de lecture physique - dans le buffer d'entrée. Au début de l'analyse de la ligne lue, nous avons donc le tampon suivant :

(CAR '(A B C)) (CDR '(A B C)) ®
↑

avec un pointeur vers le début du buffer. Ce pointeur est représenté ici par la flèche en dessous du rectangle représentant le buffer.

Après l'exécution du **READ**, *seule* la première expression aura été lue. Le pointeur du buffer sera positionné à la fin de cette première expression, au début de la suite de caractères non encore analysée.

(CAR '(A B C)) (CDR '(A B C)) ®
↑

Il faudra un appel supplémentaire de la fonction **READ** pour *lire* également la deuxième expression. (Vous voyez la différence entre lecture physique et lecture LISP ?) Evidemment, si LISP rencontre pendant une lecture le caractère <RETURN> (représenté ici par le signe '® '), LISP relance cette fonction auxiliaire de lecture physique.

Ce qui est important, c'est de bien voir la différence entre lecture physique et l'analyse du texte lue. La fonction **READ** peut induire ces *deux* processus.

Si dans l'état

(CAR '(A B C)) (CDR '(A B C)) ®
↑

nous appelons la fonction **READCH**, LISP va lire *sans faire une analyse quelconque* le *caractère* suivant du buffer d'entrée. 'Lire' veux dire ici : ramener le caractère suivant en valeur et avancer le pointeur du buffer d'entrée d'une position. Après nous nous trouvons donc dans l'état suivant :

(CAR '(A B C)) (CDR '(A B C)) ®
↑

Le caractère lue est (dans ce cas-ci) le caractère <ESPACE>, normalement un caractère séparateur ignoré.

Voici la définition de la fonction **READCH** :

(READCH) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **READCH** lit vraiment n'importe quel caractère.

Cette fonction sert beaucoup si vous voulez lire des caractères non accessibles par la fonction **READ**, tels que les caractères '**(**', '**)**', '**;**', '**.**', <RETURN> ou <ESPACE>.

La fonction **PEEKCH** est exactement comme la fonction **READCH** mais elle ne fait pas avancer le pointeur du buffer d'entrée. Le caractère lu reste donc disponible pour le **READCH** ou **READ** suivant. Voici sa définition :

[1] le signe ® indique ici le caractère <RETURN>

(PEEKCH) → *caractère*
lit le caractère suivant dans le flot d'entrée et le ramène en valeur sous forme d'un atome mono-caractère. **PEEKCH** lit vraiment n'importe quel caractère mais le laisse disponible dans le flot d'entrée pour des lectures suivantes.

LISP met deux fonctions d'entrée/sortie immédiates à votre disposition : les fonctions **TYI** et **TYO**. '*Immédiat*' veux dire que ces deux fonctions n'utilisent pas les tampons d'entrée/sortie. Voici leurs définitions respectives :

(TYI) → *valeur numérique*
Cette fonction lit un caractère du *terminal* et retourne la valeur ascii du caractère *aussitôt* après la frappe de ce caractère. Nul besoin de terminer la ligne par un <RETURN> ! Le caractère tapé n'apparaiˆt pas sur l'écran, on dira que l'écho est inhibé.

(TYO *n*) → *n*
Cette fonction affiche sur l'écran, à la position courante du curseur, le caractère dont la valeur ascii est *n*.

Remarquons qu'en LE_LISP, la fonction **TYO** admet un nombre quelconque d'arguments et affiche donc la suite de caractères correspondantes.

Ces deux fonctions sont complémentaires, ainsi si vous voulez un écho (c'est-à-dire : si vous voulez imprimer le caractère frappé) pour un appel de la fonction **TYI**, il suffit d'écrire

(TYO (TYI))

TYO vous permet de sortir n'importe quel caractère sur votre terminal. Si vous voulez, par exemple, faire faire un petit 'beep' à votre terminal, il suffit d'appeler **(TYO 7)**, où 7 est le code ascii du caractère <BELL>.

Bien évidemment, ces quatre dernières fonctions sont les fonctions d'entrée/sortie les plus élémentaires de LISP. Avec elles vous pouvez écrire toutes les interfaces possibles.

Après toute cette théorie il nous faut absolument revenir vers un peu de pratique de programmation, afin de voir comment se servir de ces fonctions.

La fonction **TYI** est très souvent utilisée pour la programmation de jeux interactifs (à cause de son effet immédiat dès que vous avez tapé le caractère demandé, et donc à cause de la grande nervosité des programmes utilisant cette forme de lecture) ou pour éviter un défilement trop accéléré des choses que vous imprimez (puisque la machine *attend* la frappe d'un caractère, mais le caractère particulier que vous donnez n'a pas vraiment d'importance.

Pour illustrer ce dernier point, ci-dessous la fonction **COMPTE** qui imprime tous les nombres de **X** jusqu'à 0, l'un en-dessous de l'autre :

```
(DE COMPTE (X)
    (IF (LE X 0) "boum"
        (PRINT X)
        (COMPTE (1- X))))
```

Si vous appelez cette fonction avec **(COMPTE 4)**, elle vous imprime :

```
4
3
2
1
= boum
```

Le problème avec cette fonction est que, si vous l'appelez avec, par exemple, le nombre 1000, les nombres successifs seront imprimés à une telle vitesse que la lecture ne pourra plus suivre leur défilement sur l'écran. Pour prévenir cela, dans la deuxième version un petit compteur qui compte les lignes imprimées a été inseré. Après l'impression de 23 lignes,[2] le programme attend que vous tapiez un caractère quelconque avant de continuer. De cette manière vous pouvez interactivement déterminer la vitesse d'affichage :

```
(DE COMPTE (X)
   (LET ((X X)(Y 1))
        (IF (LE X 0) "boum"
            (PRINT X)
            (SELF (1- X) (IF (< Y 23) (1+ Y) (TYI) 1)))))
```

Observez donc la différence entre les deux versions de cette fonction.

Rappelons encore une fois qu'en LE_LISP l'appel de la fonction **LET** dans **COMPTE** doit être remplacé par **(LETN SELF . . . .**

Construisons maintenant un petit interprète LISP utilisant une fonction de lecture qui, en parallèle à la lecture, garde des statistiques sur l'utilisation des atomes : à chaque atome sera associé un compteur d'occurrences qui sera mis à jour au cours de la lecture.

Tout d'abord, construisons notre nouvelle fonction de lecture : **STAT-READ**.

```
(DE STAT-READ ()
  (LET ((X (PEEKCH))) (COND
        ((EQUAL X "(") (READCH) (READ-LIST))
        ((EQUAL X ")") (READCH) 'PARENTHESE-FERMANTE)
        ((MEMBER X '(" " "\\n" "\\t")) (READCH) (STAT-READ))
        (T (READ-ATOM)))))
```

Cette fonction débute, avec l'appel de la fonction **PEEKCH**, par un regard tentatif vers le buffer d'entrée : si le caractère prêt à être lu est une parenthèse ouvrante, nous appelons la fonction **READ-LIST**, une fonction chargée de lire des listes. Si le caractère en tête du buffer est une parenthèse fermante, nous enlevons ce caractère du buffer (avec l'appel de la fonction **READCH**) et sortons de **STAT-READ** en ramenant l'indicateur **PARENTHESE-FERMANTE**. Dans le cas où nous trouvons un séparateur standard en tête du buffer, nous l'éliminons et continuons à examiner les caractères restants. Si aucun de ces cas ne se présente, alors nous sommes suˆrs que nous avons affaire à un atome, et ce sera alors la fonction **READ-ATOM** qui se chargera de la lecture de l'atome.

Notons les chaiˆnes spéciales :

"\n" et **"\t"**

qui sont des écritures commodes pour le caractère <RETURN> et le caractère <TAB> à l'intérieur des chaiˆnes de caractères. Ces caractères s'écrivent en LE_LISP **#\CR** et **#\TAB** respéctivement, mais ne sont pas interprétés dans des chaines de caractères.

Regardez bien l'utilité de la fonction **PEEKCH** : c'est elle qui permet de faire toute cette analyse en cas

[2] D'ailleurs, le nombre 23 est non seulement le nombre de lignes de la majorité des terminaux actuels moins un, mais c'est également le premier nombre n, tel que la probabilité pour que deux personnes parmi n possèdent la même date anniversaire soit supérieure à ½.

distincts, *sans* toucher réellement au caractère analysé. Celui-ci reste disponible pour des lectures ultérieures. On aura bien besoin du caractère analysé dans la suite de la lecture d'un atome : il sera le premier caractère de son P-name.

La fonction **READ-LIST** doit lire un élément après l'autre de la liste à lire, jusqu'à la rencontre d'une parenthèse fermante, et construire réellement cette liste. La voici :

```
(DE READ-LIST ()
    (LET ((X (STAT-READ)))
        (IF (EQ X 'PARENTHESE-FERMANTE) ()
            (CONS X (SELF (STAT-READ))))))
```

A l'appel de la fonction **READ-ATOM** nous savons qu'en tête du buffer nous trouvons un atome : un appel de la fonction **READ** standard ne lira donc *que* cet atome !

```
(DE READ-ATOM ()
    (LET ((X (READ)))
        (IF (NUMBERP X) X
            (IF (GET X 'USE)
                (PUT X 'USE (1+ (GET X 'USE)))
                (PUT X 'USE 1)) X)))
```

Maintenant nous avons une fonction qui lit des expressions LISP quelconques et qui compte le nombre d'occurrences des atomes.

Voici finalement la fonction **LISP**, le moteur de notre micro-interprète :

```
(DE LISP ()
    (PRINC "une expression: ")
    (LET ((X (STAT-READ)))
        (IF (NEQ X 'PARENTHESE-FERMANTE) (PRINT (EVAL X)))
        (PRINC "une expression: ")
        (SELF (STAT-READ))))
```

Ci-dessous une petite fonction qui imprime nos statistiques. Cette fonction utilise la fonction standard **OBLIST**, qui livre une liste de tous les atomes connus du système LISP. On appelle cette liste de tous les atomes : l'*obliste*.[3] La fonction **STAT** prend donc un élément après l'autre de l'obliste, regarde si l'indicateur **USE** se trouve sur sa P-liste et, si oui, imprime le nom de l'atome suivi du nombre d'utilisations.

```
(DE STAT ()
   (LET ((Y (OBLIST)))
        (IF (NULL Y) ()
            (IF (GET (CAR Y) 'USE)
                (PRINT (CAR Y) (GET (CAR Y) 'USE)))
        (SELF (CDR Y)))))
```

Testons donc cette nouvelle fonction. Voici le début d'une interaction possible :

? (LISP)	*on lance la fonction interprète*
une expression: ? 1	*évaluons le nombre 1*
1	*la valeur de l'évaluation*
une expression: ? (QUOTE a)	*une deuxième expression*
a	*sa valeur*
une expression: ? 'a	*une troisième expression*
USE	*sa valeur ??!!??*

Que se passe-t-il ?

[3] L'obliste peut être considérée comme un sorte de micro-manuel LISP, accessible en ligne.

C’est très compliqué ! Regardons : le tout est évidemment duˆ au quote-caractère. Rappelez-vous ce que nous disions de ce caractère au chapitre 2 : nous insistions sur le fait que le caractère "**’**" n’est qu’une abréviation commode pour un appel de la fonction **QUOTE**, et si vous donnez à la machine LISP l’expression

’a

pour elle tout se passe comme si vous aviez donné l’expression

(QUOTE a)

C’est effectivement cette deuxième expression qui aura été enregistrée par le lecteur LISP. Cette transformation se joue directement au niveau de la lecture. Des caractères qui modifient le comportement du lecteur LISP s’appellent des *macro-caractères*. Nous les verrons plus en détail au paragraphe suivant.

Regardons quand même ce qui se passait ici :

1. Tout d’abord, la fonction **LISP** appelle la fonction **STAT-READ**.
2. La fonction **STAT-READ** teste les caractères se trouvant en tête du buffer d’entrée (ici nous trouvons la suite de caractères "**’a**") pour des occurrences de caractères spéciaux tels que ’**)**’, ’**(**’, ’<ESPACE>’, ’<TAB>’ ou ’<RETURN>’. Aucun de ces caractères n’étant présent, **STAT-READ** appelle donc la fonction **READ-ATOM**.
3. La fonction **READ-ATOM** débute par un appel, dans la ligne

   (LET ((X (READ)))

   de la fonction **READ** standard. La valeur de la variable **X** sera, d’après tout ce que nous savons maintenant, la liste

   (QUOTE a)

   C’est cette liste qui sera le premier argument de la fonction **PUT**. Horreur ! Nous étions suˆrs que la valeur de **X** serait toujours un atome; ici c’est une liste ! D’ailleurs, la fonction **PUT** attend un argument de type atome. Que fait donc la fonction **PUT** ? Soit qu’elle produit une erreur, soit qu’elle modifie, comme en VLISP, la P-liste de l’atome argument. Rappelons-nous que la P-liste est (en VLISP) le **CDR** de l’atome. La fonction **PUT** insère donc dans le **CDR** de son argument le couple <indicateur, valeur>. Ici, ça veut dire que l’appel :

   (PUT ’(QUOTE a) ’USE 1)

   qui ne correspond pas à la définiton de **PUT**, aura, en VLISP, comme résultat la nouvelle liste :

   (QUOTE USE 1 a)

   C’est cette liste qui sera le résultat de l’appel de la fonction **READ-ATOM**.
4. Finalement, de retour dans la fonction **LISP** nous aurons encore à calculer la valeur de l’expression
   (QUOTE USE 1 a)

   et, étant donné que la valeur d’un appel de la fonction **QUOTE** est son premier argument *tel quel*, le résultat final sera donc l’atome **USE**. On est loin de ce qu’on voulait calculer, c.à.d.: **a**.

La correction de cette erreur consistera à insérer dans la fonction **STAT-READ** une ligne spéciale traitant le macro-caractère "**’**"; ce qui nous donne la version suivante :

```
(DE STAT-READ ()
  (LET ((X (PEEKCH))) (COND
        ((EQUAL X "(") (READCH) (READ-LIST))
        ((EQUAL X ")") (READCH) 'PARENTHESE-FERMANTE)
        ((EQUAL X "'") (READCH) (LIST 'QUOTE (STAT-READ)))
        ((MEMBER X '(" " "\\n" "\\t")) (READCH) (STAT-READ))
        (T (READ-ATOM)))))
```

Ci-dessous une interaction avec cette nouvelle version de notre micro-interprète LISP :

```
? (LISP)                                  on relance la machine
une expression: ? 1                       une première expression à évaluer
1                                         sa valeur
une expression: ? '2                      une deuxième expression
2                                         sa valeur
une expression: ? 'a                      encore une expression quotée
a                                         ça semble marcher
une expression: ? (CAR '(a b c))
a
une expression: ? (CONS '(a b) '(c d))
((a b) c d)
une expression: ? (STAT)
CAR 1                                     regardons si ça compte bien le nombre d'occurrences
CONS 1                                    de chacun des atomes
a 3
b 2
c 2
d 1
STAT 1
nil
une expression: ? (CAR '(a b c))          continuons encore un peu
a
une expression: ? (CAR (CAR '((a b c))))
a
une expression: ? (STAT)                  et encore une statistique
CAR 4
CONS 1
d 1
a 5
b 4
c 4
STAT 2
nil
. . .
```

Bien évidemment le vrai lecteur de LISP est considérablement plus compliqué (nous en construirons un dans un chapitre ultérieur), mais ce modèle donne une bonne première approximation. Examinez bien le rôle des divers caractères séparateurs, ainsi que l'activité du quote-caractère.

13.3. LES DIVERS TYPES DE CARACTERES

13.3.1. détermination du type d'un caractère

Visiblement les divers caractères du jeu ascii n'ont pas tous le même effet pendant la lecture. La plupart sont considérée comme des caractères constituants des noms des atomes. Par contre, les parenthèses, l'espace <ESPACE>, la tabulation <TAB>, le return <RETURN>, le point-virgule, le double-quote et le quote-caractère ont des rôles particuliers.

Résumons ce que nous savons jusqu'à maintenant du rôle de ces différents caractères :

<ESPACE> <TAB> <RETURN>	Ces caractères ont le rôle de séparateur. Normalement, ils ne peuvent pas faire partie des noms des atomes; ils servent à séparer des suites d'atomes.
;	Le point-virgule est soit le caractère *début de commentaire*, soit le caractère *fin de commentaire*. Le caractère <RETURN> est également un caractère *fin de commentaire*. Si vous voulez mettre des commentaires à l'intérieur de votre programme, ils doivent donc débuter par un point-virgule. Les commentaires se terminent soit par un deuxième point-virgule ou par un <RETURN> en VLISP, soit uniquement par un <RETURN> en LE_LISP. Le lecteur LISP standard ignore tout commentaire.
"	Le doublequote délimite des chaîˆnes de caractères. A l'intérieur d'une chaîˆne de caractères aucun contrôle quant au rôle des caractères n'est effectué. Par contre, comme dans le langage C, il existe en VLISP quelques conventions pour y mettre des caractères bizarres. Ainsi, pour mettre une fin de ligne (un <RETURN>) à l'intérieur d'une chaîˆne, vous pouvez écrire ´\\n´, pour y mettre une tabulation vous pouvez utilisez ´\\t´ et pour un 'backspace´ utilisez ´\\b´.
(,)	Bien évidemment, les caractères parenthèse ouvrante et parenthèse fermante débutent et finissent l'écriture de listes. Ces deux caractères sont également des séparateurs et ne peuvent donc pas, normalement, être utilisés à l'intérieur des noms des atomes.
'	Le quote-caractère indique au lecteur LISP de générer un appel de la fonction **QUOTE** avec comme argument l'élément suivant directement ce caractère. Des caractères qui modifient le comportement du lecteur LISP s'appellent des *macro-caractères*.
---	---
	Tous les autres caractères (sauf le point (**.**), ainsi que le crochet ouvrant ([) et le crochet fermant (]), et en LE_LISP le caractère **#** et le caractère **|**, qui seront examinés par la suite) n'ont aucun rôle particulier et peuvent donc être utilisés pour les noms des atomes.

Si, à un instant quelconque de votre interaction avec LISP, vous ne vous rappelez plus du rôle d'un caractère, vous pouvez appeler la fonction **TYPCH** (cette fonction s'appelle **TYPECH** en LE_LISP) qui vous donne le TYPe du CHaractère. Voici sa définition :

(TYPCH *e***)** → *type*
l'argument *e* peut être une chaîne de caractères (d'un unique caractère) ou un atome mono-caractère. *type* est le type courant de ce caractère.

(TYPCH *e type***)** → *type*
l'argument *type* doit avoir une valeur d'un type de caractère. *type* sera le nouveau type du caractère *e*.

Le type *type* d'un caractère est définie comme suit :

type **VLISP**	*type* **LE_LISP**	*caractère*
0	CNULL	caractères ignorés
1	CBCOM	début de commentaire
2	CECOM	fin de commentaire
rien	CQUOTE	caractère QUOTE en LE_LISP
4	CLPAR	parenthèse ouvrante
5	CRPAR	parenthèse fermante
6	*rien*	crochet ouvrant en VLISP
7	*rien*	crochet fermant en VLISP
8	CDOT	le point **.**
9	CSEP	séparateur
10	CMACRO	macro-caractère
11	CSTRING	délimiteur de chaînes
12	CPNAME	caractère normal
rien	CSYMB	délimiteur symbole spécial |
rien	CPKGC	délimiteur de package
rien	CSPLICE	splice-macro
rien	CMSYMB	symbole mono-caractère

Voilà, muni de ces connaissances et de cette nouvelle fonction, vous pouvez écrire des choses étranges. Par exemple, si les parenthèses ne vous plaisent plus, après les instructions

(TYPCH "{" (TYPCH "("))
(TYPCH "}" (TYPCH ")"))

Vous pouvez aussi bien écrire

(CAR '(DO RE MI))

que

{CAR '{DO RE MI}}

13.3.2. les macros-caractères

Les macro-caractères modifient donc le comportement du lecteur LISP de manière à générer des appels de fonctions. Ainsi le quote-caractère, l'unique macro-caractère que nous connaissons actuellement, traduit

'quelque-chose

en

(QUOTE quelque-chose**)**

L'utilisateur LISP peut définir ses macro-caractères personnels grâce à la fonction de Définition de Macro-Caractères : **DMC**. Cette fonction est syntaxiquement définie comme suit :

(DMC *c* **(***variable*$_1$ *variable*$_2$ . . . *variable*$_n$**)** *corps-de-fonction***)**

Le premier argument, *c*, peut être une chaîne ou un atome mono-caractère. Lorsque le lecteur LISP rencontrera ce caractère, il lancera l'évaluation du *corps-de-fonction*, les variables de la liste de variables, $variable_1$ à $variable_n$, serviront de variables locales si besoin est. Ensuite, le résultat de l'évaluation sera placé à la place du dit caractère dans le flot d'entrée.

Par exemple, si le quote-caractère n'était pas défini, nous pourrions le définir comme :

(DMC "'" () (LIST (QUOTE QUOTE) (READ)))

En LE_LISP, il est préférable de quoter le nouveau macro-caractère par le caractère spécial '|'. La définition du macro-caractère quote s'écrit donc, en LE_LISP :

(DMC |'| () (LIST (QUOTE QUOTE) (READ)))

Cette définition peut se lire comme un ordre au lecteur LISP disant qu'à chaque rencontre d'un caractère **'** dans le flot d'entrée il faut remplacer ce caractère par une liste dont le premier élément est l'atome **QUOTE** et le deuxième élément sera l'expression LISP qui suit directement ce caractère.

Prenons un autre exemple : construisons des macro-caractères qui nous permettent d'écrire

[$argument_1$ $argument_2$ . . . $argument_n$]

au lieu de

(LIST $argument_1$ $argument_2$ . . . $argument_n$**)**

Pour cela nous devons définir deux macro-caractères : le macro-caractères '[' qui doit construire l'appel de la fonction **LIST** avec autant d'arguments qu'il y a d'expressions LISP jusqu'au caractère ']' suivant. Ce dernier caractère doit devenir également un macro-caractère, soit seulement pour le rendre séparateur. Voici les deux définitions :

```
(DMC "[" ()
   (LET ((X 'LIST))
      (AND (NEQUAL X "]") (CONS X (SELF (READ))))))

(DMC "]" () "]")
```

La fonction **NEQUAL** est bien évidemment une abréviation pour **(NULL (EQUAL**,[4] de manière telle que nous avons la relation suivante :

(NEQUAL *x y***)** ≡ **(NULL (EQUAL** *x y***))**

Comme vous voyez, les définitions de macro-caractères peuvent être de complexité arbitraire. Leur rôle est, dans tous les cas, de simplifier l'écriture de programmes.

Regardons donc quelques exemples d'utilisation de ces macro-caractères :

[1 2 3]	→	**(1 2 3)**
['a 'b (CAR '(1 2 3)) 'c]	→	**(a b 1 c)**
'[1 2 [3 4] 4]	→	**(LIST 1 2 (LIST 3 4) 4)**
(CONS 1 [2 3 4])	→	**(1 2 3 4)**

Maintenant nous pouvons utiliser le macro-caractère quote (**'**) pour entrer des listes et les macro-caractères

[4] Si cette fonction n'existe pas dans votre LISP, vous pouvez la définir aisément comme :

(DE NEQUAL (X Y) (NULL (EQUAL X Y)))

'[' et ']' pour générer des appels de la fonction **LIST**.

Construisons donc un dernier jeu de macro-caractères et définissons le macro-caractère *back-quote* (**‘**). Ce macro-caractère, d'une grande utilité, existe dans la plupart des systèmes LISP. Il est une sorte de combinaison des caractères **'**, [et]. On l'utilise quand on a une longue liste quotée à l'intérieur de laquelle on veut mettre un élément à évaluer, ou, autrement dit : on l'utilise quand on construit une liste ou la majorité des éléments sont quotés.

Prenons un exemple. Supposons que la variable **X** est liée à la liste **(D E)** et qu'on veuille construire une liste avec, dans l'ordre, les éléments **A**, **B**, **C**, la valeur de **X** suivi de l'élément **F**. L'unique manière disponible actuellement est d'écrire :

(LIST 'A 'B 'C X 'F)

ou

['A 'B 'C X 'F]

Si nous voulons construire à partir de ces éléments la liste :

(A B C D E F)

nous devons écrire :

(CONS 'A (CONS 'B (CONS 'C (APPEND X (LIST 'F)))))

Finalement, si nous voulons, toujours à partir de ces quelques éléments disponibles, construire la liste :

(A B C E F)

nous devons écrire :

(CONS 'A (CONS 'B (CONS 'C (CONS (CADR X) (LIST 'F)))))

Toutes ces écritures semblent bien lourdes, sachant qu'un seul élément dans la liste est calculé, tous les autres sont donnés tels quels.

Le macro-caractère *back-quote* nous livre un dispositif typographique permettant d'écrire ces formes beaucoup plus aisément. Voici, comment construire les trois listes données en exemple, en utilisant ce nouveau macro-caractère **‘** :

‘(A B C ,X F)	→	**(A B C (D E) F)**
‘(A B C ,@X F)	→	**(A B C D E F)**
‘(A B C ,(CADR X) F)	→	**(A B C E F)**

Visiblement, le macro-caractère backquote utilise deux macro-caractères supplémentaires : le virgule (**,**) et l'escargot (**@**). A l'intérieur d'une liste *back-quotée* une expression précédée d'une virgule donne sa valeur qui est ensuite insérée comme *élément* dans le résultat final. Une expression précédée des caractères **,** et **@** donne sa valeur qui est ensuite insérée comme *segment* dans le résultat final. Toutes les autres expressions sont prises telles quelles. Les résultats d'un appel de la fonction **QUOTE** et d'un appel de la fonction **BACKQUOTE** sont donc identiques s'il n'y a pas de virgules ou d'escargots à l'intérieur de l'expression back-quotée.

Il ne reste qu'à écrire la définition de ce macro-caractère, dont voici la définition (très compliquée) :

```
(DMC "‘" ()
   (LET ((VIRGULE (TYPCH ",")))
      (TYPCH "," (TYPCH "’"))
      (LET ((QUASI [’EVAL [’BACKQUOTE [’QUOTE (READ)]]]))
         (TYPCH "," VIRGULE)
         QUASI)))
```

Ce que nous avons fait ici correspond à deux choses : d’abord nous avons limité la portée du macro-caractère **,** (virgule) : par la sauvegarde de son type de caractère à l’entrée de la définition et la détermination de son type à la valeur du type du macro-caractère quote, nous nous assurons qu’à l’intérieur de la portée du macro-caractère **‘** (backquote) le macro-caractère **,** (virgule) soit défini, à la sortie du backquote nous restaurons, par l’instruction

(TYPCH "," VIRGULE)

le type de ce caractère à l’extérieur du backquote. Ensuite, nous déléguons toute l’activité du backquote à une fonction auxiliaire nommée **BACKQUOTE**. Cette fonction devra construire les appels des fonctions **LIST** et **CONS** nécessaires pour construire la bonne liste résultat. Evidemment, pour réellement construire cette liste, nous devons évaluer les appels générés; ce que nous faisons par l’appel explicite de la fonction **EVAL**.

Regardons donc la fonction **BACKQUOTE** :

```
(DE BACKQUOTE (EXPR) (COND
   ((NULL EXPR) NIL)
   ((ATOM EXPR) [QUOTE EXPR])
   ((EQ (CAR EXPR) ’*UNQUOTE*) (CADR EXPR))
   ((AND (CONSP (CAR EXPR))(EQ (CAAR EXPR) ’*SPLICE-UNQUOTE*))
      [’APPEND (CADAR EXPR) (BACKQUOTE (CDR EXPR))])
   ((COMBINE-EXPRS (BACKQUOTE (CAR EXPR))
         (BACKQUOTE (CDR EXPR)) EXPR))))
```

BACKQUOTE reçoit comme argument l’expression qui suit directement le caractère **‘** (backquote). Cette expression est liée à la variable **EXPR** afin de procéder à une analyse de cas. Les cinq cas suivants peuvent se présenter :

1. L’expression **EXPR** est vide : alors il n’y a rien à faire;
2. L’expression **EXPR** est un atome : alors **BACKQUOTE** génère un appel de la fonction **QUOTE**;

   exemple : si la valeur de **EXPR** est *atome*, **BACKQUOTE** génère :

   (QUOTE *atome***)**
3. L’expression débute par l’atome spécial ***UNQUOTE*** : cet atome a été généré par le caractère **,** (virgule); il faut alors ramener la valeur de l’argument de **,**. Ceci sera fait en remplaçant l’expression **(*UNQUOTE*** *arg***)** dans l’expression lue, par **arg** dans l’expression générée.

   exemple : si la valeur de **EXPR** est **(*UNQUOTE*** *argument***)**, **BACKQUOTE** génère :

   argument
4. Le premier élément de **EXPR** est une liste débutant par l’atome spécial ***SPLICE-UNQUOTE*** : cet atome a été généré par les caractères **,@**; il faut alors concaténer la valeur de l’argument de ***SPLICE-UNQUOTE*** à la suite de la liste. Ceci est fait en générant un appel de la fonction **APPEND** qui a comme premier argument l’argument de ***SPLICE-UNQUOTE*** et comme deuxième argument le résultat de l’appel récursif de **BACKQUOTE** avec le **CDR** de l’expression **EXPR**.

   exemple : si la valeur de **EXPR** est **((*SPLICE-UNQUOTE*** *argument***)** *reste-des-arguments***)**,

BACKQUOTE génère :

(APPEND *argument* **(BACKQUOTE** *reste-des-arguments*)**)**

5. Dans tous les autres cas, il faut combiner, de manière adéquate les résultats de l'analyse du **CAR** et du **CDR** de **EXPR**. Nous y distinguons quatre cas :
   a. le **CAR** et le **CDR** sont des constantes : dans ce cas il suffit de générer un appel de la fonction **QUOTE** avec l'expression entière.
   b. le **CDR** est **NIL**, alors il faut générer un appel de la fonction **LIST** avec le **CAR** comme argument.
   c. le **CDR** est une liste qui commence par un appel de **LIST** : on sait alors que cet appel a été généré par **BACKQUOTE** et on peut donc insérer le **CAR** comme nouveau premier argument de cet appel.
   d. Dans tous les autres cas, on génère un appel de la fonction **CONS** avec le **CAR** comme premier et le **CDR** comme deuxième argument, ceci restaure donc bien la liste originale.

Voici la fonction **COMBINE-EXPRS** :

```
(DE COMBINE-EXPRS (LFT RGT EXPR) (COND
    ((AND (ISCONST LFT) (ISCONST RGT)) [QUOTE EXPR])
    ((NULL RGT) ['LIST LFT])
    ((AND (CONSP RGT) (EQ (CAR RGT) 'LIST))
        (CONS 'LIST (CONS LFT (CDR RGT))))
    (['CONS LFT RGT])))
```

qui utilise la fonction auxiliaire **ISCONST** :

```
(DE ISCONST (X)
    (OR (NULL X) (EQ X T)(NUMBERP X)(AND (CONSP X)(EQ (CAR X) 'QUOTE))))
```

Finalement, il ne reste qu'à définir le macro-caractère **,**. Ce macro-caractère ne devant être valide qu'à l'intérieur du **BACKQUOTE**, il faut bien faire attention que la définition même du macro-caractère ne modifie pas *définitivement* son type. C'est pourquoi nous mettons la **DMC** à l'intérieur d'un **LET** qui se contente de sauvegarder et restaurer le type du caractère **,** avant et après sa définition en tant que macro-caractère. Voici donc ce bout de programme :

```
(LET ((VIRGULE (TYPCH ",")))
    (DMC "," ()
        (LET ((X (PEEKCH)))
            (IF (NULL (EQUAL X "@")) ['*UNQUOTE* (READ)]
                (READCH)
                ['*SPLICE-UNQUOTE* (READ)])))
    (TYPCH "," VIRGULE))
```

Regardez bien l'utilisation de la fonction **PEEKCH** !

Après tout ce qui précède, examinons donc quelques exemples de ce nouveau macro-caractère. Voici un premier exemple : si vous donnez à LISP l'expression :

'(A B C)

puisque tout les éléments de cette liste sont des constantes, la fonction **BACKQUOTE** génère l'appel :

(QUOTE (A B C))

et son évaluation donne donc la liste :

(A B C)

Reprenons les exemples ci-dessus et supposons que la variable **X** soit liée à la liste **(D E)** et la variable **Y** à l'atome **F**.

Si nous livrons à LISP l'expression :

‘(A B C ,X Y)

le lecteur LISP traduira ceci en :

(EVAL (BACKQUOTE (QUOTE (A B C (*UNQUOTE* X) Y))))

ce qui sera, après évaluation de l'appel de **BACKQUOTE** :

```
(EVAL ’(CONS (QUOTE A)
             (CONS (QUOTE B)
                   (CONS (QUOTE C)
                         (CONS X (QUOTE (Y))))))
```

dont l'évaluation nous livre finalement :

(A B C (D E) Y)

Pour notre deuxième exemple, l'expression :

‘(A B C ,@X ,Y)

sera traduite par le lecteur LISP en :

```
(EVAL (BACKQUOTE
       (QUOTE (A B C
                 (*SPLICE-UNQUOTE* X)
                 (*UNQUOTE* Y)))))
```

ce qui se transformera, après l'évaluation de l'argument de **EVAL** en :

```
(EVAL ’(CONS (QUOTE A)
             (CONS (QUOTE B)
                   (CONS (QUOTE C)
                         (APPEND X (LIST Y))))))
```

qui produira la liste :

(A B C D E F)

Arrêtons ici avec les macro-caractères d'entrée (nous y reviendrons) et regardons brièvement comment changer de fichier d'entrée/sortie.

13.4. COMMENT CHANGER DE FICHIERS D'ENTREE/SORTIE

Jusqu'à maintenant nous ne pouvons entrer des fonctions LISP qu'à travers le terminal, et nous ne pouvons écrire les résultats d'un calcul que vers ce même terminal. On désire souvent préparer ses programmes sous éditeur pour seulement ensuite les charger en LISP, ou sauvegarder ses résultats dans des fichiers afin de pouvoir les réutiliser par la suite. Ce petit paragraphe vous montrera comment réaliser ce genre d'opérations.[5]

[5] Les entrées/sorties sur fichiers sont une des choses qui changent le plus d'un système LISP à l'autre. Dans ce court paragraphe nous ne donnons que quelques fonctions significatives d'*un* système LISP particulier. Avant de tester ces fonctions sur votre version personnelle de LISP, consultez donc d'abord votre documentation

Tout d'abord, LISP met à votre disposition une fonction de *lecture* de fichiers. Cette fonction s'appelle **LIB** en VLISP et **LOAD** en LE_LISP. Elle est fort utile pour *charger* un programme qui se trouve dans un fichier. Voici la définition de **LIB** :

(LIB *nom-de-fichier***)** → *nom-de-fichier*
et lit le fichier de nom *nom-de-fichier* en entier.

Ainsi, si vous travaillez sous un système UNIX, et si le programme du paragraphe précédent (la définition du *backquote*) se trouve dans le fichier **backquote.vlisp**, la commande

(LIB backquote.lisp) ou **(LOAD "backquote.lisp")**

va lire ce fichier, définition par définition.

L'argument *nom-de-fichier* peut être une chaiˆne de caractères ou un atome.

Si vous travaillez sous VLISP, et si le nom de votre fichier se termine par *.vlisp*, vous pouvez omettre cette partie. L'écriture :

(LIB backquote)

est donc tout à fait suffisante pour lire le fichier **backquote.vlisp**. En LE_LISP, l'extension *.ll* sert le même rôle : pour lire le fichier **backquote.ll**, il suffit d'écrire :

(LOAD "backquote")

Notez toutefois que si vous avez réellement un fichier nommé 'backquote´, ce sera ce fichier qui sera lu, et non le fichier 'backquote.vlisp' ou 'backquote.ll' !

En VLISP, si vous voulez lire un fichier, dont vous n'êtes pas suˆr qu'il existe, le prédicat **PROBEF** répond à la question : "est-ce que tel ou tel fichier existe ?". ce prédicat est défini comme :

(PROBEF *nom-de-fichier***)** → *nom-de-fichier* si le fichier de ce nom existe
→ **NIL** si aucun fichier de nom *nom-de-fichier* n'existe

Donc : si, dans un programme, vous voulez lire un fichier particulier et que vous êtes très angoissé quant à son existence, la suite d'instruction :

(IF (PROBEF *nom***) (LIB** *nom***)** *message-d'erreur***)**

est la plus adéquate.

De même que pour la fonction **LIB**, l'argument *nom-de-fichier* de **PROBEF** peut être un atome ordinaire ou une chaîne de caractères.

Sous VLISP-UNIX il existe une généralisation de la fonction **LIB**, c'est la fonction **INCLUDE**.

Elle a la même syntaxe et la même sémantique (le même sens) que **LIB**, mais, cette instruction peut se trouver à l'intérieur d'un fichier que vous êtes en train de charger. Ainsi vous pouvez *pendant* le chargement d'un fichier lancer la commande de chargement d'un autre fichier.

Cette fonction **INCLUDE** n'est pas nécessaire en LE_LISP : la fonction **LOAD** permet pendant le chargement d'un fichier le rencontre d'autres appels de **LOAD**.

---

technique !

Afin de rendre ceci plus explicite, imaginez que vous ayez un fichier nommé **foo.vlisp** qui contienne - entre autre - un appel de **INCLUDE** du fichier **bar.vlisp** :

fichier **foo.vlisp** :

premières définitions
. . .
(INCLUDE bar)
reste des définitions
. . .

Quand vous lancez la commande :

(INCLUDE foo)

LISP va commencer à lire les *premières définitions* du fichier **foo.vlisp** jusqu'à la rencontre de l'instruction :

(INCLUDE bar)

Là, LISP va interrompre la lecture du fichier **foo.vlisp**, lire tout le fichier **bar.vlisp** (et éventuellement d'autres fichiers, si **bar.vlisp** contient des appels de **INCLUDE**), pour ensuite continuer à lire le *reste des définitions* du fichier **foo.vlisp**.

Vous pouvez ainsi *inclure* jusqu'à 10 niveaux de fichiers. C'est très utile de temps à autre.

Voilà, maintenant vous pouvez garder vos programmes d'une session à l'autre !

Remarquez que les fonctions **LIB** et **INCLUDE**

1. lisent tout le fichier et
2. n'évaluent pas leur argument, nul besoin de quoter le nom de fichier.

Vous ne pouvez donc pas utiliser ces deux fonctions pour lire par programme une expression après l'autre. Pour faire ceci, c.à.d.: pour lire des données d'un fichier une à une, vous devez utiliser la fonction **INPUT**. Cette fonction ne fait rien d'autre qu'indiquer à la machine *où* doivent lire les appels des fonctions **READ**, **READCH** ou **PEEKCH** ultérieurs.

Voici la définition de cette fonction :

(INPUT *atome***)** → *atome*
indique à la machine LISP que les lectures suivantes se font à partir du fichier de nom *atome*. Si *atome* est égal à **NIL**, les lectures suivantes se font à partir du terminal.

C'est comme si LISP ne pouvait se connecter, à un instant donné, qu'à *un seul* fichier d'entrée particulier et que la fonction **INPUT** réalise les différentes connections nécessaires. Bien entendu, comme sous système UNIX, le terminal n'*est* rien d'autre qu'un fichier particulier.

Prenons un exemple, et supposons qu'on ait un fichier, nommé **valeurs.vlisp**, contenant la suite d'atomes que voici :

1
2
3
4
5

6
7
8
9
10
FIN

et construisons une fonction qui lit sur un fichier un nombre après l'autre, jusqu'à rencontrer l'atome **FIN**, et calcule leur somme. La voici :

```
(DE SOMME-DE-NOMBRES (FICHIER)
    (INPUT FICHIER)
    (SOMME-AUX 0))

(DE SOMME-AUX (X)
    (IF (NEQ X 'FIN) (+ X (SOMME-AUX (READ)))
        (INPUT ())  ; pour revenir vers le clavier ;
        0))
```

Pour calculer la somme des nombres contenus dans le fichier **valeurs.vlisp**, il suffit d'appeler cette fonction comme :

```
(SOMME-DE-NOMBRES "valeurs.vlisp")
```

Finalement, pour terminer ce chapitre, regardons comment écrire les résultats d'un calcul vers un fichier avec la fonction **OUTPUT**. Cette fonction est l'inverse de la fonction **INPUT**. Voici sa définition :

(OUTPUT *atome***)** → *atome*
indique à la machine LISP que les impressions suivantes se font vers le fichier de nom *atome*. Si *atome* est égal à **NIL**, les impressions suivantes se font vers le terminal.

La fonction **OUTPUT** crée le fichier de nom *atome* s'il n'existe pas encore. Par contre, s'il existe déjà, les impressions suivantes seront ajoutées à la fin du fichier.

Pour prendre un exemple, voici la fonction **CREE-VALEURS** qui génère les valeurs nécessaires pour notre fonction **SOMME-DE-NOMBRES** ci-dessus :

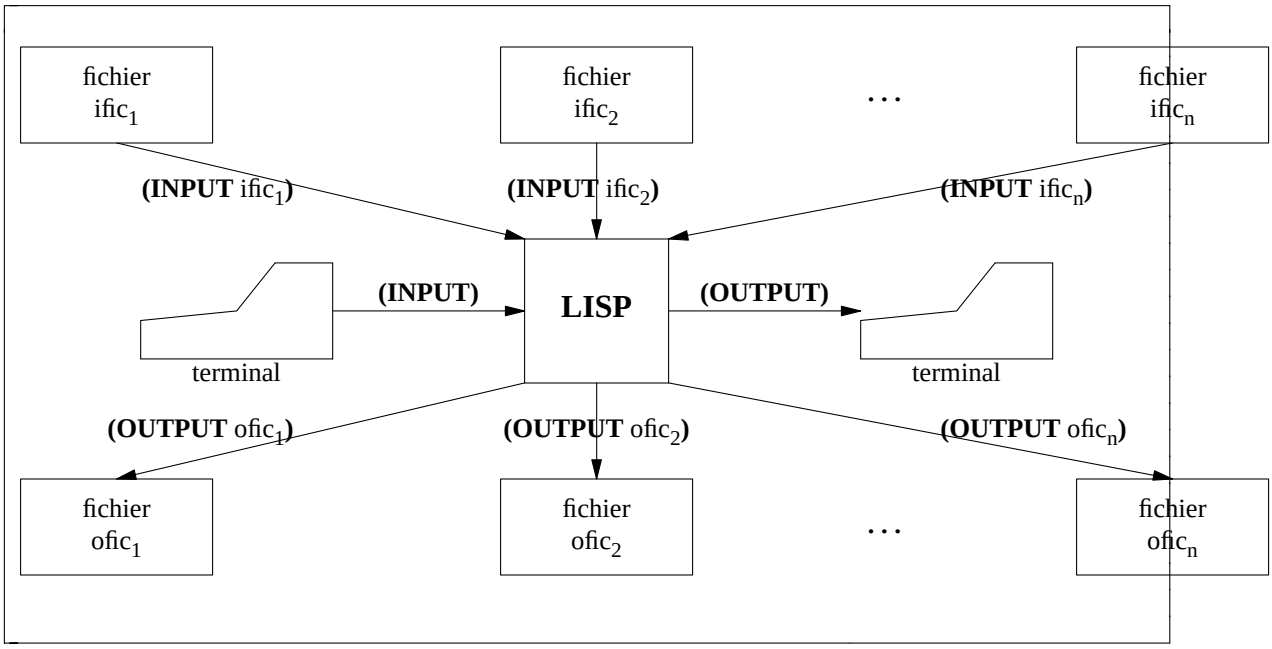
```
(DE CREE-VALEURS (N FICHIER)
    (OUTPUT FICHIER)
    (CREE-AUX 1))

(DE CREE-AUX (X) (COND
    ((> X N) (PRINT 'FIN)
             (OUTPUT ()))  ; pour revenir vers le terminal ;
    (T (PRINT X) (CREE-AUX (1+ N)))))
```

Pour créer le fichier **valeurs.vlisp** ci-dessus, il suffit d'activer cette fonction par l'appel :

```
(CREE-VALEURS 10 "valeurs.vlisp")
```

Voici graphiquement, les possibilités d'échanges entre LISP et le monde extérieur :



13.5. EXERCICES

1. Au lieu d'ecrire **(LIB** *nom-de-fichier***)** chaque fois que vous voulez charger un fichier, écrivez un macro-caractère '^' qui vous permette de charger des fichiers simplement par la commande

   ^*nom-de-fichier*

2. Redéfinissez les macro-caractères >= et <= pour **GE** et **LE** respectivement.
3.
   a. Réalisez un programme qui écrive dans un fichier nommé **pair.nb** les nombres pairs de 1 à *n* et l'atome **FIN**.
   b. Réalisez également un programme qui écrive dans un fichier nommé **impair.nb** les nombres impairs de 0 à *m* et l'atome **FIN**.
   c. Ecrivez un troisième programme qui écrive dans un fichier nommé **somme.nb** les sommes des doublets de nombres pair et impair des fichiers **pair.nb** et **impair.nb**.

chapitre prcédent

solutions

chapitre suivant