Manuel de référence ORICLISP

Table des matières

I. Une introduction au langage LISP

LISP est un langage créé en 1964 par John MacCarthy pour traiter des informations organisées en arbres binaires (formules mathématiques, phrases de la langue naturelle, connaissances...). Ces arbres sont représentés par des listes en LISP (qui est une abréviation de LISt-Processing).

La capacité de LISP à traiter des informations symboliques en a fait un des "langages machine" de domaines de l'informatique tels que l'Intelligence Artificielle, la Robotique, le Calcul Formel, la Théorie de la Programmation (les preuves de programme par exemple), la théorie des jeux, les éditeurs syntaxiques (type Emacs) et plus généralement les environnements de programmation, etc.

Pour le situer par rapport aux autres langages, on a l'habitude de considérer trois familles de langages (certains langages sont bien sûr à l'intersection de ces familles): les langages impératifs comme Fortran, Pascal, Cobol, Basic, Ada, C... où une suite d'ordres permet de converger vers la solution; les langages déclaratifs comme Prolog où l'on se contente de décrire le problème (c'est une logique déductive qui permet d'aller vers la solution); les langages fonctionnels et applicatifs comme LISP, ML, FP où l'on applique des fonctions au sens mathématique du terme (il n'y a plus de "variables" dans le programme). On peut considérer que les premiers langages à objets comme SmallTalk descendent de LISP, et on trouve aussi son influence dans le langage LOGO.

Hormis cette classification, les principales caractéristiques des langages fonctionnels de la famille LISP sont les suivantes (il n'y a en effet pas un mais des dialectes LISP: InterLisp, MacLisp, VLisp, LeLisp, MuLisp, FranzLisp...):

- ils procèdent par composition de fonctions récursives alors que les langages classiques (impératifs) procèdent par répétition de séquences d'affectation.
- ils manipulent des arborescences et un programme est un arbre de même nature qu'un arbre de données (un programme peut donc être traité par un autre programme !)
- ils sont structurés et modulaires (par composition de fonctions)
- ils sont interactifs (le dialogue avec l'interprète permet de construire les programmes de manière incrémentale)
- la gestion de la mémoire est dynamique et automatique (grâce à un "ramasse-miettes" qui récupère les données qui ne sont plus utilisées)

I.1 Syntaxe

Un programme LISP est un arbre binaire, soit, comment donc le représenter avec une suite de caractères ASCII ? Bien sûr, il n'est pas question de faire un dessin, alors il a été choisi d'utiliser les parenthèses, l'espace et le point pour représenter ces arborescences. Dès que l'on aura remarqué que les deux branches d'un arbre binaire sont elles-mêmes des arbres, on acceptera sans peine la représentation suivante dite "paire pointée": (S1 . S2) où S1 et S2 sont aussi soit des paires pointées soit des feuilles de l'arbre, que l'on appelle des "atomes".

Les atomes, comme en physique, sont donc les éléments constitutifs et deux sortes sont distinguées : les nombres entiers (parfois aussi des réels) et les "symboles" qui sont des identificateurs et jouent le rôle de données abstraites. Par exemple, deux symboles sont utilisés pour représenter les booléens classiques : NIL pour "Faux" et T pour "Vrai" (bien que tout ce qui n'est pas NIL est aussi "Vrai"). De plus, NIL sert aussi pour dénoter le "Vide", la liste vide plus précisément...

Puisque nous évoquons le terme de "liste", précisons que les listes sont des arbres particuliers (les arbres sont appelés "S-expressions" en LISP) : il a été choisi de faire porter le premier élément d'une liste par la partie gauche de la paire pointée et la suite de la liste par la partie droite, ce qui fait qu'une liste de fruits s'écrirait, à l'aide de paires pointées :
(pomme . (poire . (abricot . nil)))

Les listes sont tellement courantes que la notation ci-dessus est peu pratique et l'on a bien sûr une première écriture abrégée :

(pomme poire abricot . nil)

C'est une simplification d'écriture qui donne une signification spéciale à la séparation par des espaces à l'intérieur de parenthèses. On abrège encore cette écriture en donnant la signification de fin de liste au symbole NIL avec :

(pomme poire abricot)

Voilà qui est bien plus simple ! Mais pour compliquer un peu, rappelons qu'une liste est une S-expression; rien n'empêche donc un ou des éléments d'une liste d'être eux-mêmes des listes... Par exemple,

(marcel (age 20) (enfants (lucie (age 2)) (alain (age 1))))

est une liste de 3 éléments ! Le premier est un atome, le second une liste de deux atomes, le troisième une liste d'un atome et de deux listes... on commence à comprendre que savoir compter les parenthèses va être important !

I.2 L'interprète

Comment ? On change de paragraphe sans donner la syntaxe des programmes LISP ? Eh oui, si vous savez construire des listes, vous savez programmer ! Un programme LISP, c'est une S-expression, disons plutôt un atome ou une liste, que l'interprète va "évaluer". Par exemple, si vous donnez un nombre à évaluer à l'interprète, il vous renverra ce même nombre (facile...). Si vous donnez un symbole à évaluer, l'interprète retournera la "valeur" de ce symbole s'il en a une, par exemple : MON_DOCTEUR pourrait s'évaluer en

(Dugenou Leon (23 chemin du cubitus))

Enfin, si vous donnez une liste à évaluer, l'interprète considèrera que le premier élément de la liste est une fonction et que les autres éléments sont les arguments donnés à la fonction. Par exemple :

(+ 2 2) renverra 4

(CAR MON_DOCTEUR) renverra Dugenou, on verra en effet plus loin que CAR est une fonction qui renvoie le premier élément d'une liste.

L'interprète LISP peut donc être décrit par l'application successive des trois fonctions suivantes :
- READ qui lit une S-expression (nous dirons par la suite une expression pour simplifier) au clavier ou depuis un fichier.
- EVAL qui évalue la valeur d'une expression.
- PRINT qui affiche une expression sur l'écran ou dans un fichier.

Selon le dialecte LISP utilisé, cet interprète peut être une fonction standard (TOPLEVEL ou DRIVER ou autre), dont la définition contient une composition des trois fonctions précédentes : (PRINT (EVAL (READ)))
On remarque que les arguments étant évalués avant d'être passés aux fonctions, c'est la fonction READ qui est invoquée d'abord, son résultat est passé à la fonction EVAL, et le résultat d'EVAL est envoyé à PRINT ! (on verra que la plupart des fonctions évaluent leurs arguments, c'est l'appel par valeur, seules quelques fonctions ne les évaluent pas, comme les structures de controles, on parle alors d'appel par nom).

C'est donc la fonction EVAL qui est au coeur du langage LISP: lorsque vous demandez (+ 2 2), c'est elle qui invoquera le code de la fonction addition.

I.3 Les fonctions prédéfinies

Tous les dialectes LISP partagent un même noyau minimal de fonctions qui est suffisant pour écrire n'importe quel programme. Les dialectes se différencient par des jeux de fonctions étendus, plus ou moins riches, qui permettent d'écrire les programmes de manière plus concise. Nous décrivons ici un noyau minimal:

a) QUOTE

QUOTE est une fonction qui permet de protéger son argument de l'évaluation. Elle n'évalue donc pas son argument et le renvoie tel quel en résultat.
Exemple:

```
(QUOTE (+ 2 2))    -->    (+ 2 2)
```

Son usage très courant fait qu'une abréviation a été définie :

```
'A     -->   A
'(CAR MON_DOCTEUR)    -->   (CAR MON_DOCTEUR)
```

Pour les curieux, le caractère ' est un "macro-caractère", c.a.d que c'est en fait une fonction qui s'évalue dès qu'elle est rencontrée dans le flot de caractères venant du clavier (l'action du macro-caractère ' est d'invoquer la fonction READ pour lire l'expression suivante, puis de construire une liste avec QUOTE en premier élément et cette expression en second élément, et enfin de renvoyer cette liste !)

b) Les sélecteurs CAR et CDR

Ces fonctions évaluent leur argument, CAR prend la partie gauche de la paire pointée donnée en argument et CDR la partie droite (les noms de ces fonctions viennent hélas de l'appellation A et D de registres de la première machine ayant implémenté LISP...)
Exemple:

```
(CAR '(+ 2 3))    -->  +
(CAR '((A B) C))  -->  (A B)
(CDR '(+ 2 3))    -->  (2 3)
(CDR '((A B) C))  -->  (C)
```

Suivant les dialectes LISP, appliquer la fonction CAR à un atome peut renvoyer une erreur ou la valeur de cet atome. De même, la fonction CDR appliquée à un atome peut retourner une erreur ou une liste de propriétés associées à l'atome (ces deuxièmes possibilités permettent d'appliquer CAR et CDR à n'importe quelle expression LISP, mais il vaut mieux réserver ces fonctions aux paires pointées (listes ou arbres) par souci de portabilité).

c) Les constructeurs

CONS est la fonction constructeur de base : elle construit une paire pointée à partir de deux expressions.
Exemple:

```
(CONS 'A 'B)       -->  (A.B)
(CONS 'A '(B C))   -->  (A B C)
(CONS '(A B) '(C)) -->  ((A B) C)
(CONS 'A NIL)      -->  (A)
(CONS NIL '(A))    -->  (NIL A)
```

LIST existe toujours pour simplifier la construction de listes de longueur quelconque. La fonction LIST accepte un nombre quelconque d'arguments.
Exemple:

```
(LIST 'a 'b 'c 'd)    -->  (a b c d)
(LIST 'a '(b c) 'd)   -->  (a (b c) d)
```

d) Les prédicats

Ce sont des fonctions renvoyant une valeur booléenne (NIL représente la valeur "Faux", tout autre résultat est "Vrai"), elles évaluent leurs arguments.

```
ATOM  renvoie T si son argument est un atome,    sinon NIL.
NUMBERP     renvoie T si son argument est un nombre,   sinon NIL.
CONSP renvoie T si son argument est une liste,   sinon NIL.
NULL  renvoie T si son argument est NIL,         sinon NIL.
EQUAL renvoie T si ses arguments sont égaux,     sinon NIL.
EQ    renvoie T si ses arguments sont les mêmes, sinon NIL.
```

Ces deux derniers méritent une petite explication: EQ teste si les deux objets donnés en argument ne font qu'un en mémoire (égalité physique) tandis que EQUAL compare toutes les branches des objets pour vérifier qu'ils ont même représenta-
-tion externe.
Exemple:

```
(ATOM 'A)       -->  T
(ATOM '(A))     -->  NIL
(ATOM '())      -->  T  en effet, () est la liste vide représentée par
                     l'atome NIL !
(NULL 'A)   -->  NIL
(NULL '(A)) -->  NIL
(NULL NIL)  -->  T
(EQUAL '(A (B C) D)  '(A (B C) D)) -->  T
(EQ    '(A (B C) D)  '(A (B C) D)) -->  NIL
(EQUAL 'A 'A)     -->  T
(EQ    'A 'A)     -->  T      car les atomes sont présents de manière unique
                     en mémoire...
```

e) La CONDitionnelle

COND accepte en paramètre un nombre quelconque de clauses (non évaluées) de la forme "(prédicat expression)". COND n'évalue pas ses arguments à priori, mais évalue séquentiellement les prédicats dans les clauses jusqu'à en trouver un "Vrai" (non NIL), dans ce cas il évalue l'expression associée dans la clause et la renvoie en résultat. Sinon (tous les prédicats ont été évalués à NIL), COND renvoie NIL.
Exemple:

```
(COND ((EQ 'A 'A) 'OUI))  -->  OUI
(COND ((EQ 'A 'B) 'OUI))  -->  NON
(COND ((EQ 'A 'B) 'OUI)
      (T        'NON))       -->  NON
           ce qui montre l'utilisation de T comme dernier prédicat,
           mais on peut s'en passer comme suit :
(COND ((EQ X 1) 'PREMIER)
      ((EQ X 2) 'DEUXIEME)
      ((EQ X 3) 'TROISIEME)
      (         'AUTRE))    --> PREMIER, DEUXIEME, TROISIEME ou AUTRE
                        suivant la valeur de X...
```

f) L'arithmétique

Les noms de fonction varient d'un dialecte à un autre, mais on trouvera toujours les 4 opérations et des prédicats d'ordre.
Exemple:

```
(- 239 (* 7 (/ 239 7)))    -->  1
(< 3 7)                    -->  T
```

g) L'affectation

Pour les puristes, c'est une horreur comme un GOTO dans un langage structuré. En pratique, on réserve souvent son usage au niveau le plus haut d'un programme pour associer un nom à une valeur, comme on peut le faire en mathéma-
-tique, mais on se garde de modifier cette association (les symboles ne sont pas des variables !) pour éviter les effets de bord.
Exemple:

```
(SETQ A '(1 2 3))       -->  (1 2 3)
A                 -->  (1 2 3)
```

SETQ n'évalue pas son premier argument (symbole), et renvoie son second argument après l'avoir associé au symbole. SET est encore plus dangereux car il évalue aussi le premier argument:

```
(SETQ B 'A) -->  A
B           -->  A
(SET B '(A B))    -->  (A B)
B           -->  A
A           -->  (A B)
```

h) La définition de fonctions

Une fonction LISP est une expression de la forme

(LAMBDA (param1 param2 ... paramN) expr)

où param1, param2 ... paramN sont les noms des paramètres formels et expr le corps de la fonction.

Exemple:

```
(LAMBDA (X) X)    est la fonction identité (elle renvoie son argument)
(LAMBDA (X Y)
  (COND     ((ATOM X) (CONS X Y))
      (T      (CONS (CAR X) Y))))
            est une fonction qui rajoute X en tête de la liste Y si
            X est un atome, sinon rajoute le CAR de X en tête de la
            liste Y.
```

La manière d'associer une définition de fonction à un nom n'est pas standard, LeLisp utilise la fonction DE (par exemple: "(DE IDENTITE(X) X)"), VLisp range la définition dans la liste de propriété d'un atome, d'autres associent la définition à la valeur d'un atome. Pour la suite, nous utiliserons la fonction MuLisp PUTD.

Exemple:

```
(PUTD 'IDENTITE '(LAMBDA(X) X))
```

En MuLisp, les atomes ont par défaut pour valeur eux-mêmes, ce qui permet de supprimer l'apostrophe (quote) devant le nom de fonction lorsque aucune valeur ne lui est associée. Allez, quelques classiques :

```
(PUTD LENGTH
  '(LAMBDA (X)
     (COND ((NULL X) 0)
         (T      (+ 1 (LENGTH (CDR X))))
)  ) )
      calcule la longueur d'une liste: (LENGTH '(A B C)) --> 3
(PUTD MEMBER
  '(LAMBDA (X L)
     (COND ((NULL L)          NIL)
         ((EQUAL X (CAR L)) T)
         (T             (MEMBER X (CDR L)))
)  ) )
      cherche si une expression est présente dans une liste :
            (MEMBER '(A B) '(A B (A (A B)) (B A) (A B))  --> T
      car on trouve (A B) en quatrieme position.
(PUTD FACT
  '(LAMBDA (N)
     (COND ((EQ N 0) 1)
         (T        (* N (FACT (- N 1))))
)  ) )
      calcule la factorielle d'un nombre...
```

II. Présentation d'OricLisp

OricLisp est un dialecte LISP inspiré de LeLisp (Jérôme Chailloux) pour les fonctions définies par l'utilisateur (LAMBDA, FLAMBDA, MLAMBDA, cf plus bas) et de MuLisp (Albert D.Rich, David R.Stoutemyer, Roy Feldman) pour tout le reste même si l'implémentation est très différente (MuLisp a été implémenté sur les processeurs Intel 8080 et 8086).
Un nombre relativement faible de fonctions sont prédéfinies (plus de 60 tout de même !) afin de laisser le maximum d'espace disponible à l'utilisateur, mais ses caractéristiques puissantes permettent de développer des applications très conséquentes.

OricLisp a été écrit en 1986 (mais ce manuel ne l'a suivi qu'en 1995 !).

II.1 Principales caractéristiques:

* sauvegarde d'images mémoire permettant de reprendre une session au point exact où elle a été sauvée, ou d'enrichir le langage par des définitions personnalisées, ou de produire des applications Lisp démarrant automatiquement.
* arithmétique entière sur 32 bits, exprimée dans une base arbitraire (de la base 2 à la base 36)
* symboles de taille arbitraire (jusqu'à 256 caractères) et pouvant comporter n'importe quel caractère ascii (même l'espace). Les symboles peuvent ainsi jouer le rôle de chaînes de caractères.
* pile virtuelle segmentée permettant au processeur d'utiliser une pile de près de 3 Ko avec la rapidité de la pile habituelle de 256 octets.
* espace utilisateur important avec en standard 16 Ko pour les paires pointées et les chaînes de caractères (soit plus de 4000 paires pointées), et près de 12 Ko pour les atomes. Cette répartition peut être changée.
* ramasse-miettes transparent pour l'utilisateur, compactant tous les espaces utilisateurs (chaînes, paires, nombres, symboles).
* fonctions évaluant ou n'évaluant pas leurs arguments, plus macro-fonctions et macro-caractères !
* espace clos de pointeurs et données typées par leur adresse pour une plus grande efficacité.

II.2 Premier contact :

Taper CLOAD"ORICLISP" depuis le Basic 1.1 de l'Oric pour charger et exécuter le système (ou LOAD"ORICLISP" pour un système à disquettes, mais attention, les images mémoires ne sont sauvées que sur cassette). La ligne supérieure de l'Oric affiche le copyright, et le prompt "point d'interrogation" signale que le système attend des caractères tapés au clavier. Entrer des expressions montre que l'interprète est actif:
Exemple:

```
? 'HELLO
=HELLO
? (+ 2 2)
=4
```

a) L'écran et le clavier

Pour plus de rapidité, les routines d'écrans ont été redéfinies, elles permettent de travailler avec 38 ou 40 colonnes et affichent sur la totalité des 28 lignes en mode rouleau avec une routine de défilement rapide.

La scrutation du clavier s'adapte au travail de l'utilisateur: lorsqu'il est en phase d'entrée de caractères, elle est effectuée tous les 3/100e de seconde, alors qu'elle n'est effectuée qu'une fois par seconde lorsque le système calcule, et une fois tous les 1/10e de seconde lorsque le système affiche. Ceci permet d'allier performance et confort d'utilisation, un programme peut toujours être stoppé en cours de calcul par un Ctrl-C (appuyé 1 seconde) et gagne ainsi environ 15% en vitesse d'exécution.

b) L'édition de lignes

Elle est très différente de celle du Basic et gère un buffer de 128 caractères (plus de 3 lignes écran). La dernière entrée (terminée par la touche RETURN) est conservée dans le buffer pour permettre une correction aisée de celle-ci.

Deux modes sont gérés: insertion et suppression, par défaut le mode suppression est actif et Ctrl-I permet de basculer du mode suppression au mode insertion et vice-versa.

En mode suppression, tout caractère ASCII tapé vient remplacer le caractère qui occupe la même position dans le buffer (donc remplace un caractère de l'entrée précédente).

En mode insertion, tout caractère ASCII tapé vient s'insérer à la position courante du buffer, décalant ainsi vers la droite le reste du buffer.

Le dernier caractère ASCII (127) a une signification spéciale puisqu'il s'agit de DEL qui détruit le dernier caractère tapé (et revient donc une position plus à gauche dans le buffer). Deux caractères de contrôle sont définis en plus de la bascule insertion/suppression pour utiliser l'entrée précedente conservée dans le buffer :

- Ctrl-A récupère en effet le caractère courant de l'entrée précédente, laisser un Ctrl-A appuyé permet de recopier une entrée jusqu'au dernier caractère désiré.
- Ctrl-D détruit le prochain caractère du buffer. Le résultat de cette action est malheureusement invisible pour l'utilisateur (tant qu'il n'entre pas un Ctrl-A).

Exemple: supposons que l'utilisateur ait tapé la ligne suivante:

```
? (PUTD FACT '(LAMBDA(N) COND ((EQ N 0) 1)) (T (* N (FACT (- N 1))))))
=(LAMBDA(N) COND ((EQ N 0) 1))
```

Il s'aperçoit alors sans peine qu'il a oublié une parenthèse ouvrante avant le COND et tapé une parenthèse de trop après la première clause. Pour corriger son entrée, il suffit qu'il appuie sur Ctrl-A jusqu'à ce que

```
? (PUTD FACT '(LAMBDA(N)
```

soit affiché, puis passer en mode insertion avec Ctrl-I, taper une parenthèse ouvrante, de nouveau enfoncer Ctrl-A jusqu'à ce que

```
? (PUTD FACT '(LAMBDA(N) (COND ((EQ N 0) 1)
```

soit affiché, puis détruire la parenthèse fermante superflue de l'entrée précédente par Ctrl-D, et terminer en laissant Ctrl-A enfoncé...

II.3 Les primitives OricLisp

a) QUOTE

la fonction QUOTE classique, protège son argument de l'évaluation.

```
? (QUOTE (+ 2 2))
=(+ 2 2)
```

Remarque : le macro-caractère ' a le même effet

```
? '(+ 2 2)
=(+ 2 2)
```

b) Les sélecteurs

* les sélecteurs CAR et CDR sont présents, ainsi que les compositions de 2 ou 3 CAR ou CDR (par exemple, "(CADR X)" est l'équivalent de "(CAR (CDR X))"). OricLisp possède un espace clos de pointeurs, on peut toujours appliquer un CAR ou un CDR sans erreur; dans le cas d'un atome, CAR renvoie la valeur associée à l'atome et CDR la liste de propriétés associée à l'atome. La valeur d'un symbole est par défaut lui-même (tant qu'aucune valeur ne lui a été liée), celle d'un nombre est toujours le même nombre. La liste de propriétés d'un symbole est vide par défaut (elle vaut NIL), celle d'un nombre reflète son signe : T pour un nombre positif ou nul, NIL pour un nombre négatif.

Exemple:

```
? (CAR '(A.B))
=A
? (CADR '(A B C))
=B
? (CDDR '(A B C D))
=(C D)
```

* LAST renvoie la dernière paire pointée de premier niveau de son argument (et non le dernier élément) ou NIL si l'argument est un atome. Renvoyer la dernière paire pointée plutôt que le dernier élément permet de modifier cette paire pointée (pour ajouter en fin de liste avec RPLACD par exemple, mais attention aux effets de ces modifications physiques), et le dernier élément peut être facilement obtenu avec un CAR supplémentaire.

Exemple:

```
? (LAST '(A B C D))
=(D)
? (LAST '(A B.C))
=(B.C)
? (CAR (LAST '(A B C D)))
=D
```

* ASSOC cherche le premier argument (la clef) dans la A-liste fournie par le deuxième argument. Une A-liste (liste associative) est une liste de la forme

```
((clef1.val1) (clef2.val2) ... (clefN.valN))
```

ASSOC compare (avec EQUAL) le premier argument avec chacune des clefs (les éléments atomiques de la A-liste sont sautés) et renvoie l'élément complet de la première comparaison avec succès, sinon NIL. Certains Lisp ne renvoient que la partie valeur de la paire pointée, il suffit de rajouter un CDR pour obtenir le même résultat (tandis que renvoyer l'ensemble permet de modifier la valeur associée à la clef).

Exemple:

```
      ? (ASSOC 'MARTIN '((DUPONT JEAN 61586273) (MARTIN JACQUES 61483922)
                         (DUPONT ALAIN 61289019)))
      =(MARTIN JACQUES 61483922)
```

* MEMBER cherche (avec EQUAL) le premier argument dans la liste fournie par le second argument et renvoie la fin de liste commençant à l'élément trouvé, ou NIL s'il n'est pas trouvé. Certains Lisp ne renvoient qu'une valeur booléenne T ou NIL, ici le résultat peut servir (pour détruire l'élément de la liste, par exemple).

Exemple:

```
      ? (MEMBER '(MARTIN JACQUES 61483922)
              '((DUPONT JEAN 61586273) (MARTIN JACQUES 61483922)
                         (DUPONT ALAIN 61289019)))
      =((MARTIN JACQUES 61483922) (DUPONT ALAIN 61289019)))
```

c) Les constructeurs
---

* CONS construit une paire pointée avec les deux arguments fournis.

Exemple:

```
      ? (CONS 'A 'B)
      =(A.B)
      ? (CONS 'A '(B C))
      =(A B C)
```

* LIST construit une liste avec tous les arguments fournis.

```
      ? (LIST 'a '(b c) 'd)
      =(a (b c) d)
```

* OBLIST construit une liste de tous les symboles

```
      ? (OBLIST)
      =(LAST OBLIST DIV MOD ..... LAMBDA T NIL)
```

* APPEND construit une liste qui est la concaténation des deux listes passées en argument. De nouvelles paires pointées sont crées pour le début de cette liste, contrairement à NCONC. Remarque : APPEND avec un seul argument peut être utilisé pour copier une liste.

```
      ? (APPEND '(A (B C) D) '(E F (G H)))
      =(A (B C) D E F (G H))
```

* REVERSE construit une nouvelle liste qui est une liste renversée de son premier argument (l'implémentation utilise un deuxième paramètre pour cumuler le résultat partiel. De ce fait, si un deuxième argument est fourni, il sera concaténé à la liste renversée)

```
      ? (REVERSE '(A (B C) D))
      =(D (B C) A)
      ? (REVERSE '(A (B C) D) '(E F))
      =(D (B C) A E F)
```

* GC n'est pas un constructeur comme CONS, LIST, OBLIST, APPEND ou REVERSE. Mais il intervient lui aussi dans la gestion de l'espace mémoire. Tous les construc--teurs font appel à CONS pour allouer une paire pointée. Lorsque plus aucune paire ne peut être allouée, le ramasse-miettes (Garbage Collector) est invoqué automatiquement. Il peut aussi être appelé explicitement avec la fonction GC.

d) Les prédicats
---

* EQUAL renvoie T si ses deux arguments sont égaux. Toutes les branches sont comparées.

```
      ? (EQUAL '(A (B C) D) '(A (B C) D))
      =T
```

* EQ renvoie T si ses deux arguments ne sont qu'un même objet en mémoire. Il ne peut exister deux symboles de même nom. Il peut exister plusieurs nombres de même valeur mais EQ considère que ce sont les mêmes. Lorsque l'on partage des paires pointées, EQ est un test intéressant, parce qu'il est très rapide (il ne regarde pas les branches)

```
      ? (EQ '(A (B C) D) '(A (B C) D))
      =NIL
```

* ATOM renvoie T si son argument est un atome (donc pas une paire pointée)

```
      ? (ATOM 'A)
      =T
      ? (ATOM '(A B))
      =NIL
      ? (ATOM '())
      =T
```

* NULL renvoie T si son argument est NIL (la liste vide)

```
      ? (NULL 'A)
      =NIL
      ? (NULL '(A B))
      =NIL
      ? (NULL '())
      =T
```

* PLUSP renvoie T si son argument est un nombre positif ou nul.

```
      ? (PLUSP 0)
      =T
      ? (PLUSP 'A)
      =NIL
```

* MINUSP renvoie T si son argument est un nombre négatif

```
      ? (MINUSP -3)
      =T
      ? (MINUSP '(A B))
      =NIL
```

* ZEROP renvoie T si son argument est le nombre 0

```
      ? (ZEROP NIL)
      =NIL
      ? (ZEROP 0)
      =T
```

Remarque : n'importe quelle expression peut servir de prédicat puisque toute valeur non NIL est considérée comme "Vrai". Par exemple, T est le prédicat toujours vrai, utile pour la dernière clause d'une conditionnelle.

e) La conditionnelle et les structures de contrôle
---

* COND est la conditionnelle classique de la forme suivante :

```
      (COND (pred1 exp11 exp12 ... exp1N)
            (pred2 exp21 exp22 ... exp2M)
            ....
      )
```

Chaque prédicat pred1, pred2 ... est évalué séquentiellement jusqu'à trouver une valeur non NIL pour un prédicat predI, alors les expressions associées expI1, expI2... sont évaluées en séquence, la dernière donnant la valeur de retour du COND (si aucune expression n'est associée au prédicat, c'est la valeur du prédicat qui est renvoyée). Avoir plusieurs expressions au lieu d'une à la suite du prédicat n'est utile que si les expressions ont des effets de bord.
Exemple:

```
      ? (COND ((< A B) A) (B)) renvoie le minimum des nombres A et B
```

* PROGN évalue séquentiellement tous ses arguments et renvoie la valeur du dernier. PROGN n'a d'intérêt que si les expressions font des effets de bord et peut être généralement omis du fait du PROGN implicite dans les définitions de fonction et dans la conditionnelle COND.
Exemple:

```
      ? (PROGN (PRIN '(FACT 5)) (FACT 5))
      (FACT 5)=120
```

* PROG1 évalue séquentiellement tous ses arguments et renvoie la valeur du premier. Son usage est de même lié aux effets de bord.
Exemple:

```
      ? (SETQ A (PROG1 B (SETQ B A))) échange les valeurs associées à A et B
```

* AND est à la fois le "et" logique classique et une structure de contrôle (équivalente à des "si...alors..." imbriqués). Les arguments sont évalués en séquence jusqu'à ce qu'une valeur NIL soit trouvée, auquel cas le résultat est NIL et les arguments suivants sont "court-circuités". Si tous les arguments s'évaluent à une valeur non NIL, la valeur du dernier est renvoyée, le résultat est donc "Vrai".
Exemple:

```
      ? (AND (ZEROP 1) (PRINT 'ARGH))
      =NIL  et on remarque que le deuxième argument n'est pas évalué
      ? (AND A B C) est équivalent à (COND (A (COND (B (COND (C))))))
```

* OR est à la fois le "ou" logique classique et une structure de contrôle (une suite de "si...alors...sinon si"). Les arguments sont évalués en séquence jusqu'à ce qu'une valeur non NIL soit trouvée (cette valeur est alors retournée et l'évaluation des arguments restants est court-circuitée). Si tous les arguments sont NIL, NIL est renvoyé.
Exemple:

```
      ? (OR NIL '() 'A (PRINT 'ARGH))
      =A
      ? (OR A B C) est équivalent à (COND (A) (B) (C))
```

* NOT est le "non" logique (ce n'est pas une structure de contrôle mais il permet d'inverser la signification de prédicats) et renvoie T si son argument est NIL, sinon NIL. NOT est donc identique au prédicat NULL.
Exemple:

```
      ? (NOT NIL)
      =T
```

* WHILE est une structure de contrôle de style impératif de la forme
      (WHILE pred exp1 exp2 ... expN)
Tant que pred s'évalue à "Vrai" (non NIL), WHILE itère sur l'évaluation de exp1, exp2 ... expN (il faut donc nécessairement un effet de bord pour que la valeur de pred devienne NIL). WHILE est implémenté sans récursivité et permet donc des boucles importantes (ou infinies) sans saturation de la pile.
Exemple:

```
      ? (WHILE T (PRINT (EVAL (READ)))
            une boucle infinie pour un nouvel interprète !
      ? (WHILE (NOT (ZEROP N)) (SETQ N (- N 1)))
      =NIL
            décrémente N jusqu'à 0
```

f) L'arithmétique
---

* RADIX change la base pour la représentation des nombres entrés au clavier et affichés à l'écran. La nouvelle base est donnée en argument et doit être comprise entre 2 et 36 (au delà de la base 10, jusqu'à 26 lettres peuvent être employées). RADIX renvoie la nouvelle base en résultat (mais on peut remarquer que l'affichage d'une base est toujours "10"). Si un argument non numérique est fourni, la base n'est pas changée et est renvoyée en résultat.
Exemple:

```
      ? (RADIX 8)
      =10
      ? (+ 6 7)
      =15
      ? (RADIX 12)
      =10
      ? (+ 6 7)
      =13
```

* les 4 opérations entières sont +,-,*,/. Elles prennent deux nombres en argument et renvoient le résultat. Une erreur NONNUMERIC est levée si un opérande n'est pas un nombre et l'erreur DIVBYZERO est levée en cas de division par 0. / renvoie le quotient de la division entière tandis que MOD renvoie le reste. Souvent, quotient et reste sont tous deux utiles, DIV permet de ne faire qu'une fois la division et renvoie un paire pointée (quotient.reste)
Exemple:

```
      ? (DIV 10 3)
      =(3.1)
```

* les prédicats d'ordre sont < et >. Ils renvoient T si l'ordre est vérifié entre les deux arguments et NIL sinon. Les comparaisons au sens large (égalité incluse) ne sont pas des primitives, il faut inverser le test.
Exemple:

```
      ? (NOT (< A B))
            renvoie T si A>=B, et NIL sinon.
```

g) Les modifications physiques et les fonctions à effet de bord

* SETQ lie la valeur fournie par le second argument au symbole spécifié par le premier (le premier argument n'est pas évalué). Si le symbole était local à une fonction, la liaison est perdue à la sortie de la fonction.
Exemple:

```
? (SETQ A '(1 2))
=(1 2)
? A
=(1 2)
? (SETQ A 'B)
=B
? A
=B
```

* SET lie la valeur fournie par le second argument au symbole spécifié par le premier (les deux arguments sont évalués).
Exemple:

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

* RPLACA et RPLACD remplacent respectivement le CAR et le CDR de la paire pointée donnée en premier argument par le deuxième argument, et renvoient la paire modifiée.
Exemple:

```
? (SETQ A '(1 2))
=(1 2)
? (RPLACA A 0)
=(0 2)
? A
(0 2)
? (RPLACD A '(3 4))
=(0 3 4)
? A
(0 3 4)
```

* NCONC est la concaténation de liste, comme APPEND mais sans consommation de paire pointée. NCONC modifie la fin de la liste donnée par son premier argument pour lui faire suivre la deuxième liste.
Exemple:

```
? (SETQ A '(1 2))
=(1 2)
? (NCONC A '(3 4 5))
=(1 2 3 4 5)
? A
=(1 2 3 4 5)
```

* MEMORY permet de lire ou d'écrire un octet en mémoire. Si un seul argument est donné, c'est une adresse dont MEMORY renverra le contenu. Si deux arguments numériques sont donnés, MEMORY change l'adresse mémoire avec l'octet donné en second argument, et renvoie l'ancienne valeur.
Exemple:

```
? (RADIX 16)
=10
? (MEMORY 26B 1)
=7    change la couleur de papier...
```

* TIME renvoie la valeur d'une horloge en 1/100e de secondes et peut donc servir à mesurer le temps d'exécution d'un programme.
Exemple:

```
? (SETQ T1 (TIME)) (GC) (- (TIME) T1)
=5
```

* PRINT affiche l'expression donnée en paramètre, et retourne cette expression en résultat. PRIN fait la même chose mais sans terminer par un retour charriot.
Exemple:

```
? (PRINT 'HELLO)
HELLO
=HELLO
? (PRINT '(A (B C) D))
(A (B C) D)
=(A (B C) D)
? (PRIN 0)
0=0
```

* READ lit une expression au clavier et la renvoie en résultat. Si plus d'une expression est donnée, les autres restent dans le buffer clavier à la disposi-
-tion d'un READ ultérieur.
Exemple:

```
? (PRIN '(+ 2 2)) (+ 2 2)
(+ 2 2)=(+ 2 2)
=4
? (SETQ A (READ)) (+ 2 2)
=(+ 2 2)
      On remarque que (+ 2 2) n'a pas été lu par l'interprète mais
      bien par le READ.
```

* SAVE produit sur cassette une image mémoire du nom de l'argument donné en paramètre. Il n'y a pas de fonction LOAD, une image mémoire contient le noyau OricLisp et démarre automatiquement au point exact du SAVE en tapant la commande CLOAD du Basic de l'Oric.
Exemple:

```
? (PROGN (SAVE 'IMAGE1) 'HELLO)
=HELLO     et un fichier IMAGE1 est écrit sur cassette que l'on lancera
      depuis le Basic :
CLOAD"IMAGE1"
=HELLO
?
```

h) La définition de fonction
---

* Une fonction OricLisp évaluant ses arguments est une expression de la forme
      (LAMBDA (param1 ... paramN) exp1 exp2 ... expM)
où param1 ... paramN sont les noms des paramètres formels et exp1, exp2 ... expM le corps de la fonction (l'évaluation d'une LAMBDA comporte donc un PROGN implicite).
Exemple:
```
      (LAMBDA () (/ (TIME) 100))
            une fonction sans paramètres qui renvoie l'horloge en secondes
      (LAMBDA (N) (+ N 1))
            une fonction qui renvoie le nombre successeur de son argument
```

Attention ! le deuxième élément de la liste de la fonction (les paramètres formels) doit impérativement être une liste (éventuellement vide).

Les LAMBDA sont des fonctions "sans nom" qui peuvent être utilisées de la même façon que les primitives du langage.
Exemple:
```
      ? ( (LAMBDA(N)(+ N 1)) 3)
      =4
```
Pour associer un nom à une LAMBDA, on utilise la primitive PUTD.
```
      ? (PUTD 'INCR '(LAMBDA(N)(+ N 1)))
      =(LAMBDA (N) (+ N 1))
      ? (INCR 3)
      =4
```
Remarque : un symbole peut simultanément avoir une valeur, une liste de propriétés et une définition de fonction associées.
Exemple:
```
      ? (SETQ INCR '(1 2))
      =(1 2)
      ? (INCR 3)
      =4
      ? INCR
      =(1 2)
```

* GETD permet de lire la définition de fonction d'un symbole. GETD renvoie la LAMBDA d'une fonction définie en LISP, T pour une fonction en langage machine et NIL si la fonction n'est pas définie.
Exemple:
```
      ? (GETD 'T)
      =NIL
      ? (GETD 'EVAL)
      =T
      ? (GETD 'INCR)
      =(LAMBDA (N) (+ N 1))
```

* MOVD permet de copier la définition de fonction d'un symbole dans un autre symbole, définissant un synonyme pour une fonction sans consommer d'espace mémoire supplémentaire (en dehors du symbole).
Exemple:
```
      ? (MOVD 'APPEND 'COPY)
      =T
      ? (COPY '(A B (C D) E))
      =(A B (C D) E)
```

II.4 Concepts avancés

a) Fonctions sans évaluation

OricLisp permet de définir des fonctions n'évaluant pas leurs arguments, une telle fonction est de la forme

```
(FLAMBDA param exp1 exp2 ... expM)
```

Un seul symbole est fourni en paramètre formel, il sera lié à la liste de tous les paramètres non-évalués.
Les fonctions n'évaluant pas leurs arguments permettent de définir de nouvelles structures de contrôle (elles sont parfois aussi utilisées pour définir des fonctions dont le nombre de paramètres est indéterminé)
Exemple:

```
(FLAMBDA L
  (COND ((EVAL (CAR L)) (EVAL (CADR L)))
      (T              (EVAL (CADDR L)))
) )                     définit un IF X THEN Y ELSE Z !!
```

que l'on liera bien sûr de la façon suivante:

```
? (PUTD 'IF '(FLAMBDA L
?              (COND ((EVAL (CAR L)) (EVAL (CADR L)))
?                    (T              (EVAL (CADDR L))) )))
=(FLAMBDA L (COND ((EVAL (CAR L))(EVAL (CADR L))) (T (EVAL (CADDR L)))))
? (IF NIL (PRINT 'ARGH) (PRINT 'OK))
OK
=OK   et on constate qu'effectivement, le "alors" n'est pas évalué,
      contrairement à ce qui serait arrivé si on avait défini le IF
      par (LAMBDA (X Y Z) (COND (X Y) (Z)))
```

Exemple:

```
(PUTD 'PLUS '(FLAMBDA L
  (COND ((NULL L) 0)
      ((+ (EVAL (CAR L)) (PLUS (CDR L))))
)))
      définit une opération PLUS qui somme tous ses arguments
? (PLUS 1 2 3 4 5 6 7)
=28
```

b) Macro-caractères

Les macro-caractères sont des caractères qui ont une fonction associée appelée lorsqu'ils sont rencontrés durant une lecture clavier. Le caractère apostrophe (quote) est le seul macro-caractère défini par défaut. READ consulte une table (à l'adresse $8000) pour savoir si un caractère est un macro-caractère. Pour définir un macro-caractère, il suffit de positionner un octet dans cette table et d'associer une fonction au caractère choisi.

Exemple:

```
? (PUTD '"#" '(LAMBDA () (EVAL (READ))))
=(LAMBDA NIL (EVAL (READ)))
? (RADIX 16) (MEMORY (+ 8000 3) FF)
=10
=0    # a pour code ASCII 23h, la table commence au caractère 20h...
? (SETQ N 8) (PUTD 'TEST '(LAMBDA (X) (/ X #(* N N))))
=8
=(LAMBDA (X) (/ X 64))
      et on constate que l'expression précédée de # a été évaluée...
? (QUOTE # (+ 2 2))
=4    ... même à l'intérieur d'une expression protégée
```

c) Macro-fonctions

Les macro-fonctions sont des fonctions à double évaluation définies ainsi :

```
      (MLAMBDA param exp1 exp2 ... expN)
```

Un seul symbole est fourni en paramètre formel; à l'exécution il est lié à l'ensemble de la liste invoquant la macro (sans évaluation), y compris la fonction qui apparaît en tête de liste. Le corps de la macro est exécuté (les expressions exp1 ... expN) et le résultat est de nouveau fourni à l'évaluation ! Le but d'une macro est donc le suivant : remplacer l'appel à la macro par une autre expression.

Exemple:

```
(PUTD 'IF '(MLAMBDA L
  (LIST 'COND
      (LIST (CADR L) (CADDR L))
      (LIST (CAR (CDDDR L)))
)))
      définit une macro "IF" qui va construire une expression COND
      contenant les arguments du IF.
```

Ainsi,

```
(IF (PLUSP N) N (- 0 N))
```

va construire la liste

```
(COND ((PLUSP N) N) ((- 0 N)))
```

puis celle-ci sera évaluée, donnant la valeur absolue de N.

Ce mécanisme puissant qui peut paraître bien inefficace pour définir une fonction IF (appel de la macro, parcours des arguments et allocation de paires pointées pour construire une expression COND, avant d'enfin évaluer cette dernière), surtout lorsque l'on compare à la version du IF avec FLAMBDA, révèle tout son intérêt dans le cadre des macro-fonctions "écrasantes". Modifions légèrement la définition du IF ainsi :

```
      (PUTD 'IF '(MLAMBDA L
        (RPLACA L 'COND)
        (RPLACD L (LIST (LIST (CADR L) (CADDR L))
                    (LIST (CAR (CDDDR L)))))
      ))
```

et maintenant le IF va se remplacer une fois pour toute par un COND à la première exécution.

Exemple:

```
      ? (PUTD 'FACT '(LAMBDA (N)
      ?    (IF (ZEROP N) 1 (* N (FACT (- N 1))))))
      =(LAMBDA (N) (IF (ZEROP N) 1 (* N (FACT (- N 1)))))
      ? (FACT 3)
      =6
      ? (GETD 'FACT)
      =(LAMBDA (N) (COND ((ZEROP N) 1) ((* N (FACT (- N 1))))))
```

à la première exécution du IF, le code de FACT s'est modifié ! Le code est maintenant plus efficace que si on utilisait un IF implémenté avec une FLAMBDA !

Ce genre de manipulation est intéressant dans le cadre de portage de programmes LISP d'une machine à une autre (par exemple, ici, lorsque le IF n'existe pas) ou pour définir efficacement des fonctions. Les macros ne sont jamais très lisibles, et plus délicates à écrire que des LAMBDA normales, mais utiliser des macros rend souvent un programme plus lisible sans perdre en efficacité.

Exemple:

la construction LET existe dans de nombreux LISP pour définir des symboles locaux à un bloc. Ainsi, le IF précédent s'écrirait plus lisiblement :

```
      (PUTD 'IF '(MLAMBDA L
        (LET (
            (X (CADR L))
            (Y (CADDR L))
            (Z (CAR (CDDDR L)))
             )
           (RPLACA L 'COND)
           (RPLACD L (LIST (LIST X Y) (LIST Z)))
      )))
```

le LET peut être défini efficacement par une macro le remplaçant en LAMBDA :

```
      (PUTD 'LET '(MLAMBDA L
        (RPLACA L (CONS 'LAMBDA
                    (CONS (ALLCAR (CADR L))
                        (CDDR L))))
        (RPLACD L (ALLCADR (CADR L)))
      ))
```

où ALLCAR et ALLCADR permettent de récupérer respectivement les paramètres formels et les arguments effectifs :

```
      (PUTD 'ALLCAR '(LAMBDA (L)
       (COND ((NULL L) NIL)
             ((CONS (CAAR L) (ALLCAR (CDR L)))))))
      (PUTD 'ALLCADR '(LAMBDA (L)
       (COND ((NULL L) NIL)
             ((CONS (CADAR L) (ALLCADR (CDR L)))))))
```

d) Fonctions d'ordre supérieur

OricLisp permet de passer une fonction en paramètre d'une autre fonction. Toutefois, le premier élément d'une liste (la fonction) n'est jamais évalué par la fonction EVAL, on ne peut donc invoquer directement une fonction obtenue en argument. Par exemple, la fonctionnelle classique MAP (qui applique une fonction à tous les éléments d'une liste et renvoie la liste des résultats) ne peut s'écrire ainsi avec OricLisp :

```
(PUTD 'MAP '(LAMBDA (F L)
  (COND ((NULL L) NIL)
      ((CONS (F (CAR L))
             (MAP F (CDR L)))))))
```

parce que le symbole F en position de fonction ne sera pas évalué (Remarque : dans beaucoup de dialectes LISP, cette définition de MAP ne marche que si F n'a pas de définition associée).
De plus, OricLisp n'a pas de fonction APPLY mais les constructions de type (APPLY F ARGS) peuvent être facilement remplacées par (EVAL (CONS F ARGS)). Ainsi, le MAP peut s'écrire :

```
(PUTD 'MAP '(LAMBDA (F L)
  (COND ((NULL L) NIL)
      ((CONS (APPLY F (LIST (CAR L)))
             (MAP F (CDR L)))))))
```

et cette définition marche sans restriction. APPLY peut être défini par une simple (LAMBDA (F L) (EVAL (CONS F L)))
ou par une macro :

```
(MLAMBDA L
  (RPLACA L 'EVAL)
  (RPLACD L (LIST (LIST 'CONS (EVAL (CADR L)) (EVAL (CADDR L))))))).
```

Exemple:

```
? (MAP '(LAMBDA(X)(* X X)) '(1 2 3 4 5))
=(1 4 9 16 25)
? (MAP 'PLUSP '(1 8 -2 6 -3))
=(T T NIL T NIL)
```

A. Annexe A: Copyright

B. Annexe B: Définitions formelles

Pour le "LISPien", une fonction LISP vaut mieux qu'un long discours, comment mieux décrire le comportement d'une primitive sinon avec son code écrit en LISP ? Bon nombre des fonctions implémentées en langage machine dans OricLisp peuvent être écrites en LISP, mais elle seraient alors moins efficaces et souvent très consommatrices d'espace de pile (à cause des récursivités).

Le noyau vraiment minimal est composé des définitions LAMBDA, FLAMBDA, MLAMBDA et des primitives READ, EVAL, PRIN, COND, CAR, CDR, CONS, PUTD, GETD, NULL, ATOM, RPLACA, RPLACD, EQ, TIME, MEMORY, RADIX, <, >, +, -, *, /, MOD. Les autres peuvent s'écrire en fonction de ces primitives de base. Les définitions qui suivent ont exactement le même comportement que les routines en langage machine (elles renvoient les mêmes résultats en toutes circonstances), même si l'implémentation effective est différente pour des raisons de performance. Les seules primitives implémentées de façon récursive sont LIST et APPEND (elles ne le devraient pas...). Enfin, APPLY n'est pas un symbole défini dans OricLisp, mais il existe en interne; il peut être défini par
(PUTD 'APPLY '(LAMBDA (F ARGS) (EVAL (CONS F ARGS))))
même si l'implémentation réelle ne consomme pas de paire pointée.

```
(PUTD 'AND
  '(FLAMBDA L
      (COND ((ATOM L) T)
            ((ATOM (CDR L)) (EVAL (CAR L)))
            ((EVAL (CAR L)) (APPLY 'AND (CDR L))))))

(PUTD 'APPEND
  '(LAMBDA (X Y)
      (COND ((ATOM X) Y)
          ((CONS (CAR X) (APPEND (CDR X) Y))))))

(PUTD 'ASSOC '(LAMBDA (X L)
            (COND ((ATOM L) NIL)
                        ((ATOM (CAR L)) (ASSOC X (CDR L))
                        ((EQUAL X (CAAR L)) (CAR L))
                        ((ASSOC X (CDR L))))))

(PUTD 'CAAAR '(LAMBDA (X) (CAAR (CAR X))))
(PUTD 'CAADR '(LAMBDA (X) (CAAR (CDR X))))
(PUTD 'CADAR '(LAMBDA (X) (CADR (CAR X))))
(PUTD 'CADDR '(LAMBDA (X) (CADR (CDR X))))
(PUTD 'CDAAR '(LAMBDA (X) (CDAR (CAR X))))
(PUTD 'CDADR '(LAMBDA (X) (CDAR (CDR X))))
(PUTD 'CDDAR '(LAMBDA (X) (CDDR (CAR X))))
(PUTD 'CDDDR '(LAMBDA (X) (CDDR (CDR X))))
(PUTD 'CAAR '(LAMBDA (X) (CAR (CAR X))))
(PUTD 'CADR '(LAMBDA (X) (CAR (CDR X))))
(PUTD 'CDAR '(LAMBDA (X) (CDR (CAR X))))
(PUTD 'CDDR '(LAMBDA (X) (CDR (CDR X))))

(PUTD 'DIV '(LAMBDA (X Y) (CONS (/ X Y) (MOD X Y))))

(PUTD 'EQUAL
  '(LAMBDA (X Y)
       (COND ((ATOM X) (EQ X Y))
           ((ATOM Y) NIL)
           ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y))))))
```

```
(PUTD 'LAST '(LAMBDA (L)
            (COND ((ATOM L) NIL)
                        ((ATOM (CDR L)) L)
                        ((LAST (CDR L))))))

(PUTD 'LENGTH '(LAMBDA (L)
             (COND      ((ATOM L) 0)
                        ((+ 1 (LENGTH (CDR L)))))))

(PUTD 'LIST
  '(FLAMBDA L
      (COND ((ATOM L) NIL)
          ((CONS (EVAL (CAR L)) (APPLY 'LIST (CDR L)))))))

(PUTD 'MEMBER
  '(LAMBDA (X L)
      (COND ((ATOM L) NIL)
            ((EQUAL X (CAR L)) L)
            ((MEMBER X (CDR L))))))

(PUTD 'NCONC '(LAMBDA (X Y)
                  (COND ((ATOM X) Y)
                  (T (RPLACD (LAST X) Y) X))))

(PUTD 'OR '(FLAMBDA L
            (COND ((ATOM L) NIL)
                        ((EVAL (CAR L)))
                        ((APPLY 'OR (CDR L))))))

(PUTD 'PROG1
  '(FLAMBDA L
      (COND ((ATOM L) NIL)
            (T ((LAMBDA (X) (APPLY 'PROGN (CDR L)) X) (EVAL (CAR L)))))))

(PUTD 'PROGN
  '(FLAMBDA L
      (COND ((ATOM L) NIL)
            ((ATOM (CDR L)) (EVAL (CAR L)))
            (T (EVAL (CAR L)) (APPLY 'PROGN (CDR L))))))

(PUTD 'QUOTE '(FLAMBDA L (CAR L)))

(PUTD 'REVERSE
  '(LAMBDA (X Y)
      (COND ((ATOM X) NIL)
            ((ATOM (CDR X)) (CONS (CAR X) Y))
            ((REVERSE (CDR X) (CONS (CAR X) Y))))))

(PUTD 'SET '(LAMBDA (X Y) (RPLACA X Y) Y)))

(PUTD 'SETQ '(FLAMBDA L (COND ((NAME (CAR L)) (SET (CAR L) (EVAL (CADR L)))))))

(PUTD 'WHILE
  '(FLAMBDA L
      (COND ((EVAL (CAR L))
            (APPLY 'PROGN (CDR L))
            (APPLY 'WHILE L)))))
```

C. Annexe C: portage de programmes MuLisp

En dehors de l'évaluation des fonctions utilisateurs (LAMBDA, FLAMBDA, MLAMBDA), OricLisp possède des primitives très proches de celles de MuLisp, il est donc aisé de porter des programmes de MuLisp vers OricLisp (et vice-versa). Pour laisser plus de place à l'utilisateur, OricLisp ne définit pas toutes les fonctions qui existent dans MuLisp.
On pourra utiliser les définitions suivantes qui fournissent un équivalent exact :

```
(PUTD 'NTH '(LAMBDA (N L)
      (COND ((ZEROP N) (CAR L))
            ((ATOM L) NIL)
            ((NTH (- N 1) (CDR L)))))) pour une version à la MacLisp

(PUTD 'NTH '(LAMBDA (L N)
      (COND ((EQ N 1) L)
            ((ATOM L) NIL)
            ((NTH (CDR L) (- N 1)))))) pour une version à la InterLisp

(PUTD 'TCONC '(LAMBDA (PTR OBJ)
      (SETQ OBJ (LIST OBJ))
      (COND ((ATOM PTR) (CONS OBJ OBJ))
            ((ATOM (CDR PTR)) (RPLACA PTR OBJ) (RPLACD PTR OBJ))
            (T (RPLACD (CDR PTR) OBJ) (RPLACD PTR OBJ)))))

(PUTD 'LCONC '(LAMBDA (PTR L)
      (COND ((ATOM L) PTR)
            ((ATOM PTR) (CONS L (LAST L)))
            ((ATOM (CDR PTR)) (RPLACA PTR L) (RPLACD PTR (LAST L)))
            (T (RPLACD (CDR PTR) L) (RPLACD PTR (LAST L))))))

(PUTD 'EVENP '(LAMBDA (N) (COND ((NUMBERP N) (ZEROP (MOD N 2))))))

(PUTD 'POP '(FLAMBDA P
      (COND ((NOT (NAME (CAR P))) NIL)
            ((ATOM (CAAR P)) NIL)
            ((PROG1 (CAAAR P) (SET (CAR P) (CDAAR P)))))))

(PUTD 'PUSH '(FLAMBDA P
      (COND ((NULL (CADR P)) NIL)
            ((NAME (CADR P)) (SET (CADR P) (CONS (EVAL (CAR P))
                                    (CADR P)))))))

(PUTD 'PUT '(LAMBDA (NAM KEY OBJ)
  ((LAMBDA(ELEM)
      (COND ((NULL ELEM) (RPLACD NAM (CONS (CONS KEY OBJ) (CDR NAM))) OBJ)
            (T (RPLACD ELEM OBJ) OBJ)))  (ASSOC KEY (CDR NAM)))))

(PUTD 'GET '(LAMBDA (NAM KEY) (CDR (ASSOC KEY (CDR NAM)))))
```

```
(PUTD 'REMPROP '(LAMBDA (NAM KEY)
      (COND ((ATOM (CDR NAM)) NIL)
            ((EQUAL (CAADR NAM) KEY)
                  (SETQ KEY (CDADR NAM))
                  (RPLACD NAM (CDDR NAM))
                  KEY)
            ((REMPROP (CDR NAM) KEY)))))

(PUTD 'FLAGP '(LAMBDA (NAM ATT) (MEMBER ATT (CDR NAM))))

(PUTD 'FLAG '(LAMBDA (NAM ATT)
      (COND ((FLAGP NAM ATT) ATT)
            (T (RPLACD NAM (CONS ATT (CDR NAM))) ATT))))

(PUTD 'REMFLAG '(LAMBDA (NAM ATT)
      (COND ((ATOM (CDR NAM)) NIL)
            ((EQUAL ATT (CADR NAM)) (RPLACD NAM (CDDR NAM)) T)
            ((REMFLAG (CDR NAM) ATT)))))

(PUTD 'GCD '(LAMBDA (X Y)
      (COND ((NOT (ZEROP Y)) (GCD Y (MOD X Y)))
            ((PLUSP X) X)
            ((- 0 X)))))

(PUTD 'COMMENT '(FLAMBDA L NIL))
```

Les différences suivantes peuvent apparaître lors d'un portage :

* le corps d'une fonction MuLisp et d'une fonction OricLisp sont évalués différemment. MuLisp implémente un COND/PROGN implicite au lieu d'un PROGN implicite dans les corps de fonction (et les clauses de COND), ce qui permet de réduire la taille du code mais introduit une ambiguïté dans le cas de LAMBDAs qui sont alors prises pour des prédicats. Le COND est obligatoire dans les corps de fonction OricLisp.

* PLUSP renvoie NIL pour 0 dans MuLisp et T dans OricLisp. De plus, le CDR d'un nombre (la liste de propriétés) à une signification inversée : il vaut T pour un nombre positif ou nul dans OricLisp, tandis qu'il vaut T pour un nombre négatif dans MuLisp.
GREATERP et LESSP acceptent plus de deux arguments dans MuLisp, de telles expressions doivent être écrites avec plusieurs > et < dans OricLisp. De même, PLUS, DIFFERENCE, TIMES admettent plus de deux arguments dans MuLisp, il est facile de réécrire ces expressions avec plusieurs +, - ou *.

* les fonctions sur les chaînes de caractères de MuLisp (SUBSTRING, FINDSTRING, PACK, UNPACK, LENGTH, ASCII) ne sont pas implémentables avec OricLisp.

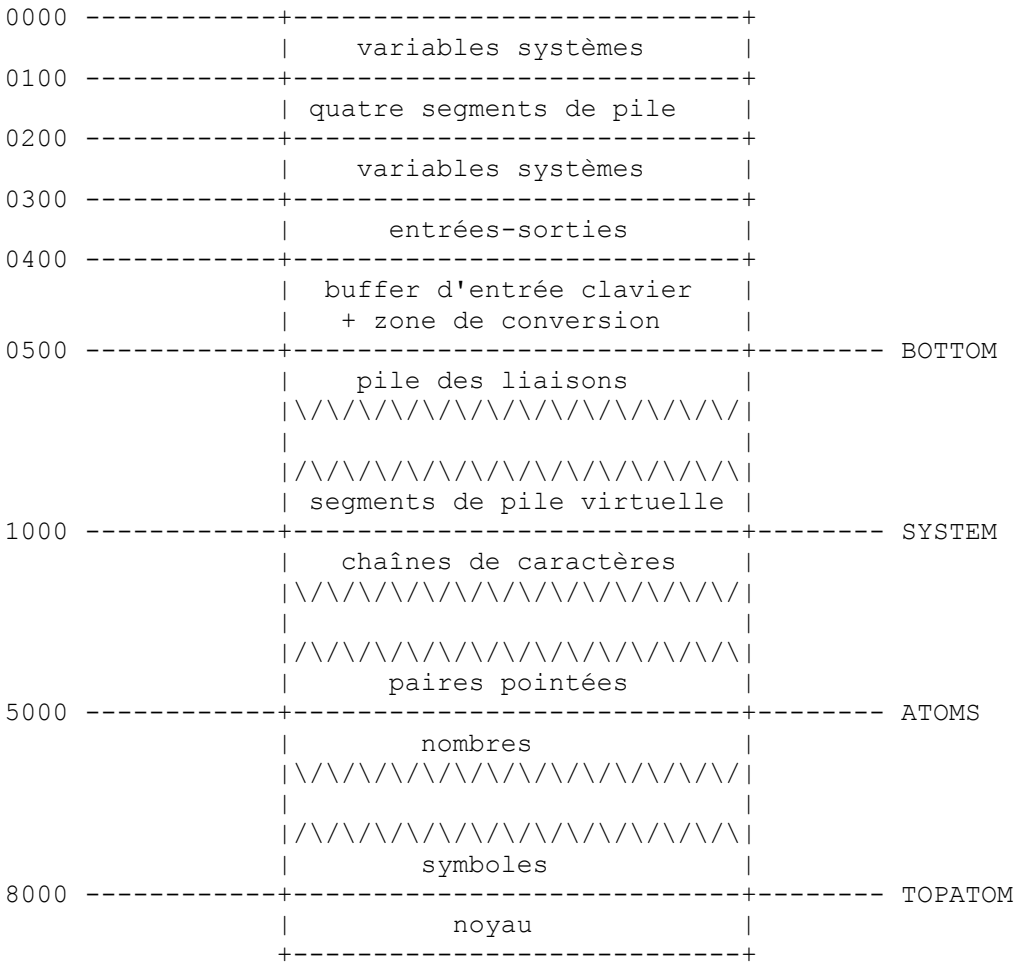
* les échappements de MuLisp (CATCH, THROW) ne sont pas implémentables avec OricLisp, le traitement d'erreur non plus.

* les fonctions de lecture/écriture sur fichier ne sont pas disponibles avec OricLisp.

* les puissantes macro-fonctions d'OricLisp n'existent pas avec MuLisp, si un programme OricLisp les utilise intensivement, on préfèrera alors porter vers un autre dialecte comme LeLisp.

D. Annexe D: Détails d'implémentation

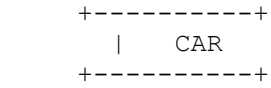
a) Carte mémoire



Les adresses BOTTOM, SYSTEM et ATOMS peuvent être modifiées pour s'adapter aux consommations différentes en nombres et symboles, paires pointées ou récursivité d'une application. Malheureusement, cette transformation doit se faire en modifiant des adresses du noyau AVANT qu'il ne s'exécute...
La page de BOTTOM peut être changée en modifiant l'adresse 9AA6, la page de SYSTEM à l'adresse 9AAA et la page d'ATOMS à l'adresse 9AB0.

b) Structures de données

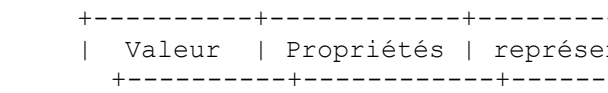
Les paires pointées ont la structure suivante:



Les symboles ont la structure suivante :

Le CAR d'un symbole donne donc sa valeur, un symbole n'ayant pas reçu de valeur a pour valeur lui même (la valeur pointe vers le même symbole).
Le CDR d'un symbole porte la liste de propriétés associée au symbole, elle est NIL par défaut et peut être changée par RPLACD.
Le champ fonction du symbole est NIL si aucune fonction n'est associée, il pointe vers la liste LAMBDA (ou FLAMBDA ou MLAMBDA) pour une fonction utilisateur, et contient l'adresse de début pour une fonction en langage machine (les deux derniers sont différenciés par le bit de poids fort, il est à 1 pour une routine en langage machine au delà de l'adresse 8000h et à 0 pour une fonction utilisateur)
Le champ nom externe (ou Print-name) pointe vers la chaîne de caractères représentant le symbole.

Les nombres ont la structure suivante:



Le CAR d'un nombre (sa valeur) pointe toujours sur lui-même.
Le CDR d'un nombre (sa propriété) est T s'il est positif ou nul, NIL si négatif.

c) La pile virtuelle

Le langage LISP est par définition récursif et ceci pose problème pour l'implémentation sur le processeur 6502 qui possède un pointeur de pile 8 bits. OricLisp implémente une pile virtuelle inédite qui lui dispense de simuler une pile 16 bits, et lui permet d'utiliser les instructions normales de pile du 6502 : JSR, RTS, PHA, PLA...
La pile virtuelle est constituée de segments de 64 octets, 4 segments sont donc présents dans la pile physique du 6502. De temps à autre (dans la routine EVAL par exemple), la position du pointeur de pile est vérifiée afin de détecter si une frontière de segment a été franchie. Ce franchissement peut provoquer la sauvegarde d'un segment de la pile physique vers la pile virtuelle, ou la récupération d'un segment de la pile virtuelle, mais l'algorithme utilisé est adapté au parcours récursif d'arbres binaires et fait que le transfert d'un segment est peu courant (il n'y a pas de transfert lorsque le pointeur oscille de part et d'autre d'une frontière de segment).

d) Le ramasse-miettes

Une première passe marque les paires et les atomes accessibles à partir des champs valeur, propriétés et fonction des symboles (on ne marque pas un symbole dont la valeur pointe sur lui-même et qui n'a pas définition de fonction), ou à partir de la pile des liaisons.
Une deuxième passe supprime les paires, les symboles et les nombres non marqués et les compacte.
Une troisième passe enlève le marquage des objets et ajuste les pointeurs vers des objets déplacés par le compactage.
Une dernière passe compacte les chaînes des symboles qui n'ont pas été éliminés par le ramasse-miettes.

E. Annexe E: Exemple du parcours du cavalier

Le parcours du cavalier sur un échiquier est un problème classique qui consiste à parcourir toutes les cases de l'échiquier une fois et une seule en respectant la marche du cavalier aux échecs. LISP est bien adapté à ce genre de recherche de solution, en voici une résolution assez concise pour un échiquier de taille N quelconque:

```
; une liste des 8 sauts possibles du cavalier, classés par une heuristique :
(SETQ DIR '((-2 1)(-2 -1)(-1 -2)(1 -2)(2 -1)(2 1)(1 2)(-1 2)))

; d'abord une petite fonction qui construit la liste des entiers M,M-1...1
(PUTD REBOURS '(LAMBDA (M) (COND ((ZEROP M) NIL) ((CONS M (REBOURS (- M 1)))))))

; puis une autre qui teste si une case n'est pas en dehors de l'échiquier
(PUTD TEST '(LAMBDA (X Y) (AND (< -1 X) (< X N) (< -1 Y) (< Y N))))

; pour construire la liste des positions atteignables à partir de M
(PUTD SAUTS '(LAMBDA (PAS)
  (COND ((NULL PAS) NIL)
      ((TEST (+ (CAAR PAS) (MOD (- M 1) N)) (+ (CADAR PAS) (/ (- M 1) N)))
         (CONS (+ (+ M (CAAR PAS)) (* N (CADAR PAS))) (SAUTS (CDR PAS))))
      ((SAUTS (CDR PAS))))))

; on construit en effet une table de ces listes pour l'ensemble des cases
(PUTD TABLE '(LAMBDA (M)
  (COND     ((ZEROP M) NIL)
      ((CONS (CONS M (SAUTS DIR)) (TABLE (- M 1))))))

; maintenant le programme lui-même: cherche un chemin par toutes les cases
(PUTD CHEMIN '(LAMBDA (CASES RESTE PARCOURS)
  (COND     ((ZEROP RESTE) PARCOURS)            ; trouvé !
      ((NULL CASES) NIL)                  ; une impasse...
      ((MEMBER (CAR CASES) PARCOURS)            ; déjà passé par ici...
            (CHEMIN (CDR CASES) RESTE PARCOURS))
      ((CHEMIN (CDR (ASSOC (CAR CASES) TAB))
             (- REST 1)
             (CONS (CAR CASES) PARC)))    ; on essaie par là...
      ((CHEMIN (CDR CASES) REST PARC))))) ; et les autres aussi...

; un programme principal pour lancer le tout, et c'est fini !
(PUTD PARCOURS '(LAMBDA (N TAB)
  (SETQ TAB (TABLE (* N N)))       ; on calcule la table une seule fois
  (CHEMIN (REBOURS (* N N)) (* N N))))    ; puis on cherche
```

Remarque: Pour trouver toutes les solutions, il suffit de remplacer la première clause de la fonction chemin par :
`(COND ((ZEROP RESTE) (PRINT PARCOURS) NIL)`

Remarque2: Une image mémoire est fournie avec OricLisp, contenant une version améliorée de ce programme pour trouver plus rapidement une solution.