Universidad Nacional Autónoma de México

Unidad Académica de los Ciclos
Profesional y de Postgrado del
Colegio de Ciencias y Humanidades

UN INTERPRETE DE LISP EN ' C '
PARA MICROCOMPUTADORAS PC

T E S I S

QUE PARA OBTENER EL GRADO DE :

Maestro en Ciencias de la Computación

P R E S E N T A :

AMPARO LOPEZ GAONA

México, D.F. 1987

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Indice

Introducción

En la actualidad es cada vez más frecuente que el hombre utilice la computadora como herramienta para resolver cualquier problema que se le presenta. Sin embargo, para poder hacer uso de esta poderosa herramienta, es necesario que exista comunicación entre el hombre y la máquina; esta se realiza a través de los lenguajes de programación.

Los lenguajes de programación se encuentran en constante evolución desde la década de los cincuentas, en que se empezaron a describir sus principales conceptos, además de desarrollarse los primeros lenguajes de alto nivel, tales como Fortran, Algol 60, Cobol, IPL V, Comit y Lisp.

Se han desarrollado intérpretes para diferentes dialectos de Lisp pues el lenguaje original se ha modificado para que satisfaga diferentes necesidades, lo que ha provocado una falta de estandarización del mismo. Recientemente se ha venido trabajando en la búsqueda de una versión estándar de Lisp, el resultado de estos trabajos es el llamado COMMON LISP.

El intérprete presentado en este trabajo, se desarrolló en y para microcomputadoras PC, y está basado en Common Lisp; sus características principales son:

Eficiencia, tanto en lo que se refiere al tiempo de ejecución como en lo referente a la administración de la memoria.

Portabilidad. El intérprete puede trabajar en micros con sistema operativo MS- DOS, así como con sistema UNIX, con algunas restricciones mínimas.

Cuenta con facilidades para programación tales como un editor, un *"pretty printer"*, y una serie de funciones para graficación.

Cuenta con facilidades para depurar y corregir programas, tales como, interrupción de la ejecución al momento de detección de errores y funciones de monitoreo.

La propuesta de este proyecto como tema de tesis, se hace basada en varias consideraciones que a continuación se exponen brevemente:

Utilidad. Cabe mencionar que el intérprete deberá ser de una calidad equiparable a los intérpretes comerciales existentes. Además la implantación en micros PC, le dará grandes posibilidades de difusión.

Importancia de Lisp. Es cada vez más popular el uso de Lisp, debido principalmente al auge de la Inteligencia Artificial, campo en que es ampliamente utilizado. y a su simplicidad.

Aportación de conocimientos. Si bien los conocimientos teóricos para la implantación de un intérprete son del dominio público, en la práctica se presentan gran cantidad de "detalles" a los cuales se enfrenta el diseñador del intérprete. La correcta solución de todos esos "puntos finos" y el enorme trabajo involucrado, definitivamente aportan una serie de conocimientos sumamente valiosos y difícilmente cuantificables. Además estos conocimientos nos situan en una posibilidad real, no teórica, de implantar software cada vez más evolucionado.

Expansiones. El hecho de contar con los programas fuentes del intérprete permiten que en cualquier momento se le puedan hacer ampliaciones tales como agregarle un intérprete de PROLOG. (trabajo realizado en la actualidad).

El intérprete está realizado en lenguaje 'C' pues es un lenguaje diseñado para desarrollos como este, en el cual es importante tanto la rapidez de ejecución, como el ahorro en el código generado y la facilidad de transporte de un equipo a otro.

El trabajo se divide en dos partes fundamentales que son el lenguaje y la implantación del intérprete. Así en el capítulo uno se describe brevemente el desarrollo histórico de Lisp; en el capítulo dos se da tanto la definición sintáctica como la semántica de Lisp, además de describir las funciones que constituyen el llamado "Lisp puro". En el capítulo tres, se explica la forma de trabajar con el intérprete y se describen todas las funciones primitivas del mismo. Una vez que ya se conoce el lenguaje,

en el capítulo cuatro se describe el diseño y la implantación del intérprete, detallando las estructuras de datos empleadas, la forma de representar las expresiones simbólicas del lenguaje en la memoria de la computadora, así como los principales módulos en que se divide el sistema. Finalmente se describen varios algoritmos para recolección de la basura generada por el intérprete y se justifica el algoritmo empleado.

1
Historia de Lisp

En este capítulo se describe el desarrollo histórico del lenguaje de programación Lisp el cual a pesar de ser de los primeros en desarrollarse continua siendo ampliamente utilizado en el campo de la inteligencia artificial.

El lenguaje de progamación Lisp (de List Processor) fue desarrollado para la IBM-704 por John McCarthy y su grupo de trabajo en el área de inteligencia artificial en el Instituto de Tecnología de Massachusetts (M.I.T). El lenguaje se diseñó para facilitar los experimentos con un sistema llamado *"The Advice Taker"*. La principal necesidad del sistema, era un lenguaje de programación capaz de trabajar con expresiones que representaban proposiciones tanto declarativas como imperativas. Estas expresiones debían estar formalizadas de manera que el *Advice Taker* pudiera hacer deducciones.

La propuesta original se presentó en Noviembre de 1958. El lenguaje se basó principalmente en el cálculo lambda, desarrollado por Church, de ahí que la forma natural de representar la información sea a través de listas ligadas, y la forma de implantarlo sea por medio de un intérprete (usando una función llamada *apply*).

McCarthy empleó el modelo de Lisp como punto de partida para desarrollar una teoría matemática de la computación, considerando muchas de las bases teóricas de los lenguajes de programación varios años antes que fueran considerados por alguna otra persona o grupo de trabajo.

La primera implantación de un intérprete de Lisp fue hecha en forma experimental en una IBM 704 en el año 1960, ya comercialmente se implantó en 1962, luego en 1963 apareció una versión para PDP-1; más tarde para una PDP-6 y para una

PDP-10. En la actualidad existen intérpretes de Lisp en todo tipo de computadoras.

Durante el período de la primera versión (1959 – 1962), al original "Lisp puro" (constituido por las funciones *atom, car, cdr, cons, eq*) se le agregaron varias características, entre ellas las listas de propiedad, funciones aritméticas eficientes, variables libres, iteración, y la función *eval*. Todas ellas combinadas formaron lo que se conoce como Lisp 1.5.

El dialecto Lisp 1.6 se desarrolló para una DEC-10 bajo el sistema operativo TOPS-10. Este dialecto originó dos importantes versiones, una en Stanford que conserva el nombre de Lisp 1.6 y otra en el MIT llamada MACLisp. La última existe bajo el TOPS-10 y bajo el sistema operativo local llamado ITS *(Incompatible Timesharing System)*.

Un dialecto desarrollado originalmente por Bolt, Beranek and Newman, Inc. (BBN) bajo el nombre de BBN-Lisp, sufrió cambios hasta convertirse en INTERLISP. Su primera implantación fue en una DEC-10 bajo el sistema operativo TENEX, también desarrollado por BBN. INTERLISP se ha adaptado para poder usarse bajo el sistema operativo TOPS-20, aunque también existen implantaciones para la IBM 360 y 370 así como para otras computadoras.

En la Univesidad Nacional Autónoma de México (UNAM) también se han desarrollado intérpretes para Lisp, uno es el Lisp 1.6 implantado por Mario Magidin y Raymundo Segovia, en una computadora B-6700 en el año de 1972. Más tarde, en 1982, Miguel Tomasena, hizo una implantación para Lisp 1.6 en una PDP-11/34.

Existen muchos otros dialectos e implantaciones, aunque una lista completa de ellos no está disponible.

Para terminar con la fama de que la ejecución de programas en Lisp, es lenta, en Berkely, se han desarrollado compiladores llamados "Franz Lisp". Hace poco, se desarrollaron las "máquinas Lisp", que reemplazan la arquitectura tradicional de Von Newman, por una que puede evaluar en forma directa (por hardware) programas en Lisp. Estas máquinas proporcionan una gran herramienta para los experimentos en inteligencia artificial que se desarrollan usando Lisp. El poder de las máquinas Lisp no

se deriva sólo del hecho de ejecución directa, sino que proporciona varias herramientas para desarrollos, como son manejadores de bases de datos, herramientas de graficación, etcétera.

En la UNAM, Adolfo Guzmán y Raymundo Segovia desarrollaron una máquina Lisp con arquitectura de computadora configurable basada en el concepto de modificación dinámica de la estructura de máquina.

La definición básica de Lisp se encuentra en un documento escrito por McCarthy titulado *Recursive Functions of Symbolic Expressions* , sin embargo nunca se ha estandarizado. Recientemente, se han hecho esfuerzos para desarrrollar una versión estándar de Lisp, ésta es conocida con el nombre de COMMON Lisp.

El área para la cual se diseño Lisp es la inteligencia artificial, cuyas aplicaciones tratan con datos en forma de símbolos y estructuras de expresiones simbólicas. Sus primeras aplicaciones incluyeron programas que ejecutaban diferenciación simbólica, integración y verificación de teoremas matemáticos, ya que éstas eran las principales actividades de programación en inteligencia artificial en 1960.

Recientemente las aplicaciones de la inteligencia artificial han aumentado en las áreas de entendimiento de lenguaje natural, reconocimiento de formas, robótica, representación de conocimiento y sistemas expertos. En todas estas áreas Lisp ha sido el lenguaje de programación predominante dada la facilidad con que se pueden implantar modelos matemáticos.

Referencias

La mayoría de los libros que hacen algún tipo de análisis de Lisp hacen una breve reseña histórica del lenguaje, sin embargo en los artículos de Wagner P [1976] se describen desarrollos realizados en el campo de los lenguajes de programación, éstos en cuanto a conceptos y lenguajes propiamente dichos, en particular habla de Lisp. En McCarthy [1960] se detallan los antecedentes de Lisp y en Tucker[1985] se mencionan en orden cronológico diferentes dialectos originados a partir de Lisp puro.

Para un conocimiento amplio de desarrollos tanto de INTERLISP como de máquinas Lisp se recomiendan los artículos incluidos en Barstow[1984]. Para desarrollos realizados en la UNAM ver Magidin[1979], Guzmán [1976] y Tomasena[1980].

2
Lisp

El propósito de este capítulo es definir el lenguaje de programación LISP, el cual toma su nombre de "List Processor" y está basado en el cálculo lambda, desarrollado por Church. Se da tanto la definición sintáctica como la semántica del lenguaje además se describen las funciones que constituyen el llamado LISP puro.

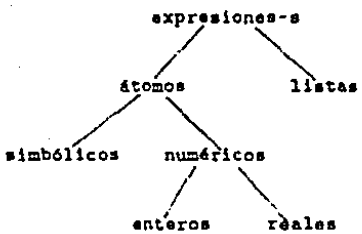
2.1 Sintaxis

Para poder definir formalmente cualquier lenguaje, primero se debe definir el alfabeto o conjunto de símbolos con que se pueden construir los elementos del lenguaje. El alfabeto de LISP consta de letras, dígitos y los siguientes caracteres especiales: ', *, /, =, >, <, \ , +, -, ', @, |, ! y ,.

Con los caracteres o símbolos del alfabeto pueden formarse los elementos del lenguaje denominados por McCarthy expresiones simbólicas (expresiones-s). Estas expresiones-s fueron definidas por McCarthy de la siguiente forma:

1. nil es una expresión-s.
2. Los átomos son expresiones-s.
3. Si e1 y e2 son expresiones-s, entonces
   (e1.e2) es una expresión-s.

En las posteriores definiciones de Lisp, las expresiones simbólicas se han clasificado en átomos y listas. Los átomos a su vez pueden ser simbólicos o numéricos, y éstos últimos pueden ser tanto reales como enteros, como se muestra en la figura.



expresiones-s

Un **átomo** es un elemento del lenguaje que no puede dividirse en unidades más pequeñas que tengan algún significado como elementos del átomo. Ejemplo: *Atomo* es un átomo y las letras que lo constituyen ya no tienen significado para Lisp, como parte de él.

Las computadoras trabajan con el subconjunto de números determinado por las características físicas de la máquina. A los elementos del subconjunto de enteros, en Lisp, se les llama **átomos numéricos enteros** y a los del subconjunto de reales, **átomos numéricos reales.**

Un átomo simbólico es una cadena de hasta treinta y dos caracteres de los cuales el primero debe ser una letra y los restantes pueden ser letras, dígitos o bien guiones (-). Si se desea que un átomo simbólico tenga algún caracter diferente de los mencionados, el átomo debe delimitarse por signos de admiración (!). Como ejemplos de átomos simbólicos se tienen los siguientes: *nombre*, *a23*, *soy-atomo-simbolico*, *!b:archivo.lis!*.

Además se consideran átomos simbólicos algunos caracteres que tienen significado para el intérprete el cual será explicado en otra sección. Los caracteres mencionados son: ', *, /, =, >, <, \ , ', @, y ,. Los símbolos de +, - se consideran átomos simbólicos o parte de uno numérico dependiendo del contexto en que se encuentren.

Una lista puede definirse como un paréntesis izquierdo seguido de cero o más átomos y/o listas y finalmente un paréntesis derecho, aunque también puede definirse de manera más formal como sigue:

1. () es una lista, llamada lista vacía
2. Si $e_1\ e_2\ \ldots\ e_n$ son átomos o listas, entonces

   $(e_1\ e_2\ \ldots\ e_n)$ es una lista.

Como ejemplos de listas se tienen las siguientes:
(soy una lista de seis elementos)
(soy (una lista de dos elementos))
((soy una lista con un solo elemento))

En el primer ejemplo se tiene una lista formada de seis átomos simbólicos, en el segundo se muestra una lista con dos elementos un átomo y una lista y en el último se trata de una lista con un solo elemento que a su vez es una lista.

La relación existente entre átomos, listas y las expresiones simbólicas definidas por McCarthy está dada por las siguientes reglas:

1. La lista vacía corresponde a la expresión-s nil.
2. Un átomo corresponde a la misma expresión-s.
3. Una lista de la forma $(e_1\ e_2\ \ldots\ e_n)$ corresponde a la expresión-s $(e_1 . (e_2 . (\ldots\ (e_n . nil) \ldots)))$

De estas reglas se puede deducir que cada lista corresponde a una expresión-s, sin embargo, no es cierto que a cada expresión-s le corresponda una lista.

2.2 Semántica

Para poder dar la definición completa del lenguaje no basta con definir las reglas de construcción de sus elementos, (en este caso las expresiones-s) se necesita también definir el significado o valor de tales elementos.

Los átomos numéricos tienen siempre asignado un valor, que es el número que representan, así el valor del átomo numérico entero 475 es 475, el del átomo numérico real -47.25 es el número -47.25.

El valor de un átomo simbólico esta indefinido hasta el momento en que el programador le asigne alguno, sin embargo, existen dos átomos simbólicos que pueden considerarse especiales, puesto que son los únicos que tienen un valor pre-definido, éstos son el átomo nil y el átomo t. El valor del átomo *nil* es *nil* y el del átomo *t* es *t* que denotan los valores lógicos de *falso* y *verdadero*, respectivamente. Esto no quiere decir que en LISP se tengan como valores Booleanos *t* y *nil*, pues cualquier cosa diferente de *nil* se considera que tiene valor lógico verdadero.

En este momento se puede notar que en *nil* existe cierta ambigëdad puesto que puede ser considerado como átomo (con valor lógico de falso) y como lista (la lista vacía) dependiendo del contexto en que se encuentre.

El valor asociado a un átomo simbólico puede ser cualquier expresión-s, en particular puede ser alguna que represente el nombre de una función, de ahí que muchos otros átomos simbólicos tengan pre-asignado un significado, en el sentido que son funciones primitivas de LISP.

Una lista representa siempre una función en notación polaca prefija, es decir, el nombre de la función a evaluar es el primer elemento de la lista, por lo tanto debe ser un átomo simbólico; y los argumentos son las expresiones-s restantes. En vista de lo cual, el valor de una lista se obtiene al aplicar la función a sus argumentos. Si la lista consta de un solo elemento, se considera que la función no tiene argumentos.

De la sintaxis y semántica del lenguaje se pueden deducir las siguientes propiedades de Lisp:

La evaluación de una expresión-s da como resultado otra expresión-s.

Sus instrucciones son todas llamadas a funciones, en lugar de proposiciones, por lo tanto la ejecución de un programa en Lisp es una problema de evaluación de funciones (expresiones-s). De ahí que Lisp se defina como un lenguaje funcional.

Los programas constan de listas en notación prefija, esto es, el primer elemento de las listas es la función que debe evaluarse y los siguientes elementos son los parámetros de dicha función. Esto da mucha uniformidad porque el nombre del procedimiento

siempre está en la misma posición sin importar la cantidad de argumentos que tenga y además la manera natural de implementarlo a través de interprétes.

De la definición de las expresiones-s se tiene que el único tipo de elementos con que se trabaja en Lisp son átomos o listas, siendo desde el punto de vista del programador la lista el principal elemento para programar.

Existe equivalencia entre datos y programas, pues ambos se representan por listas, de ahí que en un momento dado una lista que representa algún dato pueda ejecutarse como programa o bien un programa pueda servir de dato a otro.

Algunas características que se reflejan en la implementación de Lisp, son las siguientes:

La representación de las expresiones-s en la memoria de la computadora se hace por medio de listas ligadas.

Hace uso de un recolector de basura como medio de resolver el problema de recuperar el área que ocupan las expresiones-s que no son accesibles.

2.3 Lisp puro

Una vez definidos los elementos del lenguaje y su significado se puede pasar a definir las funciones que se aplicarán a estos datos. En esta sección se definen únicamente las funciones que constituyen el llamado **Lisp puro**. Se presentan en forma funcional como lo hizo McCarthy, no en su sintaxis real.

Si se va a trabajar con listas se necesita contar con alguna función que permita la construcción de las mismas, ésta se llama cons, necesita dos parámetros y regresa la lista cuyo primer elemento es el primer parámetro y el resto son los elementos del segundo parámetro. Por ejemplo:

Si A, B y m son expresiones-s entonces:
CONS [A ; B] = (A.B)
CONS [m ; nil] = (m. nil) = (m)

Una vez que se tienen definidas listas es útil poder accesar cualquier elemento de las mismas para esto existen dos funciones:

CAR con la cual se obtiene el primer elemento de la lista dada como parámetro. En caso de no ser una lista el valor que regresa no esta definido.

CDR con la cual se obtiene todos los elementos excepto el primero, de la proporcionada como parámetro. Si el parámetro no es lista el resultado esta indefinida. Ejemplos:

CAR [(A.B)] = A
CAR [A] = indefinido
CDR [(A.B)] = B
CDR [(A)] = nil
CDR [A] = indefinido

A partir de una expresión-s, se puede obtener cualquier sub-expresión-s de la misma, combinando los CAR y CDR de manera apropiada.

Una función cuyo valor es verdadero o falso se llama un **predicado**. Los predicados incluidos en Lisp puro, son los siguientes:

ATOM el cual regresa t si su argumento es un átomo y *nil* en otro caso. A continuación se muestran unos ejemplos:

ATOM [x] = T
ATOM [(x.A)] = nil
ATOM [(x)] = nil
ATOM [nil] = T

EQ el cual regresa *t* si los dos argumentos representan el mismo átomo simbólico, *nil* si no es así y un valor indefinido si los argumentos no son átomos simbólicos.

EQ [x ; x] = T
EQ [x ; y] = nil
EQ [x ; (A.B)] = indefinido

Referencias

La definición de Lisp puro se encuentra tanto en McCarthy[1965] como en McCarthy[1960], para una definición completa del original Lisp consultar McCarthy [1960].

3
MiLisp

El próposito de este capítulo es introducir a la programación en LISP y al intérprete MiLisp el cual fue elaborado para y en microcomputadoras PC con sistema operativo MS-DOS aunque también funciona en micros con sistema operativo Unix.

Los programas en LISP se ejecutan en un medio ambiente interactivo e interpretativo, propiciando que no exista un programa principal en la forma usual, en lugar de eso, el usuario desde la terminal introduce el programa principal, como una serie de expresiones que serán evaluadas. El intérprete MiLisp evalua cada expresión que le es introducida, imprimiendo automáticamente el resultado, en la terminal. En ocasiones las expresiones introducidas son definiciones de funciones, otras expresiones contienen llamadas a esas funciones, de tal suerte que la única interacción entre diferentes funciones ocurre durante la ejecución de las llamadas.

A manera de ejemplo, se muestra una función en Lisp que sirve para calcular m^n, en el ejemplo puede apreciarse que se tiene una lista con sublistas, donde cada lista representa operaciones sobre constantes, variables, iteraciones, definición de funciones, etcétera.

```
; Función para calcular m^n en forma recursiva
(Defun exp (m n)
   (cond ((zerop n) 1)
         ( t (* m (exp m (- n 1))))))
```

Al teclear la lista *defun exp (m n) ...)* se define que el átomo *exp* será una función de dos argumentos *m* y *n*. Hay que notar que no existe especificación explícita del tipo para *m* y *n*, ya que en Lisp cualquier expresión-s puede ser el valor de un átomo. El tipo de un elemento se determina al momento de ejecución, pues en las funciones se verifica el tipo de los parámetros, en caso de éste esté restrigido a cierta clase de expresiones-s.

Cond es la función para toma de decisiones y como se puede notar es también una lista, cuyos parámetros son listas de dos argumentos, el primero es la condición y el segundo el resultado. Se evalua cada condición hasta encontrar una con valor diferente de nil.

Dado que Lisp se basa en el cálculo lambda, resulta natural definir las funciones en forma recursiva, de ahí que la estructura, predominante, para control de secuencia sea la recursión.

Una variable acotada con respecto a una función es aquella en la cual las modificaciones que sufra su valor, dentro de la función, sólo son válidas dentro del marco de la misma, esto es, al salir de la función las variables acotadas conservan el valor que tenían antes de entrar en la misma. Físicamente, una variable acotada con respecto a una función, es aquella que aparece en la lista de parámetros.

De ahí que se diga que el paso de parámetros es por valor, puesto que éste se define como aquel en que la única relación entre los parámetros formales y los actuales ocurre al momento de llamada y sólo para que cada parámetro actual le de valor inicial a su correspondiente parámetro formal.

Una variable libre con respecto a una función, es aquella que no aparece en la lista de parámetros y por lo tanto si su valor cambia dentro de la función, conserva este nuevo valor al salir de la misma.

A continuación se ilustran estos conceptos por medio de un ejemplo, cabe hacer notar que a partir de este ejemplo, los resultados enviados por el intérprete se presentan con caracteres inclinados.

```
(defun incrementa (parametro)
   (setq parametro (+ parametro 10))
   (setq libre parametro))
   INCREMENTA
(setq parametro 15)
   15
(setq p-actual 10)
   10
(setq libre 10)
   10
(incrementa p-actual)
   20
libre
   20
parametro
   15
p-actual
   10
```

En este ejemplo *parametro* es una variable acotada con respecto a *incrementa*, mientras que *libre* es libre en el mismo contexto es por eso que el valor de *libre* es alterado cada vez que se efectua la función, mientras que *parametro* sólo cambia temporalmente su valor dentro de la función. El valor de *parametro* es restaurado a la salida, porque aparece en la lista de argumentos. Dado que el paso de parámetros es por valor *p-actual* no es alterado, sólo sirve para dar valor inicial a *parametro*.

En general las funciones se definen con un número fijo de argumentos que serán evaluados siempre que se ejecute tal función, pero en ocasiones es deseable poder tener algunos argumentos opcionales, en Common Lisp esto es posible, usando la palabra optional dentro de la lista de parámetros con lo cual se indica que los parámetros que siguen a esta palabra pueden estar en algunas llamadas a esta función pero en otras pueden no estar. Normalmente el valor que se asume para los parámetros opcionales es nil a menos que se especifique explícitamente alguno diferente.

A continuación se presenta un ejemplo en cual uno de los parámetros es opcional:

```
> (defun punto-final (lista  optional simbolo)
   (cond ((not simbolo) (append lista '(punto)))
         ( t (append lista (list simbolo)))))
  PUNTO-FINAL
> (punto-final '(este es un ejemplo))
  (ESTE ES UN EJEMPLO)
> (punto-final '(este es un ejemplo) 'interrogacion)
    ESTE ES UN EJEMPLO INTERROGACION)
> ; mismo ejemplo pero asignando valor inicial al
> ; parametro opcional
> (defun punto-final (lista  optional (simbolo 'punto))
    (append lista (list simbolo)))
  PUNTO-FINAL
> (punto-final '(esto es un ejemplo))
  (ESTE ES UN EJEMPLO PUNTO)
```

Los comentarios en Lisp no se consideran parte del lenguaje y por lo tanto no tienen estructura de lista. éstos empiezan con el caracter punto y coma (;) y terminan al final de la línea, además de que pueden contener cualquier caracter imprimible.

Lisp trabaja listas directamente, el cual es un tipo de datos que normalmente no está accesible en otros lenguajes. Por ejemplo si se desea realizar un programa que lea una expresión aritmética en notación prefija y la evalue. En Lisp bastaría con teclear:

```
(eval (read))
```

o escrito más explícitamente:

```
(defun evalua-expresion ()
   (print '(teclear expresion) )
   (setq exp (read) )
   (setq valor (eval x) )
   (print ' (el valor de ,exp es ,valor) ) )
```

Al ejecutarla se haría:

```
(evalua-expresion ())
(TECLEAR EXPRESION) >  (+ 3 4 5)
(EL VALOR DE (+ 3 4 5) ES 12)
```

Lisp tiene dos características que provocan que el programador lo aborrezca: la primera es la necesidad de conocer la cantidad de paréntesis que faltan para terminar de cerrar todos los abiertos, pero en MiLisp se lleva un contador automático de paréntesis sin cerrar que se imprime después de cada retorno de carro.

El otro problema es de legibilidad, el cual se ha resuelto a través de indentación automática por medio de una rutina llamada *pretty-printer*, la cual es invocada cada vez que se llama al editor, o bien se está siguiendo paso a paso la ejecución de un programa.

3.1 Forma de trabajar con Milisp

Para empezar una sesión de trabajo se debe teclear **Milisp** , en ese momento se inicializa el sistema, poniendo en memoria las funciones de biblioteca, que están escritas en Lisp y se encuentran en un archivo llamado *LISP.LIB* si este archivo no se encuentra en el mismo disco en que esta Milisp, en la pantalla aparece un mensaje indicando la falta del mismo, y el usuario debe decidir continuar o abortar la ejecución del intérprete. Al terminar la inicialización del sistema, se limpia la pantalla y en ese momento el intérprete ya esta listo para trabajar.

La forma en que trabaja el intérprete es la siguiente: lee cualquier expresión-s proporcionada por el usuario, luego la representa internamente con listas ligadas, para así poder evaluarla de acuerdo a sus reglas y finalmente imprime el resultado de esta evaluación. Para indicar que el intérprete esta listo para empezar a leer en la pantalla aparece el caracter ">". Este caracter se conoce como *prompt* .

Para Milisp son indiferentes las mayúsculas de las minúsculas pues al estar leyendo todas las letras son convertidas a mayúsculas. El intérprete envía el mensaje de error *"Caracter Inválido"* cuando dentro de alguna expresión-s encuentra algún caracter que no pertenece al alfabeto especificado en la sección anterior.

Cuando se teclea una expresión-s, el intérprete espera a que se den todos los paréntesis derechos antes de comenzar la evaluación. Si se teclea un retorno de carro

y la expresión-s que se desea evaluar no se ha terminado, en la siguiente línea aparece un número antes del *prompt* que indica la cantidad de paréntesis que no se han cerrado hasta ese momento. A continuación se muestra un ejemplo del diálogo que se establece al definir una función cuyo nombre es SEGUNDO y que se va definiendo por partes. El final de la línea, representa el retorno de carro.

```
> (Defun segundo (lista)
1> (car
2> (cdr lista)
2> ))
  SEGUNDO
>
```

Si se teclean más paréntesis derechos de los necesarios, el intérprete los ignora. La siguiente expresión-s:

```
> (+ 3 (* 5 2) (* 4 3))))))))))))))))))))))))))
    25
```

Es equivalente a :

```
> (+ 3 (* 5 2) (* 4 3))
    25
```

El intérprete imprime los resultados de sus evaluaciones dejando una sangría de dos espacios en blanco.

La evaluación de cualquier lista puede detenerse en el momento deseado con solo teclear Ctrl-D y en ese instante se tiene la opción de interrumpirla definitivamente o entrar a modo de depuración. Si se opta por interrupción definitiva, el sistema imprime el *prompt* que se tenía antes de iniciar la evaluación, para de esta manera indicar que esta listo para aceptar otra expresión-s. En caso de entrar a modo de depuración, el *prompt* cambia a ':', en ese momento se puede preguntar por el valor de cualquier átomo, alterarlo o depurar el programa de la manera adecuada, para salir de este modo de operación basta con dar la función "(bye)" y la ejecución de la función original se reanuda en el punto en que fue suspendida.

Como se ha mencionado todas las funciones a evaluar deben estar en notación prefija. La función a ejecutarse puede ser una función primitiva o bien una definida por el programador. Una función primitiva es aquella que desde el momento de iniciar la

y la expresión-s que se desea evaluar no se ha terminado, en la siguiente línea aparece un número antes del *prompt* que indica la cantidad de paréntesis que no se han cerrado hasta ese momento. A continuación se muestra un ejemplo del diálogo que se establece al definir una función cuyo nombre es SEGUNDO y que se va definiendo por partes. El final de la línea, representa el retorno de carro.

```
> (Defun segundo (lista)
1> (car
2> (cdr lista)
2> ))
  SEGUNDO
>
```

Si se teclean más paréntesis derechos de los necesarios, el intérprete los ignora. La siguiente expresión-s:

```
> (+ 3 (* 5 2) (* 4 3))))))))))))))))))))))))
    25
```

Es equivalente a :

```
> (+ 3 (* 5 2) (* 4 3))
    25
```

El intérprete imprime los resultados de sus evaluaciones dejando una sangría de dos espacios en blanco.

La evaluación de cualquier lista puede detenerse en el momento deseado con solo teclear Ctrl-D y en ese instante se tiene la opción de interrumpirla definitivamente o entrar a modo de depuración. Si se opta por interrupción definitiva, el sistema imprime el *prompt* que se tenía antes de iniciar la evaluación, para de esta manera indicar que esta listo para aceptar otra expresión-s. En caso de entrar a modo de depuración, el *prompt* cambia a ':', en ese momento se puede preguntar por el valor de cualquier átomo, alterarlo o depurar el programa de la manera adecuada, para salir de este modo de operación basta con dar la función "(bye)" y la ejecución de la función original se reanuda en el punto en que fue suspendida.

Como se ha mencionado todas las funciones a evaluar deben estar en notación prefija. La función a ejecutarse puede ser una función primitiva o bien una definida por el programador. Una función primitiva es aquella que desde el momento de iniciar la

ejecución del intérprete ya esta definida, es decir, es una función que se ha construido como parte del intérprete y por lo tanto esta en código objeto. Sin embargo este es un conjunto limitado, así que el programador tiene necesidad de crear nuevas funciones, las cuales estarán escritas en Lisp y serán interpretadas por MiLisp. Cabe mencionar que todas las funciones regresan siempre algún valor de acuerdo a la función realizada.

Las reglas que sigue Lisp y por lo tanto MiLisp para evaluar las expresiones-s son las siguientes :

1) El valor de un átomo numérico (entero o real) es el mismo.
Ejemplo:

```
> 5
  5
> 76564.87
  76564.87
> -.4e2
  -400.00
```

2) El valor de un átomo simbólico es la expresión-s que se le haya asignado, en caso de que no tenga valor, el intérprete envía el mensaje de error *"Atomo sin valor asociado"*.

Ejemplo: Supóngase que el valor de L es (A B C) y el de saludo es **HOLA** entonces se podría tener una sesión de trabajo como se muestra en seguida.

```
> l
  (A B C)
> saludo
  HOLA
> nada
  Atomo sin valor asociado
> t
  T
> nil
  NIL
```

3) Si la expresión es una lista, aplica la función que es el primer elemento de la lista al resto de los elementos que representan los parámetros, esto lo hace recursivamente para cada sublista que aparezca en la lista original. Si se tiene que el primer elemento de una lista no es función, que faltan parámetros o que el tipo de alguno no

corresponde al esperado, MiLisp envía el mensaje de error apropiado e interrumpe la evaluación de la lista original.

Ejemplos:

```
  > (+ 8 9)
17
  > ( + (+ 4 5) (+ 2 3) )
120
  > ( + 3 5 (sqrt -9) )
(sqrt -9) Se espera un entero positivo
```

Para dar por terminada una sesión de trabajo, es decir, para salir del intérprete, es necesario dar la función "(bye)".

En MiLisp se tienen funciones primitivas para asignar valores a variables, evaluar expresiones-s, realizar cálculos aritméticos, manejar listas, controlar la secuencia, verificar la validez de alguna condición, manejar procedimientos definidos por el programador, depurar programas, controlar la entrada/salida, recuperar espacio disponible, manejar arreglos, poner comentarios y terminar la ejecución del interpréte. A continuación se explica cada uno de estos grupos de funciones primitivas.

Funciones para asignación. Existen funciones para asignación de valores a variables aunque la operación de asignación no tiene un papel principal en Lisp, como en los lenguajes imperativos. Muchos programas pueden ser escritos sin hacer una sola asignación, simplemente utilizando recursión y transmisión de parámetros para obtener el mismo efecto.

Funciones para evaluación. Todas las expresiones dadas al intérprete se evaluan, pero existe una función para inhibir la evaluación de la expresión-s así como otra para que el resultado de la evaluación se vuelva a evaluar.

Funciones aritméticas. Al igual que todos los lenguajes, Lisp cuenta con funciones para realizar operaciones aritméticas, tales como suma, resta, multiplicación, división, exponenciación, raíz cuadrada, seno, coseno, obtener la división entera, convertir a número real o a entero un dato, obtener el residuo de una división, el valor absoluto de un dato, el mayor elemento de una serie de datos o el menor elemento.

Todas las operaciones aritméticas aceptan argumentos de tipo real o entero y realizan la conversión de tipo, si es necesario.

Funciones para manejo de listas. MiLisp proporciona gran variedad de funciones primitivas para la creación y modificación de listas, así como para para obtener algún elemento de la lista, o la longitud de una lista.

Casos particulares de listas son las llamadas listas de asociación y listas de propiedad. Una lista de asociación es una lista de listas, donde cada sublista consta de dos elementos.

Una lista de propiedad es una lista de asociación en la que el primer elemento de cada pareja es una propiedad del átomo al cual está asignada la lista de propiedad.

MiLisp proporciona funciones para manejo de listas de asociación y listas de propiedad, tales como buscar un elemento en una lista de asociación, recuperar, asignar o eliminar una propiedad en alguna lista de propiedad, desplegar una lista de propiedad.

Funciones para control de secuencia. El principal control de secuencia en MiLisp es la recursión, sin embargo en MiLisp se proporcionan funciones para el control de secuencia tales como una función para toma de decisiones, iteración y otras en las cuales una instrucción realiza varias asignaciones y regresan ya sea el primer valor asignado o el último.

Predicados. Los predicados son funciones primitivas que prueban la validez de una condición y regresan nil si ésta no se cumple y en caso afirmativo, generalmente, regresan t.

Entre las pruebas que se pueden hacer están la de verificar si se trata de un átomo, lista, lista vacía, función definida por el usuario, número cualquiera, negativo, par o cero, las pruebas de igualdad entre átomos, listas o números, determinar si una expresión-s pertenece a otra o si una sucesión de números es creciente o decreciente.

Funciones para manejo de procedimientos definidos por el programador. Entre éstas se encuentran funciones para definir un procedimiento con nombre o sin nombre, un macro, definir variables locales dentro de un procedimiento o pasar como parámetro el nombre de un procedimiento

Funciones de entrada/salida. Se cuenta con funciones para escribir un caracter, una expresión-s con o sin salto de línea, saltar de línea, leer una expresión-s, escribir el valor de algunos átomos dentro de una lista y otros ignorarlos. El contenido de algunas listas escribirlo como átomo y viceversa. Todo esto trabajando directamente con la terminal o con algún archivo especificado por el usuario.

También existen funciones para limpiar la pantalla, colocar el cursor en algún punto determinado de la pantalla, para dibujar líneas, ya sea en monitores a color o monocromáticos.

Funciones para depuración. Se tienen funciones para depurar programas tales como hacer la traza (indicar las rutinas en que entra y sale y cuál es el valor de los parámetros al entrar a una subrutina y cuáles valores regresa la misma), seguir paso a paso la ejecución de un programa, poner puntos de ruptura en un programa, para así poder conocer y/o modificar algunos valores. Interrumpir definitivamente un programa si se nota que está en un ciclo infinito o bien no hace lo que uno desea. El usuario puede activar un editor y trabajar con un archivo ya creado, o bien, puede al llamar al editor y crear un archivo con las funciones indicadas.

Función para recuperar espacio disponible. Por medio de esta función se llama explícitamente al colector de basura, esto puede ser útil si se desea tener el mayor espacio disponible antes de ejecutar alguna rutina.

Función para manejo de comentarios. Los comentarios son ignorados por el intérprete y existen sólo en el archivo de texto a menos que se use la función para comentarios pero con ella, cada palabra se guarda en la lista de objetos ocasionando que disminuya el espacio para nuevos objetos, por lo tanto si tienen un porcentaje alto del texto total, llegan a causar problemas con la memoria.

Funciones para manejo de arreglos. Un arreglo es una estructura de datos en la cual la información se guarda en la memoria y sólo puede accesarse a través de índices.

En MiLisp existen funciones para crear arreglos de una o dos dimensiones, accesar elementos de un arreglo, conocer los límites de un arreglo e imprimir el contenido de un arreglo.

Funciones para trabajar directamente con el procesador. En este grupo caen las funciones para leer o escribir directamente de un puerto, alterar alguna localidad de memoria o bien generar interrupciones al sistema operativo.

Función para salir del ciclo de lectura, evaluación y escritura. Para dar por terminada cualquier sesión de trabajo o bien de modo de depuración, se debe usar esta función.

3.2 Funciones primitivas de MiLisp

En las secciones anteriores se han explicado las características del lenguaje y cómo trabajar con el intérprete, sin embargo, no se han detallado las funciones proporcionadas por el intérprete y este es el objetivo de la presente sección.

La forma de describir las funciones primitivas que constituyen este intérprete, es dando el nombre de la función, una descripción de su funcionamiento, la sintaxis (para aclarar tipo, cantidad y tal vez función de los parámetros) así como un ejemplo de la función.

En cada función primitiva, se verifica tipo y número de parámetros, pero antes, se evaluan éstos, a menos que en la descripción se mencione que no hay evaluación de los parámetros. En caso de que algún parámetro tenga un tipo no esperado o de que falten parámetros, el intérprete envía un mensaje indicando el error ocurrido, y la ejecución de la rutina en donde está esta función primitiva será suspendida.

Los posibles mensajes de error son los siguientes:

Función no implementada
Atomo sin valor asociado
Faltan argumentos
El archivo no puede abrirse
El argumento debe ser una lista
El argumento debe ser un átomo simbólico
El argumento debe ser un átomo entero
El argumento debe ser un átomo real
El argumento no debe ser un átomo numérico
El argumento debe ser un átomo numérico
El argumento debe ser el nombre de un archivo ejecutable
El argumento debe ser una lista de propiedad

En la descripción sintáctica de las funciones, el tipo de expresión simbólica que se debe tener al evaluar el parámetro será especificado en letras itálicas. En caso de que no haya evaluación del parámetro el tipo será dado en letras inclinadas.

Se esperan los siguientes tipos de los valores de los parámetros: *expresión-s, lista, átomo, número, entero, real, número positivo, entero-positivo, lista de propiedad*

Nota: Los puntos suspensivos no forman parte de la sintaxis de la función, se utilizan para indicar un número variable de parámetros de ese tipo.

3.2.1 Funciones de Evaluación

EVAL El resultado de la evaluación del argumento es a su vez evaluado, y éste es el resultado de la función EVAL.

Sintaxis:

(EVAL *expresión*)

Ejemplo: Supóngase que A tiene como valor B y que a su vez B tiene como valor C, entonces

```
( eval a )
   c
```

', QUOTE Inhibe la evaluación del argumento, es decir, regresa como valor la expresión-s que se le da como argumento.

Sintaxis:

(QUOTE *expresión*)

' *expresión*

Ejemplo:

```
(quote (a b c))
   (a b c)
'(a b c)
   (a b c)
```

3.2.2 Funciones de Asignación

PSETQ Calcula los valores de las expresiones-s y luego hace las asignaciones, a su correspondiente átomo.

Sintaxis:

(PSETQ $átomo_1$ $expresión_1$ ... $átomo_n$ $expresión_n$) $n \geq 1$

Ejemplo: Si **A** tiene como valor **M** y **B** tiene como valor **N** entonces

```
(psetq a b b a )
   n
a
   n
b
   m
```

SET Asigna el valor de la expresión-s al átomo simbólico obtenido al evaluar el parámetro correspondiente. Lo hace tantas veces como parejas de parámetros tenga.

Sintaxis:

(SET *átomo*$_1$ *expresión*$_1$ ... *átomo*$_n$ *expresión*$_n$) $n \geq 1$

Ejemplo: Si el valor de *A* es *B*

```
(set a 'valor)
   valor
a
Atomo sin valor asociado
   b
valor
```

SETQ Asigna el valor de la i-ésima expresión-s al i-ésimo átomo simbólico, hace esto tantas veces como parejas de parámetros tenga.

Sintaxis:

(SETQ *átomo*$_1$ *expresión*$_1$ ... *átomo*$_n$ *expresión*$_n$) $n \geq 1$

Ejemplo: Si el valor de *A* es *B*

```
(setq a 'valor)
   valor
a
    valor
b
    Atomo sin valor asociado
```

3.2.3 Funciones Aritméticas

ABS Obtiene el valor absoluto del valor de su argumento.

Sintaxis:

```
( ABS  numérico )
```

Ejemplo:

```
(abs -78)
   78
(abs 9.5)
   9.5
```

COS Calcula el coseno del valor del paramétro. (Este valor debe estar dado en radianes).

Sintaxis:

```
( COS  real )
```

Ejemplo:

```
(cos pi)
   1.00
(Nota : Existe el átomo simbólico PI cuyo valor es π)
```

- , DIFFERENCE Resta el valor de las expresiones dadas como argumentos, éstos pueden ser reales o enteros, en caso de que alguno sea real, el resultado es real. Si sólo se tiene un argumento, el resultado es el inverso aditivo del valor del mismo.

Sintaxis:

```
( -  numérico1   numérico2 )
( -  numérico  )
( DIFFERENCE  numérico1 numérico2  )
( DIFERRENCE  numérico )
```

Ejemplos:

```
(- 8)
   -8
(- 4.75 4)
   0.75
```

/, DIVIDE Divide el valor de las dos expresiones dadas como argumento, éstos pueden ser reales o enteros, el resultado es siempre real. Si sólo se tiene un argumento, entonces regresa el inverso multiplicativo del valor del mismo.

Sintaxis:

```
( /  numérico1 numérico2 )
( /  numérico )
( DIVIDE  numérico1 numérico2 )
( DIVIDE  numérico )
```

Ejemplos:

```
(/ 14 4)
  3.0
(/ 2)
  0.5
```

EXP Calcula el valor del número e a la potencia indicada como valor de su argumento.

Sintaxis:

```
( EXP entero )        entero ≥ 0
```

Ejemplos:

```
(exp 0)
 1.0
(exp 1)
 2.7182818
```

EXPT Eleva el valor del primer argumento a la potencia indicada como resultado de la evaluación del segundo.

Sintaxis:

```
( EXPT   numérico entero )
```

Ejemplo:

```
(expt 2 3)
   8
```

FLOAT Convierte el valor del argumento en átomo real.

Sintaxis:

(FLOAT *entero*)

Ejemplo:

```
(float 14)
```
14.0

MAX Obtiene el número que es el mayor de los valores de los elementos numéricos dados como parámetros.

Sintaxis:

(MAX $numérico_1$ ... $numérico_n$) $n \geq 2$

Ejemplo:

```
(max 2 -3.1 4)
```
4

MIN Obtiene el número que es el menor de los valores de los elementos numéricos dados como parámetros.

Sintaxis:

(MIN $numérico_1$ ... $numérico_n$) $n \geq 2$

Ejemplo:

```
(min -2.5 3 4)
```
-2.5

+ , PLUS Suma el valor de las expresiones dadas como argumento, estos valores pueden ser reales o enteros, en caso de que alguno sea real, el resultado es real.

Sintaxis:

(+ $numérico_1$ ... $numérico_n$) $n \geq 2$

(PLUS $numérico_1$ ... $numérico_n$) $n \geq 2$

Ejemplo:

```
(+ 4.75 9 2)
```
15.75

REM Calcula el residuo de la división entera del valor del primer argumento entre el del segundo.

Sintaxis:

(REM *numérico*$_1$ *numérico*$_n$)

Ejemplo:

```
(rem 14 4)
   2
```

ROUND Convierte el resultado de la evaluación de su argumento en un átomo entero.

Sintaxis:

(ROUND *real*)

Ejemplo:

```
(round 4.6)
   5
(round 3.5)
   4
```

SIN Calcula el seno del valor del argumento. (Este valor debe estar dado en radianes).

Sintaxis:

(SIN *real*)

Ejemplo:

```
(sin (/ pi 2))
   1.00
```

(Nota: Existe el átomo simbólico PI cuyo valor es π)

SQRT Calcula la raíz cuadrada del valor del argumento.

Sintaxis:

(SQRT *real*) *real* $\geq$ 0

Ejemplo:

```
(sqrt 9)
   3
```

***, TIMES** Multiplica el valor de las expresiones dadas como argumento, estos valores pueden ser reales o enteros, en caso de que algún elemento sea real, el resultado es real.

Sintaxis:

(* $numérico_1$ ... $numérico_n$) $n \geq 2$

(TIMES $numérico_1$ ... $numérico_n$) $n \geq 2$

Ejemplo:

```
(* 9 3 10)
```

270

TRUNCATE Realiza la división entera del valor del primer argumento entre el del segundo, pero truncando, no redondeando

Sintaxis:

(TRUNCATE $numérico_1$ $numérico_n$)

Ejemplo:

```
(truncate 14 4)
```

3

3.2.4 Funciones para manejo de listas

APPEND Une en una lista, los elementos de las listas obtenidas al evaluar sus parámetros.

Sintaxis:

(APPEND $lista_1$ ... $lista_n$) $n \geq 2$

Ejemplo:

```
(append '((a) (b)) '((c) (d)) '())
```

((a) (b) (c) (d))

ASSOC Busca en la lista de asociación obtenida al evaluar su segundo parámetro, la sublista cuya llave es el valor del primer paámetro. Regresa la sublista (llave, valor) o bien *nil* si no lo encuentra.

Sintaxis:

```
( ASSOC   expresión1   expresión 2 )
```

Ejemplos:

```
(setq manzana '((color rojo) (clase fruta)))
    ((color rojo) (clase fruta))
(assoc 'color manzana)
    (color rojo)
(assoc 'tamano manzana)
    nil
```

CAR Regresa el primer elemento de la lista obtenida al evaluar su argumento. En caso de que el argumento sea *nil* regresa *nil*.

Sintaxis:

```
( CAR   lista )
```

Ejemplos:

```
(car '(a b c))
    a
(car nil)
    nil
```

CDR Regresa la lista que contiene todos los elementos de la evaluación de su argumento excepto el primero. Si el valor del argumento es *nil* o bien una lista con un solo elemento, el valor que regresa esta función es *nil*.

Sintaxis:

```
( CDR  lista )
```

Ejemplos:

```
(cdr '(a b c))
    (b c)
(cdr '(a))
    nil
```

CONS Agrega el valor del primer argumento al principio de la lista obtenida al evaluar el segundo.

Sintaxis:

(CONS *expresión lista*)

Ejemplos:

```
(cons 'a '(b c))
    (a b c)
(cons 'x nil)
    (x)
(cons '(b c) ('x y))
    ((bc) x y)
```

DELETE Elimina, en la lista obtenida al evaluar el segundo parámetero, las apariciones del valor del primer argumento. (Se considera una función destructiva, ya que modifica el valor del segundo elemento).

Sintaxis:

(DELETE *expresión lista*)

Ejemplo:

```
(setq ejem '(x y y x y))
    (x y y x y)
(delete 'x ejem)
    (y y y)
ejem
    (y y y)
```

GET El resultado de la evaluación del segundo argumento, es considerado una propiedad, que será buscada en la lista de propiedad obtenida al evaluar el primer argumento. El resultado de esta función es *nil* si no existe esa propiedad y el valor de la propiedad, en otro caso. de la propiedad.

Sintaxis:

(GET *lista expresión*)

Ejemplos:

```
(plist 'pelota)
    ((color azul) (forma redonda) (tamano grande))
(get 'pelota 'color)
    azul
(get 'pelota 'precio)
    nil
```

LAST Regresa la lista que contiene como único elemento el último de la lista resultante de la evaluación de su argumento.

Sintaxis:

(LAST *lista*)

Ejemplo:

```
(last '((a b) (c d)))
    ((c d))
```

LENGTH Regresa el átomo entero cuyo valor es el número de expresiones-s contenidas en el valor de su argumento.

Sintaxis:

(LENGTH *expresión*)

Ejemplos:

```
(length '((a b) (c d)))
    2
(length 'a)
    1
(length '(a b c d e))
    5
```

LIST Crea una lista cuyos elementos son los valores de los argumentos.

Sintaxis:

(LIST $expresión_1$ ... $expresión_n$) $n \geq 2$

Ejemplos:

```
(list '((a) (b)) '((c) (d)))
    (((a) (b)) ((c) (d)))
(list '(a b) '1)
    (a b 1)
```

NCONC Construye una lista con los elementos de las listas obtenidas al evaluar sus argumentos, sólo que con esta función, el valor de cada argumento es alterado.

Sintaxis:

(NCONC $lista_1$ ... $lista_n$) $n \geq 2$

Ejemplo:

```
(setq abc '(a b c))
    abc
(setq xyz '(x y z))
    (x y z)
(nconc abc xyz)
    (a b c x y z)
abc
    (a b c x y z)
xyz
    (x y z)
```

PLIST Despliega la lista de propiedad obtenida al evaluar su argumento.

Sintaxis:

(PLIST *lista de propiedad*)

Ejemplo:

```
(plist 'pelota)
    ((color azul) (forma redonda))
```

REMPROP Elimina de la lista de propiedad obtenida al evaluar el primer argumento, la propiedad obtenida al evaluar el segundo argumento. Regresa la pareja que fue removida.

Sintaxis:

(REMPROP *lista de propiedad expresión*)

Ejemplo:

```
(remprop 'pelota 'tamano)
```

(pelota tamano)

REVERSE Regresa la lista con los elementos de la lista obtenida en la evaluación de su argumento, sólo que, en orden inverso.

Sintaxis:

(REVERSE *lista*)

Ejemplos:

```
(reverse '(a b c))
```

(c b a)

```
(reverse '((a b) (c d)))
```

((c d) (a b))

RPLACA Reemplaza el primer elemento de la lista, por el valor del segundo argumento. (Se considera una función destructiva, ya que modifica el valor del primer argumento).

Sintaxis:

(RPLACA *lista expresión*)

Ejemplo:

```
(setq postre '(fresas con crema es un rico postre))
```

(fresas con crema es un rico postre)

```
(rplaca postre 'platanos)
```

(platanos con crema es un rico postre)

```
postre
```

(platanos con crema es un rico postre)

RPLACD Reemplaza a partir del segundo elemento de la lista, por el valor del primer argumento. (Se considera una función destructiva, ya que modifica el valor del segundo argumento).

Sintaxis:

(RPLACD *expresión lista*)

Ejemplo:

```
(setq ejemplo '(las-manzanas son buenas para ti))
    (las-manzanas son buenas para ti)
(rplacd ejemplo '(fueron malas para Adan))
    (las-manzanas fueron malas para Adan)
ejemplo
    (las-manzanas fueron malas para Adan)
```

SETF Asigna o reemplaza el valor de una propiedad o elemento de un arreglo. El primer argumento es una *forma de acceso* que al evaluarse produce el valor actual de la propiedad, y el valor del segundo argumento es el nuevo valor. Regresa el nuevo valor.

Sintaxis:

(SETF $expresión_1$ $expresión_2$)

Ejemplos:

```
(plist 'pelota)
    ((color rojo) (forma redonda))
(setf (get 'pelota 'precio) 800.50)
    800.50
(plist 'pelota)
    ((color rojo) (forma redonda) (precio 800.50))
(setf (get 'pelota 'color) amarillo)
    amarillo
(plist 'pelota)
    ((color amarillo) (forma redonda) (precio 800.50))
(setf (aref 'arr '(2 5)) 27)
    27 (Pone en la posición 2,5 del arreglo un 27 )
```

SUBST Sustituye las apariciones del valor del segundo argumento por el valor del primero, en la lista obtenida al evaluar el tercero.

Sintaxis:

(SUBST $expresión_1$ $expresión_2$ *lista*)

Ejemplos:

```
(subst 'a 'b '(a b c))
    (a a c)
(subst 'b 'y '(sqrt (+ (* x x) (* y y))))
    (sqrt (+ (* x x) (* b b)))
```

3.2.5 Predicados

= Regresa *t* si el valor del primer argumento y el valor del segundo son iguales y *nil* en caso contrario.

Sintaxis:

(= $numérico_1$ $numérico_2$)

Ejemplos:

```
(= 4 3)
    nil
(= 8 8)
    t
(= 1 1.0)
    t
```

> Regresa *t* si el valor de los argumentos están en orden ascendiente y *nil* en caso contrario.

Sintaxis:

($numérico_1$ ... $numérico_n$) $n \geq 2$

Ejemplos:

```
(> 2 5)
    nil
(> 7 3 -1)
    t
(> 2 2)
    nil
```

< Regresa t si los valores de los argumentos están en orden descendiente y *nil* en caso contrario.

Sintaxis:

(*numérico*$_1$ ... *numérico*$_n$) $n \geq 2$

Ejemplos:

```
(< 2 5)
    t
(< 7 3 -1)
    nil
(< 2 2)
    nil
```

AND Evalua sus argumentos de izquierda a derecha. Si se encuentra uno cuyo valor sea *nil* entonces éste se regresa y los otros argumentos ya no se evaluan. En otro caso regresa el valor del último argumento.

Sintaxis:

(AND *expresión*$_1$ ... *expresión*$_n$) $n \geq 2$

Ejemplos:

```
(and t t nil)
    nil
(setq frutas '(manzana pera lima platano))
    (manzana pera lima platano)
(and (member 'manzana frutas) (member 'lima frutas))
    (lima platano)
(and (member 'melon frutas) (member 'lima frutas))
    nil
```

ATOM Regresa *t* si el valor del argumento no es una lista y *nil* en caso contrario.

Sintaxis:

(ATOM *expresión*)

Ejemplos:

```
(atom '())
    t
(atom 8)
    t
(atom '(a b c))
    nil
```

EQL Regresa *t* si el valor del primer argumento ocupa la misma área de memoria que el valor del segundo y *nil* en caso contrario. (No trabaja con átomos numéricos).

Sintaxis:

(EQL *expresión*$_1$ *expresión*$_2$)

Ejemplos:

```
(setq l1 (list 'a 'b 'c))
    (a b c)
(setq l2 (list 'a 'b 'c))
    (a b c)
(setq l3 l2)
    (a b c)
(eql l1 l2)
    nil
(eql l2 l3)
    t
```

EQUAL Regresa *t* si el valor del primer argumento es igual al valor del segundo y regresa *nil* en caso contrario.

Sintaxis:

(EQUAL *expresión*$_1$ *expresión*$_2$)

Ejemplos:

```
(setq l1 (list 'a 'b 'c))
    (a b c)
(setq l2 (list 'a 'b 'c))
    (a b c)
(equal l1 l2)
    t
(equal l1 '( a b c))
    t
```

EVENP Regresa *t* si el argumento tiene como valor un número par y *nil* en caso contrario.

Sintaxis:

(EVENP *entero*)

Ejemplos:

```
(evenp 475)
    nil
(evenp -84)
    t
```

FBOUNDP Regresa *t* si el valor de su argumento es el nombre de una función o de un macro.

Sintaxis:

(FBOUNDP *simbólico*)

Ejemplos:

```
(fboundp 'primero)
    t (si 'primero' es una función )
(fboundp 'nada)
    nil (si 'nada' no es una función )
```

LISTP Regresa *t* si el valor del argumento es una lista y *nil* en caso contrario.

Sintaxis:

```
( LISTP  expresión )
```

Ejemplos:

```
(listp nil)
    t
(listp 'a)
    nil
(listp '(a b c))
    t
```

MEMBER Busca el átomo obtenido como resultado de la evaluación del primer argumento en la lista obtenida al evaluar el segundo. Si no lo encuentra regresa *nil*, en caso de encontrarlo, regresa la lista que contiene los elementos a partir del buscado.

Sintaxis:

```
( MEMBER  átomo lista )
```

Ejemplos:

```
(setq digitos '(uno dos tres cuatro cinco seis siete ocho))
    (uno dos tres cuatro cinco seis siete ocho)
(member cinco digitos)
    ( cinco seis siete ocho)
(member diez digitos)
    nil
```

MINUSP Regresa *t* si el argumento tiene como valor un número negativo y *nil* en caso contrario

Sintaxis:

```
( MINUSP  numérico )
```

Ejemplos:

```
(minusp -4)
    t
(minusp 23)
    nil
```

NOT Regresa *t* si el argumento tiene valor igual a *nil* y en caso contrario, regresa *nil*.

Sintaxis:

```
( NOT  expresión )
```

Ejemplos:

```
(not nil)
    t
(not t)
    nil
(not 'perro)
    nil
```

NULL Regresa *t* si el valor del argumento es la lista vacía y *nil* en caso contrario

Sintaxis:

```
( NULL  expresión )
```

Ejemplos:

```
(null '())
    t
(null t)
    nil
(null '(a b))
    nil
```

NUMBERP Regresa *t* si el argumento tiene como valor un número y *nil* en caso contrario

Sintaxis:

```
( NUMBERP  expresión )
```

Ejemplos:

```
(numberp 'a)
    nil
(numberp 5788.9)
    t
```

OR Evalua sus argumentos de izquierda a derecha. Si se encuentra algo diferente de *nil* entonces éste se regresa y los demás argumentos ya no se evaluan. En otro caso regresa *nil*.

Sintaxis:

(OR *expresión*$_1$ ... *expresión*$_n$) $n \geq 2$

Ejemplos:

```
(or t t nil)
    t
(setq frutas '(manzana pera lima platano))
    (manzana pera lima platano)
(or (member 'manzana frutas) (member 'lima frutas))
    (manzana pera lima platano)
```

ZEROP Regresa t si el argumento tiene valor igual a cero y *nil* en caso contrario

Sintaxis:

(ZEROP *numérico*)

Ejemplos:

```
(setq cero 879)
    879
(zerop cero)
    nil
(zerop 0)
    t
```

3.2.6 Funciones para control de la secuencia

APPLY Aplica el procedimiento obtenido al evaluar el primer argumento, a los elementos de la evaluación de los siguientes argumentos pero tomando cada elemento de esta lista como un argumento del procedimiento.

Sintaxis:

(APPLY *simbólico expresión*$_1$ ... *expresión*$_n$) $n \geq 1$

Ejemplo:

```
(apply '* '(4 5 6 7))
```

COND Sirve para toma de decisiones. Sus argumentos son listas, llamadas claúsulas, el primero elemento de cada claúsula es una condición. Evalua la condición de cada lista hasta encontrar una cuyo valor sea diferente de *nil*. Entonces evalua los siguientes elementos de esa lista y regresa el valor del último elemento. Si ninguna claúsula tiene valor diferente de *nil* entonces regresa *nil*. Si la claúsula sólo consta de condición este es el valor que regresa.

Sintaxis:

(COND ($expresión_{11}$ $expresión_{1m}$)
⋮
($expresión_{n1}$ $expresión_{nm}$))

$n \geq 1,\ m \geq 1$

Ejemplos:

```
(setq l nil)
   nil
(cond ((null l) 'vacia)
      (t 'no-vacia))
   vacia
(setq l '(a b c))
   (a b c)
(cond ((null l) 'vacia)
      (t 'no-vacia))
 no-vacia
```

DO Sirve para hacer funciones que iteren explícitamente. La primera parte del DO consta de definición de variables locales con sus respectivos valores iniciales (si no se incluyen valores iniciales, entonces se asigna *nil*). La asignación se hace en paralelo. Si no hay parámetros esta lista debe quedar vacía. La segunda parte es la condición que determina cuando debe dejar de ejecutarse el ciclo y que valor debe regresar. Las formas que siguen a la condición son evaluadas y regresa el valor de la última. Se prueba la condición y si ésta tiene valor diferente de *nil* entonces ejecuta el cuerpo, en caso contrario, da por terminado el ciclo y verifica que exista expresión para regresar, si ésta no existe, el valor regresado es *nil*. El cuerpo consta de un conjunto de expresiones-s que se evaluan secuencialmente. Si el cuerpo esta vacío se ejecutan como cuerpo las formas de actualización. En caso de que haya tanto cuerpo como formas de actualización, se ejecuta el cuerpo y luego las formas de actualización, cada vez.

Nota: Todas las expresiones entre "<" y ">" son expresiones simbólicas que tienen ese papel en esta función.

Sintaxis:

```
( DO ((  < variable_1 > < v. inicial_1 > < actualización_1 > )
                        ⋮
         (  < variable_n > < v. inicial_n > < actualización_n > ))
     (  < condición > < resultado >  )
        < cuerpo > )
```

Ejemplos:

```
; Programa que calcula m^n en forma iterativa
(defun eleva (m n)
   (do ((resultado 1) ; Define e inicia resultado
        (exponente n)) ; Define e inicia exponente
       ((zerop exponente) resultado) ; Condicion
     (setq resultado (* m resultado)) ; Cuerpo
     (setq exponente (- exponente 1)))) ; Cuerpo
   eleva
(eleva 2 3)
   8
```

```
: Programa que calcula el factorial de un número
(defun factorial (n)
   (do ((resultado 1 (* n resultado))
        (n n (- n 1)))
       ((zerop n) resultado)))
   factorial
(factorial 3)
   6
```

En este segundo ejemplo, se da tanto valor inicial como forma de actualización para las variables locales, además ese DO no tiene cuerpo, explícito.

DO-S Tiene la misma sintaxis y operación que el DO, excepto que la asignación se realiza en forma secuencial. Ejemplo:

```
: Programa que calcula m" en forma iterativa
(defun eleva (m n)
   (do-s ((resultado m (* m resultado))
          (exponente n (- exponente 1))
          (contador (- exponente 1) (- exponente 1)))
         ((zerop contador) resultado)))
   eleva
(eleva 2 3)
   8
```

MAPCAR Sirve para ejecutar el procedimiento obtenido al evaluar su primer argumento, sobre cada elemento de la lista obtenida en la evaluación de su siguiente argumento.

Sintaxis:

```
( MAPCAR  simbólico lista )
```

Ejemplos:

```
(mapcar 'oddp '(1 2 3))
   (t nil t)
(mapcar '+ '(1 2 3) '(4 7 9))
   (5 7 9)
```

PROG1 Evalua todos los parámetros y regresa el valor del primero.

Sintaxis:

(PROG1 *expresión*$_1$ ... *expresión*$_n$) $n \geq 2$

Ejemplo:

```
(prog1 (setq a 'x) (+ 4 3 5) (setq c 'z) )
   x
```

PROGN Evalua todos los argumentos y regresa el valor del último.

Sintaxis:

(PROGN *expresión*$_1$ ... *expresión*$_n$) $n \geq 2$

Ejemplo:

```
(progn (setq a 'x) (+ 4 3 5) (setq c 'z) )
   z
```

3.2.7 Funciones para manejo de funciones definidas por el programador.

DEFMACRO Define un macro.El primer argumento es el nombre del macro, los que están entre paréntesis son los parámetros formales del macro, aunque esta lista puede estar vacía. Finalmente las expresiones simbólicas forman el cuerpo del macro. El valor que regresa esta función primitiva es el nombre del macro definido.

Sintaxis:

(DEFMACRO *simbólico* (*simbólico*$_0$ ... *simbólico*$_n$)
expresión$_1$ ... *expresión*$_m$
$n \geq 0,\ m \geq 0$

Ejemplo:

```
(defmacro primero (s)
    '(car ,s))
  primero
```

DEFUN Define una función cuyo nombre es el primer argumento y debe ser un átomo simbólico. Los parámetros formales de la función son los definidos entre paréntesis, de tal forma que si la lista está vacía se trata de la definición de una función sin parámetros. El cuerpo puede tener tantas expresiones-s como se requiera. Regresa como valor el nombre de la función definida.

Sintaxis:

(DEFUN *simbólico* (*simbólico*$_0$ ... *simbólico*$_n$)
expresión$_1$ ... *expresión*$_m$) $n \geq 0$ $m \geq 1$

Ejemplo:

```
(defun cambia (pareja)
 (list (cadr pareja) (car pareja)))
     cambia
(cambia '(tu yo))
     (yo tu)
```

FUNCALL Ejecuta la función que corresponde al valor del primer argumento, si ésta necesita parámetros se definen a continuación del primer argumento.

Sintaxis:

(FUNCALL *simbólico*$_1$ *simbólico*$_n$) $n \geq 1$

Ejemplos:

```
(setq interes 0.1)
     0.1
(defun calcula-interes (balance)
 (* balance interes))
     calcula-interes
(setq banco 'calcula-interes)
     calcula-interes
(funcall banco 100.0)
     10.0
(funcall 'calcula-interes 100.0)
     10.0
```

LAMBDA Define una función sin nombre, de tal forma que sólo se podrá emplear una vez y en el lugar que se defina. Los átomos simbólicos que aparecen entre paréntesis son los parámetros formales de esta función y las expresiones simbólicas son el cuerpo.

Sintaxis:

(LAMBDA ($simbólico_0$ ... $simbólico_n$)
$expresión_1$ ... $expresión_m$) $n \geq 0$, $m \geq 1$

Ejemplo:

```
((lambda (x) (equal (get x 'clase) 'fruta)) 'manzana)
    t
```

LET Declara variables locales, dándoles valor inicial. Esta asignación es hecha en paralelo, es decir, todos los valores son calculados antes de que se efectue la inicialización. Una vez terminada la asignación ejecuta el cuerpo, el cual puede tener la cantidad de expresiones-s que se necesite.

La sintaxis es como sigue: el primer argumento de LET es una lista que a su vez contiene listas de dos elementos, el primero es un átomo simbólico que representa el nombre de la variable local, y el segundo es una expresión simbólica que al evaluarse constituirá el valor inicial de esa variable. Las expresiones simbólicas que están después de la lista incial, forman el cuerpo del LET.

Sintaxis:

(LET (($simbólico_1$ $expresión_1$)
⋮
($simbólico_n$ $expresión_n$))
$expresión_1$ ... $expresión_m$) $n \geq 1$, $m \geq 1$

Ejemplo:

```
(defun union (uno dos)
   (let ((elem uno) (lista dos))
 (cond ((listp uno) (setq elem dos lista uno)))
 (cond ((member elem lista) lista )
       (t (cons elem lista)))))
  union
```

El procedimiento *union* está escrito de tal forma que sus parámetros: una lista y un átomo puedan ser dados en cualquier orden. La variable local *elem* contine al primer parámetro y *lista* contiene al segundo. Finalmente se determina cuál es cual y realiza la unión del elemento a la lista.

LET-S Declara variables locales, dándoles valor inicial. Esta asignación es hecha en forma secuencial. Una vez terminada la asignación ejecuta el cuerpo, el cual puede tener la cantidad de expresiones-s que se necesite.

La sintaxis es como sigue: el primer argumento es una lista que a su vez contiene listas de dos elementos, el primero es un átomo simbólico que representa el nombre de la variable local, y el segundo es una expresión simbólica que al evaluarse constituirá el valor inicial de esa variable. Las expresiones simbólicas que están después de la lista incial, forman el cuerpo del LET-S.

Sintaxis:

(LET-S ((*simbólico*$_1$ *expresión*$_1$)
⋮
(*v.local*$_n$ *v.inicial*$_n$))
expresión$_1$ ... *expresión*$_n$)

Ejemplo:

```
(defun union (uno dos)
   (let-s ((elem (cond ((listp uno) dos)
                       (t uno)))
           (lista (cond ((equal elem dos) uno)
                        (t dos))))
   (cond ((member elem lista) lista)
         (t (cons elem lista)))))
```

3.2.8 Funciones de Entrada/Salida

' El *backquote* o apóstrofe al revés, inhibe la evaluación de los átomos simbólicos que están en la lista que se obtine como valor de su argumento, excepto que cuando encuentra una coma habilita otra vez la evaluación. Si lo que encuentra es ,@ y el resultado es una lista, entonces elimina los paréntesis

Sintaxis:

' *lista*

Ejemplos:

```
(setq variable 'ejemplo)
    ejemplo
'(esto es un ,variable)
    (esto es un ejemplo)
(setq variable '(ejemplo mas dificil))
    (ejemplo mas dificil)
'(esto es un ,variable)
    (esto es un (ejemplo mas dificil))
'(esto es un ,@variable)
    (esto es un ejemplo mas dificil)
```

BLANKS Escribe tantos espacios en blanco como el valor indicado en su parámetro. Regresa *t*.

Sintaxis:

```
( BLANKS  e.positivo  )
```

CLOSE Cierra el archivo que se esté empleando para la escritura, y vuelve a dejar como archivo de salida la terminal. Regresa *t*.

Sintaxis:

```
(  CLOSE  )
```

Ejemplo:

```
(CLOSE '!textos.out!)
```

PRIN1 Evalua su argumento y lo escribe sin hacer salto de línea. El valor que regresa es `t`.

Sintaxis:

```
( PRIN1  expresión )
```

Ejemplo:

```
(prog1 (print 'hola) (princ 'amigos))
    hola amigos
 hola
```

PRINC Evalua su argumento y lo escribe en la misma línea, en caso de haber barras verticales, las suprime. El valor que regresa es `t`.

Sintaxis:

```
( PRINC  expresión )
```

Ejemplo:

```
(princ '|hola que tal|)
    hola que tal
```

PRINT Evalua su argumento y lo imprime en una nueva línea. El valor que regresa es `t`.

Sintaxis:

```
( PRINT  expresión )
```

Ejemplos:

```
(print hola)
    hola
(print '|que tal|)
   |  que tal  |
```

READ Lee una expresión-s del archivo de entrada, y la regresa como valor del READ.

Sintaxis:

```
( READ )
```

Ejemplo:

```
(setq leido (read))
Asigna a  leido  la expresión-s leída
```

READFILE Al evaluar su argumento se obtiene el nombre del nuevo archivo de entrada. Al terminar de leer el archivo, automáticamente vuelve a poner como archivo de entrada la terminal. Regresa *t*.

Sintaxis:

(READFILE *archivo*)

Ejemplo:

```
(readfile '|datos.lis|)
```

SYMBOL-NAME Crea una lista cuyos elementos son cada una de las letras del átomo simbólico obtenido al evaluar su argumento.

Sintaxis:

(SYMBOL-NAME *simbólico*)

Ejemplo:

```
(symbol-name 'hola)
```

(h o l a)

TERPRI Hace un salto de línea

Sintaxis:

(TERPRI)

WRITEFILE Al evaluar su parámetro se obtiene el nombre del nuevo archivo de salida.

Sintaxis:

(WRITEFILE *simbólico*)

3.2.9 Funciones para depuración

Ctrl D Interrumpe la acción de la función que se esté ejecutando y puede entrar en el ciclo de lectura, evaluación y escritura. Esta interrupción puede darse en el momento deseado, es decir, no es una instrucción dada en la función que se esté interpretando.

Ejemplo:

```
(defun eleva (m n)
   (do ((resultado 1)     ; Define e inicia resultado
        (exponente n))    ; Define e inicia exponente
       ((zerop exponente) resultado)          ; Condicion
        (setq resultado (* m resultado))      ; Cuerpo
        (setq exponente (- exponente 1))))    ; Cuerpo
   eleva
(eleva 2 100)
.... pasa el tiempo
^ D
 Quieres terminar? (S | N)
```

Si se teclea 'S' se termina de ejecutar la función *eleva*, en otro caso entra a un ciclo de lectura, evaluación y escritura, para indicar que es otro ciclo, el *prompt* cambia a '?'. Para continuar con la función *eleva* se da la función (bye).

BREAK

Imprime el valor de su argumento y entra en un ciclo de lectura, evaluación y escritura, en el cual el usuario puede darle expresiones de las que desee conocer o modificar su valor. Se sale de este ciclo al darle (bye).

Sintaxis:

(BREAK *expresión*)

Ejemplo:

```
(defun eleva (m n)
   (do ((resultado 1)    ; Define e inicializa  resultado
      (exponente n))     ; Define e inicializa  exponente
     ((zerop exponente) resultado)     ; Condicion
   (setq resultado (* m resultado))    ; Cuerpo
   (setq exponente (- exponente 1))    ; Cuerpo
   (break 'alto)))                     ; Cuerpo
  eleva
(eleva 2 3)
  (break: alto)
:    ....
```

En el ejemplo cada vez que asigna valor al *exponente* va a interrumpir la ejecución de la función *eleva* y entra a un ciclo de lectura, evaluación y escritura, para salir de él se da la función (bye). El *prompt* en este caso cambia a ":".

EDITOR

Llama al editor especificado como valor del primer parámetro de esta función, si el valor del siguiente parámetro es el nombre de un archivo, éste será el archivo a editar. También puede darse como segundo parámetro el nombre de una función que se desee editar. Al salir normalmente del editor, se regresa al intérprete.

Sintaxis:

(EDITOR *simbólico*$_1$ ... *simbólico*$_n$) $n \geq 1$

Ejemplos:

```
(editor 'pw 'eleva)
```

En este caso se llama al editor *perfect writer* con parámetro *eleva*, si no existe un archivo con ese nombre y el nombre pertenece a una función crea el archivo *eleva.lis* que contiene a la función *eleva*.

```
(editor 'pw 'eleva 'escribe 'pruebas)
```

En este caso se llama al editor *perfect writer* con parámetro *eleva*, si no existe un archivo con ese nombre y el nombre pertenece a una función crea un archivo con nombre *eleva.lis* y que contiene a las funciones *eleva*, *escribe* y *pruebas*.

```
(editor 'ws '!archivo.lis!)
```

En este caso se llama al editor *word-star* y con parámetro *archivo.lis* que es el nombre de un archivo el cual se editará.

STEP Sirve para seguir paso a paso el desarrollo de la función indicada como valor del parámetro. Deja de hacer esto al teclearle un espacio en blanco.

Sintaxis:

```
( STEP expresión )
```

Ejemplos:

```
(defun factorial (n)
   (cond ((zerop n) 1) (t (* n (factorial (- n 1))))))
 factorial
(step (factorial 2))
 (factorial 2)
   2
 (cond ((zerop n) 1) (t (* n (factorial (- n 1)))))
      (zerop n)
  n = 2
nil
(* n (factorial (- n 1)))
(- n 1)
n = 2
1
1
(cond ((zerop n) 1) (t (* n (factorial (- n 1)))))
(zerop n)
n = 1
nil
 (* n (factorial (- n 1)))
n = 1
.
.
.
```

TRACE

Prende la bandera de trazado de las funciones dadas como argumentos y regresa como valor *t*. Una vez prendida, al entrar a cada una de esas funciones imprime el nombre y el valor de cada uno de los parámetros; y al salir imprime el nombre de la función y los valores que regresa.

Sintaxis:

(TRACE $simbólico_1$ $simbólico_n$) $n \geq 1$

Ejemplo:

```
(defun 'factorial (n)
   (cond ((zerop n) 1)
         (t (* n (factorial (- n 1))))))
factorial
(trace factorial)
t
(factorial 5)

(Entrando a factorial (5))
 (Entrando a factorial (4))
  (Entrando a factorial (3))
   (Entrando a factorial (2))
    (Entrando a factorial (1))
     (Entrando a factorial (0))
    (Saliendo de factorial 1)
   (Saliendo de factorial 1)
  (Saliendo de factorial 2)
 (Saliendo de factorial 6)
(Saliendo de factorial 24)
(Saliendo de factorial 120)
120
```

UNTRACE

Deshabilita la acción de trace en las funciones especificadas, apagándoles la bandera de *trazado*. Regresa t.

Sintaxis:

(UNTRACE $simbólico_1$ $simbólico_n$) $n \geq 1$

3.2.10 Funciones para manejo de pantalla y graficación

CLEAR Limpia la pantalla y deja el cursor en la esquina superior izquierda. Regresa t.

Sintaxis:

(CLEAR)

COLOR Si se tiene monitor a color selecciona el color de acuerdo al valor de su parámetro. Regresa t. La selecci'on se realiza de acuerdo a los siguientes valores:

1 Azul
2 Magenta
3 Blanco

Sintaxis:

(COLOR *entero*)

GOTOXY

EL cursor es colocado en el renglón especificado como valor de su primer argumento y en la columna especificada como valor de su segundo argumento.

Sintaxis:

(GOTOXY $entero_1$ $entero_2$) $1 \leq entero_1 \leq 24$
$1 \leq entero_2 \leq 80$

LINE

En modo gráfico, traza en la terminal la línea que va de la primera pareja de ordenadas a la segunda. Cada pareja es el resultado de la evaluación de cada uno de los argumentos.($-159 \leq x \leq 159$ $-99 \leq y \leq 99$).

Sintaxis:

```
( LINE lista1 lista2 )
```

Ejemplo:

```
(mode 5)
(line '(-159 99) '(159 -99))
: dibuja en la pantalla una línea que la cruza, desde la
: esquina superior izquierda hasta la inferior derecha.
```

MODE

Selecciona el modo de operación de la pantalla de video de acuerdo a los siguientes valores:

0 Alfanumérico blanco y negro en 40 columnas y 25 renglones
1 Alfanumérico color en 40 columnas y 25 renglones
2 Alfanumérico blanco y negro en 80 columnas y 25 renglones
3 Alfanumérico color en 80 columnas y 24 renglones
4 Gráfico en color con 320 X 200 puntos
5 Gráfico en blanco y negro con 320 X 200 puntos
6 Gráfico en blanco y negro con 640 X 200 puntos

Sintaxis:

```
( MODE entero )
```

Nota: El modo de operación por omisión es el dos.

3.2.11 Funciones para interactuar con el sistema operativo

INP Lee un byte del puerto especificado como valor de su parámetro. El valor que regresa es el átomo simbólico cuyo valor es el byte que leyó.

Sintaxis:

```
( INP entero )        0 ≤ entero ≤ 65 535
```

INT86 Indica al procesador que tiene que atender la interrupción especificada en su parámetro. En la lista se especifica el valor inicial de cada uno de los registros del procesador, antes de la interrupción. Regresa una lista con el valor de cada uno de los registros del procesador, después de la interrupción.

Sintaxis:

```
( INT86 entero )
```

Ejemplo:

```
( int86 35 nil)
; Ocasiona una interrupción de C, en este caso el valor de
; los registros no importa.
```

Nota: Para una lista completa de las interrupciones del procesador 8086, consultar: Franklin [1984]

OUTP Manda el byte especificado como valor de su parámetro al puerto indicado.

Sintaxis:

```
( OUTP  byte puerto )          0 ≤ entero ≤ 65 535
```

3.2.12 Función para finalizar

BYE Con esta función se sale del ciclo de lectura, evaluación y escritura .

Sintaxis:

```
( BYE )
```

3.2.13 Función para recuperar espacio disponible

COLECTOR Por medio de esta función se hace una llamada explícita al colector de basura. Si se da un parámetro imprime su valor, y lo sigue haciendo cada vez que se active el colector de basura, esto hasta volver a llamar a esta función con valor de parámetro igual a *t*.

Sintaxis:

```
( COLECTOR  expresión )
```

3.2.14 Función para manejo de comentarios

COMMENT Inserta comentarios en un programa y los conserva, guardando cada palabra en la lista de objetos, esto ocasiona que disminuya el espacio para nuevos objetos, por lo tanto si tienen un porcentaje alto del texto total, llegan a causar problemas con la memoria. Regresa como valor *t*.

Sintaxis:

```
( COMMENT  texto )
```

Ejemplo:

```
(comment este es un ejemplo de comentarios
      usando la funcion comment)
  t
```

3.3 Funciones de Biblioteca

Además del conjunto de instrucciones primitivas descritas en las secciones anteriores MiLisp cuenta con una serie de funciones escritas en LISP y que se encuentran en el archivo LISP.LIB, en esta sección se describen estas rutinas usando el mismo formato que en las anteriores rutinas.

CAAR Obtiene el CAR del CAR del valor de su argumento

Sintaxis:

```
( CAAR  lista )
```

Ejemplo:

```
(caar '((a b) c d)
    a
```

CADR Obtiene el CAR del CDR del valor de su argumento

Sintaxis:

```
( CADR  lista )
```

Ejemplo:

```
(cadr '((a b) c d)
    c
```

CDAR Obtiene el CDR del CAR del valor de su argumento

Sintaxis:

```
( CDAR  lista )
```

Ejemplo:

```
(cdar '((a b) c d)
    (b)
```

CDDR Obtiene el CDR del CDR del valor de su argumento

Sintaxis:

```
( CDDR  lista )
```

Ejemplo:

```
(cddr '((a b) c d)
    (d)
```

CAAAR Obtiene el CAR del CAR del CAR del valor de su argumento

Sintaxis:

```
( CAAAR  lista )
```

Ejemplo:

```
(caaar '(((a b c)) (d e f) h i j)
a
```

CAADR Obtiene el CAR del CAR del CDR del valor de su argumento

Sintaxis:

```
( CAADR  lista )
```

Ejemplo:

```
(caadr '(((a b c)) (d e f) h i j)
d
```

CADAR Obtiene el CAR del CDR del CAR del valor de su argumento

Sintaxis:

```
( CADAR  lista )
```

Ejemplo:

```
(cadar '(((a b c)) (d e f) h i j)
    nil
```

CADDR Obtiene el CAR del CDR del CDR del valor de su argumento

Sintaxis:

```
( CADDR  lista )
```

Ejemplo:

```
(caddr '(((a b c)) (d e f) h i j)
    h
```

CDAAR Obtiene el CDR del CAR del CAR del valor de su argumento

Sintaxis:

```
( CDAAR  lista )
```

Ejemplo:

```
(cdaar '(((a b c)) (d e f) h i j)
    (b c)
```

CDADR Obtiene el CDR del CAR del CDR del valor de su argumento

Sintaxis:

```
( CDADR  lista )
```

Ejemplo:

```
(cdadr '(((a b c)) (d e f) h i j)
    (e f)
```

CDDAR Obtiene el CDR del CDR del CAR del valor de su argumento

Sintaxis:

```
( CDDAR  lista )
```

Ejemplo:

```
(cddar '(((a b c)) (d e f) h i j)
    nil
```

CDDDR Obtiene el CDR del CDR del CDR del valor de su argumento

Sintaxis:

```
( CDDDR  lista )
```

Ejemplo:

```
(cdddr '(((a b c)) (d e f) h i j)
    (i j)
```

ODDP Predicado que regresa t si el valor del argumento es un número impar y nil en otro caso.

Sintaxis:

(ODDP *numérico*)

Ejemplos:

```
(oddp 45)
    t
(oddp 70)
    nil
```

GCD Calcula el máximo común divisor del valor de sus parámetros.

Sintaxis:

(GCD $entero_1$ $entero_2$)

Ejemplos:

```
(gcd 3 7)
    1
(gcd (4 38)
    2
```

WRITE-A Escribe la matriz o arreglo que se le haya dado como parámetro.

Sintaxis:

(WRITE-A *matriz*)

Ejemplo:

```
(write-a 'arr)
    1 2 3
    2 4 5
```

PI

No es una función, es un átomo cuyo valor es el del número real π.

Sintaxis:

pi

Ejemplo:

```
pi
  3.1415
```

Referencias

Para una introducción informal a Lisp consultar Allen [1979]; Pratt [1979] hace una introducción al lenguaje resaltando a través de ejemplos sus características; Organick [1978] también introduce a Lisp por medio de ejemplos mencionando sólo las principales funciones. Una introducción a Common Lisp puede encontrarse en Bridger[1985] y en Winston[1985].

4
Diseño e implantación

En este capítulo se describen las consideraciones tomadas para diseñar y más tarde implantar el intérprete MiLisp también se muestran algunas rutinas en 'C' que se emplean en la implantación.

4.1 Estructuras de Almacenamiento

Para poder describir la forma en que trabaja el intérprete, es necesario definir las estructuras utilizadas para representar, en la memoria de la computadora, las expresiones-s en forma de listas ligadas.

Para construir las listas ligadas, es necesario disponer de nodos que constituyan los elementos de las lista, para esto se necesita un lugar donde almacenarlos. A este lugar se le denomina *heap*. Un *heap* es un bloque de almacenamiento en el cual las piezas se asignan y se liberan sin ningún orden determinado con anterioridad (a diferencia de una pila o una cola). Se usa esta estructura, pues durante la ejecución de un programa en LISP el espacio es asignado y liberado en puntos arbitrarios.

La arquitectura de la IBM 704 (máquina en que se realizó la primera implantación de Lisp) dejó marca permanente al conjunto de instrucciones del lenguaje. Los dos principales registros de la 704 se llamaban *"Address Register"* y *"Decrement Register"*, los cuales contenían la información necesaria para accesar la cabeza y la cola de la lista respectivamente. Así car y cdr originalmente significaban *"Contents of the Address Register"* y *"Contents of the Decrement Register"* respectivamente, aunque la 704 desde hace mucho tiempo es obsoleta, en todos los dialectos de LISP existen las funciones CAR y CDR las cuales obtienen el primer elemento de una lista y el resto de una lista, que son el corazón de LISP.

Cada elemento del *heap* contiene tres partes, la primera (llamada CAR) es un apuntador al primer elemento de la lista y la segunda (llamada CDR) es un apuntador al siguiente nodo de la lista. Cuando el CDR tiene como valor *nil* se esta indicando que la expresión-s terminó.

En una estructura así, no se puede distinguir la clase de expresión-s con que se está trabajando, así que a cada nodo se agrega un descriptor del dato (en este caso llamado bits) del cual se toman los bits necesarios para indicar el tipo de expresión-s de que se trata. Como sólo se necesita distinguir entre cuatro elementos diferentes, basta con tomar dos bits y en este caso se codificaron de la siguiente forma:

00	Lista
01	Atomo simbólico
10	Atomo numérico entero
11	Atomo numérico real

Se toman dos bits más del campo bits en el proceso de colectar basura, uno para distinguir los elementos no-basura de los basura y otro para distiguir los elementos activos de los inactivos, (se explicará a detalle en otra sección); se toma otro más para indicar que se hará la traza de una función y otro más para indicar que no se trabajará con una copia del valor.

Gráficamente cada nodo se vería como en la figura 1 y a su vez bits se desgloza como se muestra en la figura 2.

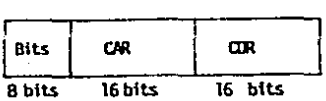


Figura 1. Nodo el *heap*.

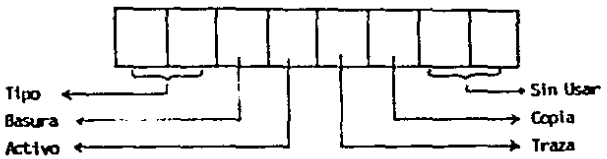


Figura 2. Campo bits.

Además del *heap* es necesario tener un lugar en donde almacenar cada elemento del lenguaje, pues tanto el car como el cdr son apuntadores. Este lugar es una tabla llamada ob-list, en la cual se encuentran todos los datos relacionados con los átomos simbólicos.

Cada átomo simbólico tiene asociado su nombre, y además puede tener un valor, el cuerpo de una función o bien una lista de propiedad, estas propiedades se encuentran en la tabla de objetos, por lo tanto, cada elemento de ella tiene el formato mostrado en la figura 3.

Función	Valor	Lista-p	Nombre

Figura 3. Elemento de la tabla de objetos

Los átomos simbólicos se almacenan de tal forma que dos nombres iguales no tienen diferentes entradas en la tabla de objetos.

Los átomos numéricos no tienen ninguna propiedad asociada, el valor de él le sirve también como nombre, así que no hay una tabla aparte para almacenar los números, la forma en que se implantaron es dandole al *CAR* un sentido múltiple, uno de éstos, como se mencionó es un apuntador al nodo cabeza de lista, otro es como apuntador a la lista de objetos y otro más es el de almacenador de valores numéricos.

El manejo del contenido del *CAR* depende del valor de *BITS*, el cual indica como interpretar su valor.

Además del *heap* y la lista de objetos, se cuenta con una pila (*stack*), en la que se va guardando el medio ambiente de operación al momento de ejecutar cualquier función o macro definida por el programador. Esta pila tiene un campo para el nombre del parámetro y otro para guardar su valor actual. Pues al salir de la rutina debe conservar el valor que tenía antes de entrar a ella.

En este punto se han definido las estructuras de datos usadas en Milisp, a continuación se muestra la versión de ellas en 'C'.

```
/* HEAP es una serie de estructuras con tres partes cada una,
 * BITS, CAR y CDR. Con estas tercias se forman las estructuras
 * internas de las expresiones dadas por el usuario
 */
struct h {
   char bits;
   union {
       struct h * dpar ;
       unsigned direccion :
       int      entero;
       float    real;
   } car;
   struct h * cdr;
} heap [LONG_HEAP];
```

El campo **bits** ocupa un byte de memoria y por medio de máscaras se accesan los bits mencionados en el diseño.

El campo car es una *union*, esto quiere decir que sus elementos pueden tomar alguno de los tipos definidos en él, éstos son o un apuntador a otro elemento del *heap* (usado en el caso de cabezas de listas), un entero sin signo que sirve como apuntador a la tabla de objetos, un número entero con signo que sirve para representar átomos enteros, o bien un número real utilizado para representar átomos numéricos reales.

El cdr es un apuntador a elementos del *heap*, pues siempre contiene la dirección del siguiente elemento en la lista, el cual es un elemento del *heap*

```
/*
 * En la estructura OB_LIST, se conservan el nombre y valor(es)
 * de cada objeto (atomo simbolico).
 */
struct ob {
   union {
         PTRH (*fun)();
         PTRH func;
   } funcion;
   PTRH valor;
   PTRH lista_p;
   char *nombre;
}ob_list [LONG_OB_LIST];
```

El campo *función* es un apuntador a una función que regresa un apuntador al *heap*, porque aquí se guarda la dirección de la función primitiva que representa ese átomo simbólico, o bien es un apuntador a la cabeza de la función o macro definida por el programador.

El campo *valor*, contiene un apuntador a la expresión interna que representa el valor actual de ese átomo simbólico. El campo *lista_p*, contiene un apuntador al primer nodo de la representación interna de la lista de propiedad asociada con ese átomo simbólico. El campo *nombre*, tiene la dirección de la cuerda de caracteres que contiene el nombre del átomo simbólico.

```
/*
 * En la estructura STACK, se conservan los valores de los
 * parámetros de las funciones definidas por el programador
 */
struct s {
   unsigned parametro;
   PTRH     valor;
} stack [LONG_STACK];
```

El campo parametro es un entero sin signo, pues es un apuntador al lugar en la tabla de objetos en donde se encuentran el nombre y todas las propiedades del parámetro formal, y el campo valor tiene el valor del parámetro formal en el momento de entrar a realizar la función.

4.2 Módulos del Sistema

Dado que los programas y los datos tienen la misma estructura sintáctica, se necesitan sólo tres módulos básicos para hacer una implementación de LISP en un intérprete. Estos módulos básicos son uno para lectura, otro para evaluación y uno más para escritura de resultados.

El módulo de lectura acepta una expresión-s de la terminal (o algún archivo) y la representa como una lista ligada que se llamará expresión interna, ésta se envía al módulo de evaluación y ahí es evaluada de acuerdo a la convención de LISP que el primer elemento de una lista especifica una función y el resto de los elementos de esa lista son los argumentos de la misma. Este módulo regresa otra expresión interna, que es tomada por el módulo de escritura. En el módulo de escritura la transforma en una expresión-s la cual ya puede ser escrita en la terminal (o un archivo). Para indicar que el intérprete ya se está listo para iniciar este ciclo de lectura, evaluación y escritura, en la pantalla se puede ver el caracter ">".

Esto en 'C' tiene la siguiente forma:

```
lee_evalua_escribe (){
PTRH raiz, par ;
unsigned sangria = 5 ;
   raiz = t ;                    /* Solo para que entre al ciclo */
   fprintf(foutput," %c ",prompt);
   while (raiz != fin) {
      raiz = lee ();
      raiz = evalua (raiz);
      pinta(raiz, sangria);
   }
}
```

4.2.1 Módulo de Inicialización

El módulo de inicialización es el encargado de preparar las condiciones para que se pueda comenzar el ciclo de lectura evaluación y escritura.

Lo primero que se hace en este módulo es inhibir la función *Ctrl C*, con la cual en el sistema operativo MS-DOS, se puede concluir la ejecución de cualquier programa en el momento que desee el usuario, la inhibición es para que la única forma de terminar la ejecución del intérprete sea por medio de la función que para ese fin, proporciona el intérprete. Una vez hecho esto, se asignan valores iniciales a las variables globales que son la variable para seguimiento paso a paso y los apuntadores a archivos, en este momento ambos apuntan a la terminal.

Luego se asigna cero al campo bit, todos los car del *heap* apuntan a nil. Para indicar qué elementos están disponibles en el *heap* se utiliza el cdr, en la inicialización los cdr apuntan al siguiente elemento en el *heap*, y el último elemento apunta a nil.

En la tabla de objetos inicialmente el campo *nombre* apunta a la cuerda vacía, y los demás campos tienen como valor *nil*.

No es necesario dar valores iniciales a los campos de la pila, pues basta con que el apuntador al tope de la misma tenga como valor la dirección de inicio de la pila.

Inicializa los apuntadores FIN y ERRORFORM. FIN es el valor que regresa la función "bye", esto para no ocupar un lugar en la tabla de objetos. ERRORFORM es el valor que regresa cualquier función primitiva, en caso de que ocurra algún error, sin importar el tipo de error de que se trate.

Como sólo existen dos átomos con valor previamente definido, éstos ocupan los dos primeros nodos del *heap*. Además de guardarlos en la tabla de objetos y darles valor inicial.

Ejecuta una rutina en la cual se ponen en la tabla de objetos, los nombres de las fuciones primitivas y la dirección física del código que las implanta.

Lee el archivo LISP.LIB, el cual contiene las funciones de biblioteca (escritas en LISP). Mientras se está ejecutando el módulo de iniciación en la pantalla se puede leer el mensaje "*Cargando el Sistema...*".

Al terminar de ejecutarse este módulo se limpia la pantalla, el cursor se posiciona en la parte superior izquierda y se despliega el *prompt* en este caso es ">", con el cual se indica al usuario que puede empezar a trabajar.

4.2.2 Módulo de Lectura

En el módulo de lectura, se lee una expresión-s de la terminal (o un archivo) y se construye su representación interna, asignando nodos disponibles del *heap* y cuando sea necesario, guardando los átomos en la lista de objetos.

El módulo de lectura está constituido de varias rutinas entre las principales se encuentran:*scanner, parser, hash.*

El *scanner o analizador léxico* es el encargado de leer caracter por caracter de la terminal y traducirlo en una serie de unidades llamadas *tokens*. Un *token* es un elemento de la grámatica que define el lenguaje LISP. Como ejemplos de *tokens* se tienen: lista, átomo simbólico, átomo numérico.

Para hacer un *scanner* es necesario describir los *tokens* que se pueden encontrar, esto se hizo usando notación BNF. Después se necesita un mecanismo que los reconozca. El utilizado en este intérprete es un autómata finito. Además de la localización de los *tokens* de entrada, se necesitan algunos mecanismos para realizar determinadas acciones con los *tokens* reconocidos, por ejemplo, guardarlo, desecharlo o producir un mensaje.

Los *tokens* reconocidos son los siguientes:

< átomo simbólico> ::= <letra> { <letra> | <dígito> | - }$_0^{30}$ |
1 {<caracter ascii imprimible>}$_0^{30}$ | <caracter especial>
< caracter especial> ::= ' | + | − | * | / | , | .Q | = | < | > | ' |
| <barra vertical>
< átomo entero> ::= <signo> <número>
< átomo real> ::= <signo> <número>.<número> |
<signo>.<número> E <entero>
< signo> ::= + | - | |
< número> ::= {<dígito> }$_0^{\infty}$
< dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< letra> ::= *A* | *B* | *C* | ... |*Z* | *a* | *b* | .... | *z*
< fin de archivo > ::= EOF
<comentario> ::= ;
<separador> ::= <cr> | <tab> | <espacio en blanco>
< paréntesis izquierdo > ::= (
< paréntesis derecho > ::=)
< apóstrofe > ::= '
< apóstrofe al revés (backquote)> ::= ‘
< inválido> ::= todo no caiga en estas categorías

El autómata que reconoce los *tokens* se muestra en la próxima página, su funcionamiento se explica a continuación.

1. Empezar en el estado inicial.
2. Leer un caracter
3. Examinar el caracter, y de acuerdo al tipo del mismo ir al siguiente estado.
4. Mientras el estado sea diferente del final y haya caracteres:
   4.1 Leer un caracter.
   4.2 Ir al siguiente estado, de acuerdo al caracter leido.
5. Detectar la clase de elemento que se ha reconocido
6. Construir la pareja *(elemento,clase)* para poder tomar la acción adecuada.

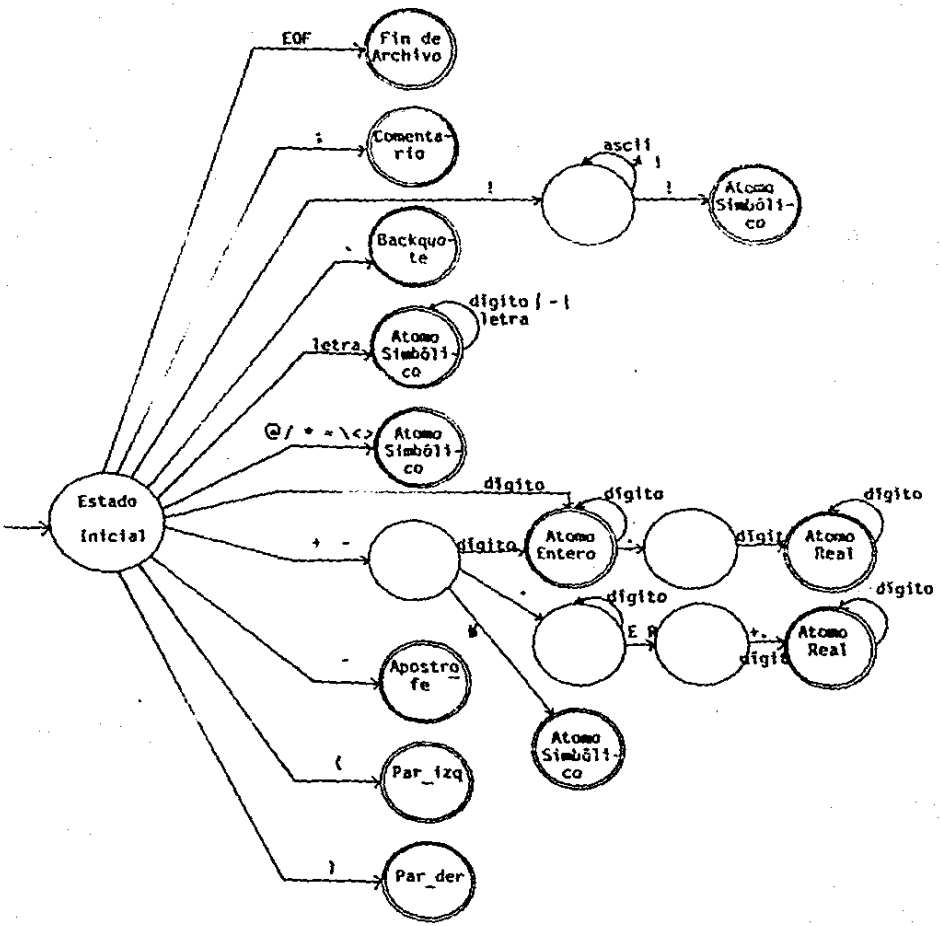


Figura 4. Autómata reconocedor de Lisp

El *parser* es el encargado de transformar la expresión-s proporcionada por el usuario, a su representación interna en forma de lista ligada, tomando elementos del *heap*.

Si se trata de un átomo simbólico, toma un nodo del *heap*, le pone como tipo átomo simbólico y en el campo de dirección pone el valor obtenido al aplicar al átomo simbólico la función de dispersión, supóngase que sea f. Además en la tabla de objetos en la i-ésima posición en el campo nombre se asigna un apuntador a la cuerda de caracteres que termina con carácter nulo y que es de menos de treinta y dos caracteres. El acceso a ese campo es sólo a través de las funciones de entrada/salida.

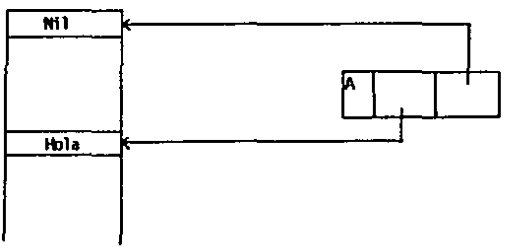


Figura 5. Ejemplo de átomo simbólico

Si se trata de un átomo numérico entero, toma un nodo de la tabla de espacio libre, le pone tipo átomo entero, y en el campo entero del CAR almacena el número entero y en el CDR pone nil. En el caso de átomo numérico real hace lo mismo excepto que el valor lo almacena en el campo real del CAR.

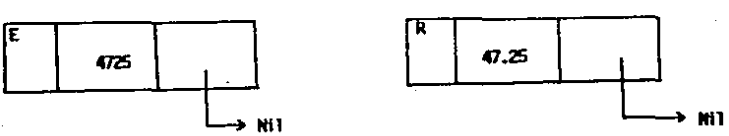


Figura 6. Ejemplo de átomos numéricos

En caso de un paréntesis izquierdo se ejecuta el siguiente algoritmo:

1. Marca el nodo como cabeza de lista
2. Toma un nuevo nodo del *heap* y lo liga al anterior por el CAR.
3. Analiza el siguiente elemento de la lista
4. Mientras el elemento sea diferente de paréntesis derecho:
   Si el elemento es un átomo entonces
   crea su representación interna de acuerdo a los procedimientos mencionados en párrafos anteriores.
   En caso contrario, ejecuta recursivamente este algoritmo.
   Analiza el siguiente elemento de la lista
   Si es diferente de paréntesis derecho entonces
   toma otro nodo del *heap* y lo liga al anterior por el CDR.
5. Asigna *nil* al CDR del nodo.

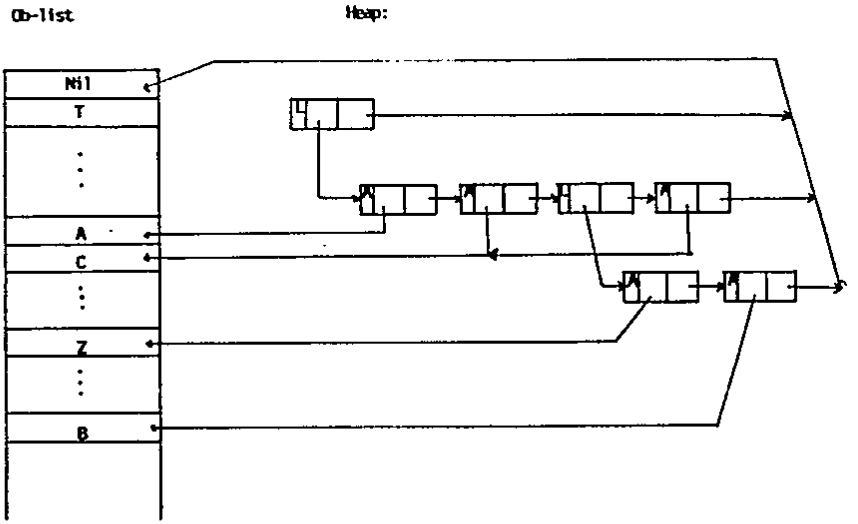


Figura 7. Representación de la lista (A C (Z B) C)

Si se tiene un apóstrofe, es equivalente al caso de la lista, pero el primer elemento de ésta es la función quote, ya que, 'cualquier-cosa es lo mismo que la siguiente lista (quote cualquier-cosa), como se puede apreciar en la figura 8. Lo mismo sucede con el apóstrofe al revés que es una abreviatura de *backquote*.

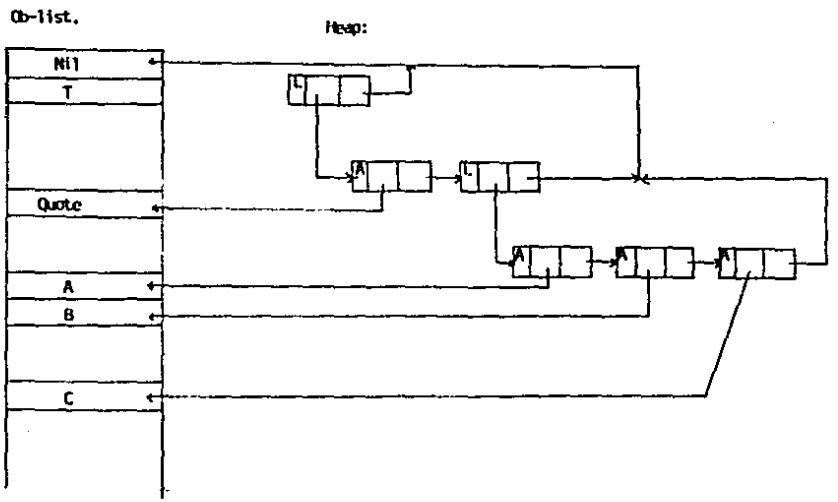


Figura 8. Representación de la lista '(A B C)

El algoritmo empleado para almacenar los átomos simbólicos en la tabla de objetos, es el llamado "Direccionamiento Abierto con doble función de dispersión", propuesto por Knuth.

Este algoritmo emplea dos funciones de dispersión, llamadas h1 y h2. La función h1 debe producir valores entre 0 y MAXHSH-1, inclusive (MAXHSH es el tamaño de la lista de objetos). La función h2 debe producir valores entre 1 y MAXHSH-1, que sean primos relativos con MAXHSH.

En el caso de MiLisp la función de dispersión h1, consiste en multiplicar los primeros seis caracteres del átomo simbólico por potencias de dos sumarlos y finalmente obtener el residuo por medio de la función *mod (MAXHSH - 1)*. MAXHSH es potencia de dos y es el tamaño de la tabla de objetos.

La función de dispersión h2 es similar a la anterior excepto que sólo que se toman en cuenta los primeros cuatro caracteres del átomo simbólico y además se le prende el bit menos significativo para que siempre sea un número impar, y con esto cumplir la condición de que sea primo relativo con MAXHSH. Esto es con el fin de evitar caer en iteraciones infinitas si el valor de h1 es múltiplo o submúltiplo del valor de h2.

El algoritmo empleado para almacenar los átomos simbólicos en la tabla de objetos, es el llamado "Direccionamiento Abierto con doble función de dispersión", propuesto por Knuth.

Este algoritmo emplea dos funciones de dispersión, llamadas h1 y h2. La función h1 debe producir valores entre 0 y MAXHSH-1, inclusive (MAXHSH es el tamaño de la lista de objetos). La función h2 debe producir valores entre 1 y MAXHSH-1, que sean primos relativos con MAXHSH.

En el caso de MiLisp la función de dispersión h1, consiste en multiplicar los primeros seis caracteres del átomo simbólico por potencias de dos sumarlos y finalmente obtener el residuo por medio de la función *mod (MAXHSH - 1)*. MAXHSH es potencia de dos y es el tamaño de la tabla de objetos.

La función de dispersión h2 es similar a la anterior excepto que sólo que se toman en cuenta los primeros cuatro caracteres del átomo simbólico y además se le prende el bit menos significativo para que siempre sea un número impar, y con esto cumplir la condición de que sea primo relativo con MAXHSH. Esto es con el fin de evitar caer en iteraciones infinitas si el valor de h1 es múltiplo o submúltiplo del valor de h2.

En la descripción del algoritmo se utilizan las siguientes variables:

MAXHSH = tamaño de la tabla,
K = elemento que se desea insertar o buscar
i = índice para buscar en la tabla
c = tamaño de las particiones en la tabla

El algoritmo empleado es el siguiente

1. [Aplica la primera función]
   $i \leftarrow h_1(k)$

2. [Prueba]
   Si tabla[i] está vacía entonces
      tabla[i] = K y termina
   de otra manera
      Si tabla[i] = K entonces termina

3. [Aplica la segunda función]
   $c \leftarrow h_2(k)$

4. [Busca un lugar vacío en la tabla]
   Mientras tabla[i] sea diferente de K
      $i \leftarrow i - c$
      Si $i < 0$ entonces $i \leftarrow i +$ MAXHSH.
      Si tabla[i] está vacía entonces
         tabla[i] = K

Se emplea este algoritmo, pues no requiere de espacio adicional para ligar los elementos. En este algoritmo la supresión de algún elemento es complicado, pero en el intérprete no está permitido dar de baja átomos simbólicos.

4.2.3 Módulo de Evaluación

El módulo de evaluación es el corazón del intérprete ya que contiene las rutinas que implementan las funciones permitidas en él así como rutinas adicionales que ayudan a interpretar las funciones primitivas.

La principal rutina se llama evalua, y es la encargada de la evaluación de la expresión interna recibida del *parser*, de acuerdo a la convención de que el valor de un átomo numérico es el mismo, el valor de un átomo simbólico es el asociado con él en la tabla de objetos, el de un átomo numérico es el mismo y el valor de una lista se determina aplicando la función especificada por el *car* de la lista a los argumentos que forman el *cdr* de la misma.

En el intérprete de Milisp, existen tres tipos de funciones, las primitivas, las definidas por el usuario y los macros. Para diferenciarlas, al definir una función o un macro se le pone un nodo al principio de la definición en el que se especifica la clase de función de que se trata.

Al evaluar las funciones con que el usuario puede definir procedimientos o macros se verifica que la función (o macro) a definir tenga el formato especificado, en particular, que la lista de parámetros exista y ya sea que tenga sólo átomos simbólicos o que este vacía. En la tabla de objetos, en el campo *función* pone un apuntador al inicio de la definición de función (o el macro). Este inicio tiene un nodo que puede ser EXPR, en caso de una función o MACRO en caso de un macro, seguido de la lista de parámetros y el cuerpo.

Cuando la rutina evalua tiene que evaluar una lista, toma el primer elemento para accesar la tabla de objetos y verificar que el campo *función* tiene valor, en caso de no tenerlo envía un mensaje de indicando que no existe una función con ese nombre. En caso de tener valor, éste puede ser EXPR, el cual indica que se trata de una función definida por el programador; MACRO que indica que se trata de un macro; o bien puede tratarse de una función primitiva.

Para evaluar las funciones primitivas se hace una transferencia inmediata a la rutina que implementa dicha función primitiva. En ella se evaluan los parámetros

actuales y se verifica el tipo de los valores de los parámetros sólo en caso de que se espere que pertenezca a una clase particular de expresión-s.

Todas las rutinas que implementan las funciones primitivas deben regresar un valor, es por eso que todas tienen tipo *apuntador a un elemento del heap.* En las que se esperan parámetros, se tiene que validar el tipo de los mismos. En caso de que alguno de ellos no satisfaga el tipo esperado se ejecuta la rutina que maneja los errores (llamada rut-err), que es la encargada de enviar, a la terminal, el mensaje de acuerdo al error ocurrido y regresar un apuntador llamado ERRORFORM para indicar, al intérprete que se produjo un error.

En todas las funciones primitivas, al evaluar los parámetros se pregunta por el apuntador ERRORFORM, y en caso de que sea afirmativa la respuesta, termina esa rutina, enviándolo como resultado. Así sucesivamente hasta acabar de ejecutar la función originalmente dada por el usuario.

Como ejemplo supóngase que el intérprete tiene que evaluar la siguiente expresión-s.

```
(cons (car '(a b c)) (cdr x))
```

Los pasos que sigue el intérprete para evaluarla son:

1. Entra a CONS y se da cuenta que necesita dos parámetros.
2. Trata de evaluar *(car '(a b c))* que es el primer parámetro de CONS.
3. En CAR evalua su parámetro que es *'(a b c)*
4. La función QUOTE regresa *(a b c)*
5. El CAR recibe *(a b c)* y regresa su primer elemento, en este caso *A*.
6. En CONS se trata de evaluar el segundo parámetro, en este caso *(cdr x)*.
7. En CDR trata de evaluar su parámetro, *x*. La rutina evalua detecta
   que no tiene valor, así que llama a RUT-ERR
8. RUT-ERR envía el mensaje *" X Atomo sin valor asociado"* y regresa el apuntador ERRORFORM.
9. El CDR recibe ERRORFORM y lo regresa inmediatamente.
10. CONS recibe ERRORFORM y lo regresa inmediatamente.
11. La función PINTA recibe ERRORFORM, y sólo escribe el *prompt.*

Los posibles errores que envía la rutina rut_err son los siguientes:

Caracter inválido
Función no implementada
Atomo sin valor asociado
Faltan argumentos
El archivo no puede abrirse
El argumento debe ser una lista
El argumento debe ser un átomo simbólico
El argumento debe ser un átomo entero
El argumento debe ser un átomo real
El argumento debe ser el nombre de un archivo ejecutable
El argumento no debe ser uno numérico
El argumento debe ser una función LAMBDA
El argumento debe ser una lista de propiedad
El argumento debe ser uno numérico

Además se tienen otros errores considerados como fatales pues ya no puede seguir trabajando MiLisp. Estos errores son los siguientes:

Ya no hay espacio para ningún nuevo elemento en el heap
Ya no hay espacio en memoria para un átomo simbólico más

Al ejecutar una función definida por el usuario se ejecuta la rutina llama-func en la cual los valores de los parámetros formales de la función se almacenan en la pila definida para ello. Se evaluan los argumentos actuales y se hace la correspondencia entre ellos, buscando la palabra optional, pues puede darse el caso que se tengan parámetros opcionales. En tal caso verifica si tiene valor inicial o debe asignárseles *nil*. Al terminar este paso evalua el cuerpo de la función, y finalmente se restauran los valores originales de los parámetros formales. Regresa como valor el obtenido al evaluar la última expresión-s del cuerpo.

Si se tiene una función lambda, se verifica que tenga el formato de una función, excepto que no debe tener nombre, y se ejecuta la función *llama-func* que se describió antes.

Si el primer elemento de la lista lleva a una expresión MACRO, se ejecuta la rutina expande-macro en la cual los parámetros actuales se asignan a los formales sin que se hayan evaluado previamente, y se almacenan en una pila. Una vez hecho esto, se evalua el cuerpo del macro, y la última expresión-s obtenida se vuelve a evaluar y éste es el resultado de la evaluación del macro.

En caso de que este habilitada la función para seguimientos paso a paso, la rutina evalua es la encargada de imprimir la expresión-s que será evaluada, así como el resultado de esta evaluación. Otra función extra que realiza evalua, también es para depuración y es la de trazado, en la cual se despliega el nombre y valor de los parámetros de la función definida por el programador que se ejecutará y al salir imprime el valor que regreso esta evaluación.

Las funciones de asignación alteran el campo *valor* que tiene el átomo simbólico en la tabla de objetos. En el caso de asignaciones en paralelo, antes de alterar la tabla de objetos, los valores que se van calculando se almacenan en una tabla auxiliar. Al terminar este proceso, se toman los valores de la tabla auxiliar, se hacen las asignaciones correspondiente, actualizando la tabla de objetos y se elimina la tabla auxiliar.

En las funciones para manejo de listas, se toman elementos del *heap* para construir las listas nuevas, o modificar las actuales. O bien se alteran algunos campos de los elementos tomados como parámetros, como en el caso de las funciones *append*, *car*, *last*, entre otras. Para no perder el valor original de los átomos simbólicos se trabaja siempre con copias de esos valores, a menos que la función no lo requiera.

La forma de implantar uso de variables locales, es en forma similar a la inicialización de parámetros. Se tiene una tabla en la que se guardan los valores actuales de los átomos, se asigna el valor inicial dado en la función, (si la asignación es en paralelo primero se calculan todos los valores iniciales y luego se asignan). Una vez hechas las inicializaciones se realiza el cuerpo de la función y finalmente se restauran los valores que pudieran tener las variables somónimas de las variables locales.

La función condicional se implementó con un ciclo que se realiza mientras haya claúsulas y el valor de la condición sea igual a *nil*. Si termina todas claúsulas y no

encuentra ninguna condición con valor igual a *nil* regresa *nil*, en caso contrario, evalua las expresiones-s que tiene a continuación de la condición con valor diferente de *nil* .

Para el ciclo iterativo, se asignan los valores iniciales a las variables definidas en él, y como antes, si se trata de asignación en paralelo, se van guardando los valores inicialies en una tabla temporal, hasta que se hayan calculado todos. A la vez se verifica si hay formas de actualización. Una vez que se han asignado los valores iniciales, en caso de haber formas de actualización, se construye el cuerpo agregando al final de las eaxpresiones-s que forman el cuerpo original, las formas de actualización y este será el cuerpo a evaluar. Una vez hecho esto, se entra a un ciclo en el cual se evalua la condición y si es diferente de t, se evaluan todas las expresiones-s del cuerpo, hasta que al evaluar la condición esta tenga valor igual a t, en cuyo, caso se evalua el resultado que regresará esta función.

Las funciones de entrada/salida trabajan todas con un apuntador a archivo que a menos que se cambie explícitamente, es la terminal. El archivo de lectura se cambia al interpretar la función *readfile* y al encontrar el fin de archivo, lo vuelve a dejar apuntando a la terminal. El archivo de salida se altera al evaluar *writefile* y se vuelve a quedar en la terminal al dar *close*.

El editor fue implantado usando la función de biblioteca *exec* del compilador para 'C' de *Microsoft*, que permite suspender temporalmente un proceso, en tanto se ejecuta otro. Así se suspende la acción del intérprete mientras se trabaja con cualquier editor. Al terminar de trabajar con el editor se vuelve a trabajar con el intérprete de igual manera que al terminar de ejecutar cualquier otra función.

Las funciones para manejo de la pantalla, se implementan usando la interrupción por *software* 10, la cual proporciona la interfase con la pantalla. Y poniendo en el registro AH el valor adecuado. Por ejemplo, para posicionar el cursor en el lugar deseado, en AH se pone un dos, y en la parte alta del registro D se pone el renglón y en la baja la columna.

Para seleccionar algún modo de operación de la pantalla, en la parte alta del registro Ax pone un cero y en la parte baja se pone el número que corresponde al modo seleccionado, de acuerdo a la siguiente tabla:

0	Alfanumérico blanco y negro en 40 columnas y 25 renglones
1	Alfanumérico color en 40 columnas y 25 renglones
2	Alfanumérico blanco y negro en 80 columnas y 25 renglones
3	Alfanumérico color en 80 columnas y 24 renglones
4	Gráfico en color con 320 X 200 puntos
5	Gráfico en blanco y negro con 320 X 200 puntos
6	Gráfico en blanco y negro con 640 X 200 puntos

Si en la parte alta del registro Ax pone un uno, en la parte baja se pone el número que corresponde al color seleccionado, de acuerdo a la siguiente tabla:

1	Cyan
2	Magenta
3	Blanco

La función *(bye)* regresa el apuntador FIN, con el cual se puede salir del ciclo de lectura, evaluación y escritura.

La implementación del colector de basura, se detalla en otra sección, por ser una parte importante del intérprete.

4.2.4 Módulo de escritura

El módulo de escritura consta de rutinas que se encargan de imprimir en el archivo de salida el resultado de la evaluación de la expresión dada por el usuario, esto puede ser con o sin formato. Los parámetros de la rutina sin formato, llamada pinta son: el nodo inicial de la expresión, un margen que indica la sangría debe dejarse. Si el nodo que recibe es ERRORFORM no escribe nada pues el mensaje de error ya se escribió y si es FIN, tampoco porque es sólo para salir del ciclo.

Si el nodo que toma como parámetro es un átomo simbólico va a la tabla de objetos, toma su nombre y lo escribe. El índice en esa tabla lo toma del campo de dirección del car.

Si el nodo que toma como parámetro es un átomo numérico, toma su valor del campo entero o real del car según le indique el tipo en el campo bits de ese nodo.

En caso de una lista imprime un paréntesis izquierdo, luego va escribiendo el car y el cdr de la lista, de acuerdo a las reglas ya mencionadas, en caso de tratarse nuevamente de una lista, vuelve a realizar este paso. Así recursivamente hasta que el cdr de la lista sea igual a nil.

Al terminar esta rutina escribe el *prompt* para indicar que se esta listo para comenzar con el ciclo de lectura-evaluación y escritura.

La impresión con formato o *pretty printer* se realiza al estar siguiendo paso a paso una función, al editar un archivo que contenga funciones en LISP o bien al llamarse explícitamente. La forma de trabajar del *pretty printer* es determinando qué clase de elemento va a escribir, y si no se trata de una lista, simplemente lo escribe.

Al encontrar una lista verifica si se trata de la definición de una función o un macro, pues en caso afirmativo, en la primera línea debe ir la definición de la función o macro con su lista de argumentos y en las siguientes se escriben cada una de las expresiones que forman el cuerpo, alineándolas debajo de la 'f' de defun o *defmacro*.

4.2.4 Módulo de escritura

El módulo de escritura consta de rutinas que se encargan de imprimir en el archivo de salida el resultado de la evaluación de la expresión dada por el usuario, esto puede ser con o sin formato. Los parámetros de la rutina sin formato, llamada pinta son: el nodo inicial de la expresión, un margen que indica la sangría debe dejarse. Si el nodo que recibe es ERRORFORM no escribe nada pues el mensaje de error ya se escribió y si es FIN, tampoco porque es sólo para salir del ciclo.

Si el nodo que toma como parámetro es un átomo simbólico va a la tabla de objetos, toma su nombre y lo escribe. El índice en esa tabla lo toma del campo de dirección del car.

Si el nodo que toma como parámetro es un átomo numérico, toma su valor del campo entero o real del car según le indique el tipo en el campo bits de ese nodo.

En caso de una lista imprime un paréntesis izquierdo, luego va escribiendo el car y el cdr de la lista, de acuerdo a las reglas ya mencionadas, en caso de tratarse nuevamente de una lista, vuelve a realizar este paso. Así recursivamente hasta que el cdr de la lista sea igual a nil.

Al terminar esta rutina escribe el *prompt* para indicar que se esta listo para comenzar con el ciclo de lectura-evaluación y escritura.

La impresión con formato o *pretty printer* se realiza al estar siguiendo paso a paso una función, al editar un archivo que contenga funciones en LISP o bien al llamarse explícitamente. La forma de trabajar del *pretty printer* es determinando qué clase de elemento va a escribir, y si no se trata de una lista, simplemente lo escribe.

Al encontrar una lista verifica si se trata de la definición de una función o un macro, pues en caso afirmativo, en la primera línea debe ir la definición de la función o macro con su lista de argumentos y en las siguientes se escriben cada una de las expresiones que forman el cuerpo, alineándolas debajo de la 'f' de *defun* o *defmacro*.

Si encuentra una condicional alinea todas las claúsulas una debajo de otra. Si va a formatear una lista que contenga como primer elemento la función *DO* o *DO-S* alinea las listas que contienen la variables locales debajo del primer paréntesis de la función. La condición queda en otra línea un caracter antes que las listas antes mencionadas y el cuerpo dos caracteres antes. El formato para *LET* o *LET-S* es similar, excepto que no hay condicional.

Si cualquier lista a formatear tiene más de cuarenta caracteres de longitud, en la primera línea escribe los dos primeros elementos de la lista y en las siguientes escribe cada uno de los restantes elementos de la lista alineados al principio del segundo elemento de la primera línea.

Los ejemplos mostrados en el ápendice A, fueron impresos con *pretty printer*.

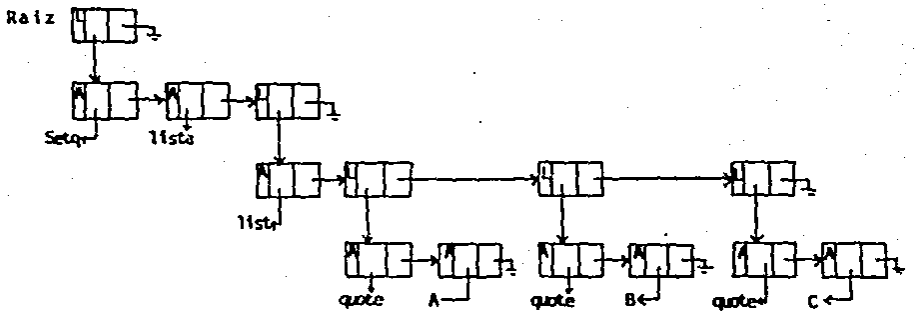
4.2.5 Colección de basura

En esta sección se hace notar la necesidad de recuperar espacio disponible, se describen diferentes técnicas para hacerlo principalmente la de colección de basura, así como algoritmos para hacerlo, en particular se describe detalladamente el algoritmo empleado en el intérprete de Milisp y se muestran las rutinas que programan este algoritmo.

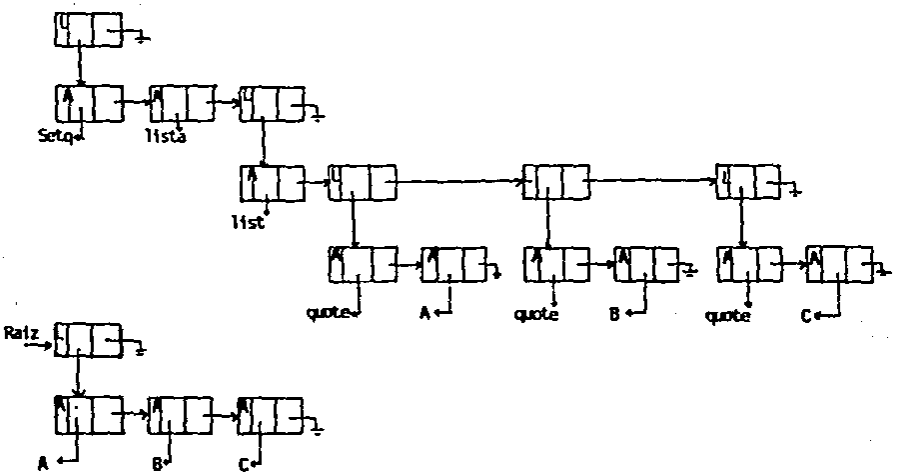
Al estar trabajando con un intérprete de Lisp, continuamente se generan nodos que dejan de ser accesibles Por ejemplo, supóngase que un usuario proporciona la siguiente expresiones-s:

```
(Setq lista (list 'a 'b 'c))
 (a b c)
```

Una vez evaluada, casi todos los nodos empleados en la representación de la expresión-s dejan de ser accesibles.



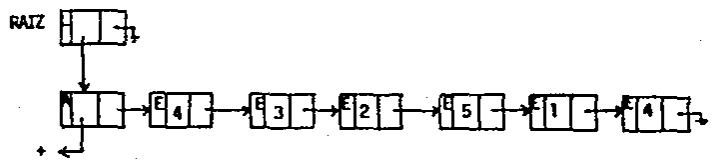
Representación antes de la evaluación



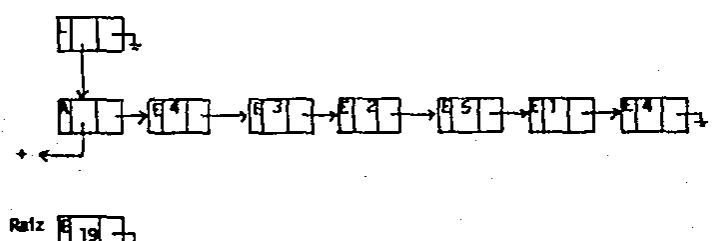
Representación después de la evaluación

Considérese otro ejemplo: al teclear la expresión-s
(+ 4 3 2 5 1 4)

el *parser* crea la siguiente lista:



Representación de (+ 4 3 2 5 1 4) antes de evaluarla



Representación después de la evaluación

El nodo raíz indica el inicio de esta lista. Al evaluarla RAIZ apunta al resultado y como se ve ya no hay forma de accesar los elementos de la suma

Por lo tanto, llega el momento en que en el *heap* no hay nodos para continuar trabajando, así que hay que recuperar aquellos nodos que fueron empleados con anterioridad pero que no son accesibles.

Existen tres métodos para hacer esto. El primero es el de regresar explícitamente a la lista de nodos disponible, aquellos que ya están libres, pero en el caso de LISP esto es imposible pues implicaría que el usuario del intérprete conociera cuáles son además de que haría lento el intérprete y podría suceder que elementos que se deberían poder accesar quedaran inaccesibles.

Otro método es el llamado de *contador de referencias*. Con este método a cada nodo se le agrega un campo numérico que indica la cantidad de apuntadores que existen hacia él. Cuando un elemento es tomado del *heap*, su contador de referencias tiene valor igual a uno. Este contador se incrementa en uno cada vez que se crea un nuevo apuntador a ese elemento y se decrementa en uno cada vez que se destruye un apuntador a ese elemento. Al tener cero como valor, se regresa al área *heap*.Este método no es utilizado en el intérprete porque además del espacio extra en cada nodo, involucra más operaciones (los incrementos, los decrementos y la prueba de cero) en cada instrucción que ejecute el intérprete.

El tercer método es llamado *recolección de basura*. Un elemento se considera basura, si está disponible para ser usado pero no está en el *heap*. En este método se permite que se genere basura, para evitar las referencias dañadas. Cuando en el *heap* ya no existen nodos disponibles (lo cual puede ocurrir en cualquier momento) y se necesita más espacio para alguna operación, ésta se suspende temporalmente y se inicia el procedimiento de colección de basura, el cual se encarga de identificar los elementos basura del *heap* y recuperarlos. La operación original continua en el punto que se suspendió, como si no hubiera pasado nada, y la basura se vuelve acumular otra vez hasta que en el *heap* no haya nodos disponibles y se vuelve a llamar al colector, así tantas veces como sea necesario. Esto da al usuario la sensación de tener una memoria ilimitada.

Antes se mencionó que el colector se activa cuando los nodos disponibles se agotan, pero el usuario puede también activarlo explícitamente en el momento que lo desee por medio de la función primitiva para tal fin.

El proceso de colección de basura emplea un bit, para poder distinguir los elementos basura de los que no lo son. Antes de activar el colector, este bit está en cero en todos los nodos del *heap*.

La colección de basura tiene dos etapas. La primera es detectar cuáles elementos son basura y cuáles son activos. A los nodos activos y accesibles les prende el bit de basura. Una vez que se han marcado todos los elementos activos, todos aquellas que tienen el bit para basura en cero, son considerados basura.

En la segunda etapa todos los elementos basura se incorporan a los disponibles dentro del *heap* y se apaga el bit para basura de todos los elementos del *heap*, para posteriores recolecciones de basura.

La forma de regresar los nodos disponibles a la lista de espacio libre consiste en revisar secuencialmente el *heap* y si el nodo que se esta revisando tiene el bit para basura prendido se ignora hasta encontrar un nodo que lo tenga apagado y este nodo se pone al final de la lista de espacio libre.

Como se puede notar recuperar los nodos disponibles es simple, el problema es determinar cuáles elementos están disponibles para ser reusados y cómo marcarlos.

Cuando se agotan los nodos disponibles y se llama al colector cada elemento en el *heap* es activo o basura. Desafortunadamente, la inspección de cada elemento no indica su estado ya que no hay nada intrínseco a un elemento que indique si es basura o no. La presencia de un apuntador a un elemento del *heap* desde otro elemento del *heap* no indica necesariamente que el elemento apuntado está activo ya que ambos elementos pueden ser basura.

Un elemento es accesible, si existe un apuntador a él desde fuera del *heap* o en otro elemento accesible dentro del *heap*. Esto es, una vez identificado un nodo accesible, toda cadena de apuntadores que parten de él también son accesibles.

Una vez que se conocen los elementos a marcar se debe saber cómo hacerlo, para esto existen varios algoritmos. El algoritmo inmediato es partir de los apuntadores externos para seguir las cadenas de apuntadores dentro del *heap* hasta que cada elemento activo se haya marcado. Para seguir la cadena de apuntadores se utiliza una pila, lo cual constituye un problema pues la pila ocupa lugar en memoria que podría ser utilizado para tener más nodos disponibles.

Otro algoritmo es el propuesto por *Schor* en el cual para marcar invierten las cadenas empezando con un apuntador externo en el *heap* una cadena de apuntadores es recorrido hasta el final. Cuando se recorre cada apuntador el elemento es marcado y el apuntador invertido. Cuando se llega al final de una cadena de apuntadores invertidos se hace el recorrido de regreso. En este proceso, se requieren dos recorridos para cada cadena de apuntadores (uno en cada dirección) pero sólo se requieren tres registros extras, para almacenamiento temporal, en lugar de una pila y un bit más en el campo de *bits* para indicar si la lista esta invertida o no.

El primer algoritmo, aunque es simple, tiene el problema que la pila, es de tamaño fijo, por lo tanto, puede saturarse sin haber termindo de marcar la cadena de nodos accesibles y el algoritmo de inversión de listas es mucho menos eficiente que el de la pila simple ya que cada lista debe recorrerse dos veces.

El algoritmo empleado en este intérprete se basa en el propuesto por *Schor* y *Waite*. El cual utiliza una pila para marcar las listas y si ésta se llena empieza a invertir el resto de la lista. En este algoritmo si las listas no son grandes las marca con rapidez (pues no hace doble recorrido), pero si son muy grandes marca rápidamente una parte y con poca memoria la otra. Por lo tanto se combina rapidez con ahorro de memoria.

Para encontrar los nodos inmediatamente accesibles se recorre secuencialmente la lista de objetos (ob-list) y se ve si el objeto tiene valor asociado, en caso afirmativo marca como no-basura los nodos que constituyen la expresión-s de ese valor y hace lo mismo en caso de que el objeto tenga asociado el cuerpo de una función y/o una lista de propiedad.

Se revisa la pila en donde se encuentran los parámetros de las funciones y todos los nodos que están ahí también forman el inicio de cadenas que deben marcarse como no-basura.

Además de los elementos inmediatamente accesibles, se deben considerar no-basura los elementos que se estén utilizando en el desarrollo de la operación actual. Estos los va marcando como activos cada función primitiva del intérprete y los marca como no-activos al salir.

Si durante el proceso de marcado de una expresión-s se agota la pila, entonces se llama a la rutina Invierte que es la encargada de continuar marcando los nodos de esa lista pero invirtiéndola.

La descripción del algoritmo empleado se da a continuación.

1. [Marca átomos]
   Si el nodo no es cabeza de lista, entonces
      Marca(nodo) ← 1 y termina.
2. [Marca listas usando una pila]
   Hacer
      Si el nodo no es cabeza de lista, entonces
         Push (nodo)
         Marca(nodo) ← 1
         Avanza por el CAR
      En otro caso
         Marca(nodo) ← 1
         Avanza por el CDR
         Si CDR igual a *nil* y no se ha llenado la pila
            nodo = Pop.
      Si la pila está llena, llamar a Invierte(nodo)
   Mientras haya nodos en la pila.

El algoritmo de inversión de listas es el siguiente:

1. [Inicializa]
   T ← vacío
   P ← vacío

2. [Marca]
   Marca(nodo) ← 1

3. [Atomo ?]
   Si Atomo (p) = 1 y T = vacío finalizar

4. [Baja por el CAR, invirtiendo apuntadores]
   Q ← CAR (nodo)
   Si Q es diferente de vacío y Marca(Q) = 0 entonces
      Atomo (Q) ← 1
      CAR (P) ← T
      T ← P
      P ← Q
      Ir al paso 2.

5. [Baja por el CDR, invirtiendo apuntadores]
   Q ← CDR (nodo)
   Si Q es diferente de vacío y Marca(Q) = 0 entonces
      CDR (P) ← T
      T ← P
      P ← Q
      Ir al paso 2.

6. [Sube, restaurando apuntadores]
   Si T = vacío entonces finaliza.
   Q ← T
   Si Atomo (Q) = 1
      Atomo (Q) ← 0
      T ← CAR (Q)
      CAR(Q) ← P
      P ← Q
      Ir al paso 5.
   En otro caso
      T ← CDR (Q)
      CDR(Q) ← P
      P ← Q
      Ir al paso 6.

Referencias

Sobre el diseño de intérpretes se puede consultar Fitch[1977] y McInter[1974], en particular de Lisp existen Tucker[1974], Allen [1980] que describe formalmente el lenguaje y su implantación. Aho & Ullman [1979] detallan técnicas para implantar analizadores léxicos y sintácticos; así como la definición formal de lenguajes. Para manejo de memoria, Bobrow [1967] y Standish[1979], en este último también se puede consultar acerca de funciones de dispersión igual que en el Knuth[1975]. Las técnicas de colección de basura son tratadas por Christopher, Pratt[1979] y Knuth[1975]. Algoritmos para "pretty printers" se pueden encontrar en Goldstein y Vaucher pero este trabajo se basó principalmente en el Winston [1984].

Conclusión

Para concluir este trabajo se presenta una comparación de MiLisp con otra implantación de LISP para computadoras personales con sistema operataivo MS-DOS, esta implantación se conoce como mulisp. Además se mencionan posibles extensiones de MiLisp.

Mulisp interpreta una versión de Lisp que es una combinación de Interlisp y de Maclisp, en cambio MiLisp interpreta a Common Lisp.

MiLisp cuenta con 103 funciones primitivas implementadas como rutinas en 'C' y además cuenta con 20 funciones implementadas como rutinas en Common Lisp; éstas últimas se encuentran en un archivo llamado LISP.LIB, esto permite que el lenguaje crezca pues el usuario puede de manera sencilla crear nuevas funciones y guardarlas en ese archivo el cual se lee de manera automática al iniciarse la sesión de trabajo.

mulisp por su parte cuenta con aproximadamente 100 funciones primitivas escritas en lenguaje ensamblador.

MiLisp maneja enteros, de 16 bits, con signo; mulisp trabaja con enteros gigantes almacenados como cuerdas binarias de longitud variable de hasta 2048 bits, sin embargo mulisp no trabaja con números reales y MiLisp sí lo hace.

MiLisp acepta arreglos multidimensionales y el total de elementos está limitado por la capacidad de memoria del intérprete; mulisp no maneja arreglos.

Para control de secuencia MiLisp cuenta con varias funciones, en cambio mulisp está limitado a dos funciones.

Las funciones en mulisp tienen un COND implícito, con lo que se obliga al programador a escribir siempre funciones condicionales.

MiLisp maneja funciones con nombre, sin nombre, con número fijo o variable de parámetros; también maneja macros, variables locales, la asignación de los valores puede ser hecha en forma secuencial o en paralelo.

mulisp maneja variable locales pero tomándolas de la llamada a funciones cuando se describen parámetros en exceso.

mulisp cuenta con un editor (escrito en Lisp) orientado al lenguaje, el cual permite editar sólo expresiones-s, este editor llama automáticamente al *"pretty printer"*, el usuario no puede abandonar la sesión de edición hasta que la expresión-s este correcta.

MiLisp en cambio, permite que se llame a ejecución cualquier editor que se desee con lo cual se pueden editar no sólo expresiones-s sino también texto.

Al estar trabajando en mulisp y ocurrir un error se detiene la ejecución y se entra a un estado de BREAK en el cual se pueden analizar y/o modificar valores de expresiones-s. En particular se cuenta con una función para hacer la traza de funciones.

MiLisp proporciona dos funciones para detener la ejecución de una expresión-s, sólo que ambas se deben dar explícitamente, una es dentro del cuerpo de la función que se desea detener y la otra puede darse en cualquier momento. Además de contar con la función para trazado de funciones se tiene una para seguir paso a paso la ejecución de una función.

MiLisp cuenta con rutinas básicas para graficación como son una para cambiar a modo gráfico con alta o baja resolución, a color o en blanco y negro, si es a color se puede elegir el color deseado; para trazado de líneas, para manejo de pantalla como son limpiarla, poner el cursor en una posición determinada.

En mulisp no existen esas facilidades de graficación pero se tienen funciones para interactuar directamente con el sistema operativo, así que a través de interrupciones se pueden obtener las rutinas para graficación.

MiLisp cuenta con un contador automático de paréntesis sin cerrar, el cual se va imprimiendo cada vez que se teclea un retorno de carro, pero si se teclean paréntesis en exceso los ignora; mulisp simplemente ignora los paréntesis en exceso.

Hasta el momento se ha estado hablando de lo que tiene Milisp, a continuación se mencionan algunas posibles extensiones al intérprete.

Actualmente se tiene la posibilidad de tener un *heap* de hasta 64 Kbytes (2^{16}), esto podría ampliarse hasta 1 Megabyte, tomando del campo de bf bits los cuatro sobrantes y uniendolos con los 16 que se toman actualmente formar las direcciones. Si bits en lugar de ocupar un byte ocupara una palabra se tendría posibilidad de accesar hasta 8 Megabytes, desde luego siempre y cuando la computadora tenga memoria suficiente para ello.

Otra posible extensión sería el agregarle rutinas para permitir al intérprete manejar números de longitud ilimitada como mulisp.

Se podría desarrollar también un compilador y permitir que LISP ejecute funciones compiladas.

Una posibilidad que desde mi punto de vista daría mayor velocidad a los intérpretes, en general, es que el proceso de colección de basura se haga en forma concurrente con el proceso de evaluación o simplemente en cuanto el usuaro deje de teclear.

Bibliografía

Aho, Alfred & Ullman Jeffrey [1979]
Principles of Compiler Design
Addison-Wesley, Reading, Mass.

Allen John [1978]
Anatomy of Lisp
Mc Graw Hill

Allen John
An Overview of LISP
Byte August 1979 (10-17)

Barstow David, Shrobe Howard and Sandewall Erick [1984]
Interactive Programming Environments
Mc Graw Hill

Bridger Mark & Frampton John
Creating a Standard LISP
Tech Journal
Dec. 1985 Vol 3 No. 12 (98-112)

Bobrow G. Daniel and Murphy Daniel
Structure of a LISP system using two-level storage
Communications of the ACM
Vol 10, no. 3, march 1967 p. 155-159

Christopher T.W.
Reference Count Garbage Collection
Software Practice and Experience
(vol 14, No. 16) p. 503- 507

Fitch J.O and Norman A. C.
Implementing LISP in a High-level language
Software Practice and Experience
Vol 7, (1977) p. 713-725

Franklin, Marck [1984]
Using the IBM Personal Computer:
Organization & Assambly Language Programming

Gabriel, Richard P. [1985]
Performance & Evaluation of Lisp Systems
The MIT Press

Guzmán Adolfo, Segovia Raymundo
A Parallel Configurable Lisp Machine
Comunicaciones Técnicas 1976
Serie Naranja Vol 7 No. 133

Hearn A.C. and Norman A. C
A One-pass pretty- printer
ACM Sigplan Notices Vol 14 # 12. (50-58)

McCarthy, John [abril 1960]
Recursive Functions of Symbolic Expressions
Communications of ACM,3,4 (184-195)

MacCarthy John [1965]
Lisp 1.5 Programmer's Manual

Mc Intere, thomas [1974]
Software Interpreters for Microcomputers
John Wiley and Sons.

Magidin Mario; Segovia Raymundo
Implementation of Lisp 1.6 on the B6700 Computer
Comunicaciones Técnicas CIMAS Serie B: Vol 5 #70

Knuth, Donald E [1975]
The Art of Computer Programming, Vol 1 : Fundamental Algorithms,
Addison-Wesley, Reading, Mass.

Knuth, Donald E [1975]
The Art of Computer Programming, Vol 2 : Sorting and Searching,
Addison-Wesley, Reading, Mass.

Organick, Forsythe, Plummer [1978]
Programming Languages Structures
Academic Press.

Pratt, Terrence W [May 1979]
Programming Languages: Design and Implementation
Prentice Hall, Inc.

Sandewall Erik
Programming in an interactive environment: The "LISP" experience
Computer Surveys Vol 10 No. 1, March 1978

Standish Thomas A. [1979]
Date Structure Techniques
Addison-Wesley

S Tucker Taft
The Design of an M6800 LISP interpreter
Byte August 1979

Tucker, Allen B [1985]
Programming Languages
Mc Graw Hill

Vaucher Jean G.
Pretty-Printing of trees,
Software Practice and experience
Vol 10 (jul 1980) p. 553-561

Winston Patrick H.; Berthold Klaus Paul Horn
Lisp 2nd Edition
Addison-Wesley

Wegner P [Dec 1976]
Programming Languages - The first 25 years.
IEEE Transactions on Computers
(1207-1225)

MPC Operations Guide Columbia

P-LISP Version 3.0 for the apple II/II+ (1982)
Published by GNOSIS a division of pegasys systems, inc.

Cromemco LISP instruction manual
by Cromemco Inc. (1980)

muLISP-83 Reference Manual

Anexo
Ejemplos

La mayoría de los ejemplos presentados en esta sección se deben a personas que emplearon MiLisp durante sus respectivos cursos. Estas personas son:

Salvador López Mendoza

Rafael Morales Gamboa

José Antonio López Saucedo.

Los ejemplos presentados son:

1. Colocación de reinas en un tablero de ajedrez
2. Evaluación de expresiones en notación prefija
3. Discos de Banerji
4. Tipos de datos abstractos (ADT)
5. Gráficas
6. Send + More = Money

```
;
;  Con este programa se muestran las posiciones en que puede ser
;  acomodada una reina de ajedrez, en un tablero de n x n (n >= 4).
;

(DEFUN QUEEN (SIZE)
   (QUEEN-AUX NIL 0 SIZE) )

(DEFUN QUEEN-AUX (BOARD N SIZE)                  ; Empieza en el siguiente renglon
   (COND ((= N SIZE) (PRINT (REVERSE BOARD) ) ); Encuentro una solucion
         (T (QUEEN-SUB BOARD N 0 SIZE) ) ) )    ; Trato en este renglon

(DEFUN QUEEN-SUB (BOARD N M SIZE)                ; trato siguiente columna
   (COND ((= M SIZE) )                           ; es el fin del renglon?
         (T (COND ((CONFLICT N M BOARD) )        ; conflicto ?
                  (T (QUEEN-AUX (CONS (LIST N M)
                                      BOARD)
                                (+ N 1)
                                SIZE)))
            (QUEEN-SUB BOARD N (+ M 1) SIZE) ) ) ) ) )

(DEFUN CONFLICT (N M BOARD)
   (COND ((NULL BOARD) NIL)
         ((OR (INTENTO N M (CAAR BOARD) (CADAR BOARD))
   (CONFLICT N M (CDR BOARD))))))

(DEFUN INTENTO (I J A B)
   (OR (= I A)
       (= J B)
       (= (- I J) (- A B))
       (= (+ I J) (+ A B))))


;; Ejecucion

(queen 4)
  ((0 1) (1 3) (2 0) (3 2))
  ((0 2) (1 0) (2 3) (3 1))
```

```
;
;    Parte del trabajo de un compilador es traducir las expresiones aritmeticas
;    al lenguaje de maquina. En este ejemplo se considera una computadora que
;    tiene siete registros que se pueden usar para almacenar resultados en forma
;    temporal y que ademas cuenta con las instrucciones MOV, ADD, SUB, MUL y DIV
;    El ejemplo es un programa que genera codigo para expresiones aritmeticas en
;    notacion prefija


(DEFUN COMPILE (S)
   (COMP1 1 S))

(DEFUN COMP1 ( R S)
   (COND ((ATOM S) (LIST (LIST 'MOVE R S)))
         (T (APPEND (COMP1 R (CADR S))
                    (COMP1 (+ R 1) (CADDR S))
                    (LIST (LIST (OPCODE (CAR S)) R (+ R 1)))))))

(DEFUN OPCODE (OP)
   (COND ((EQUAL OP '+) 'ADD)
         ((EQUAL OP '-) 'SUB)
         ((EQUAL OP '*) 'MUL)
         ((EQUAL OP '/) 'DIV)
         (T 'ERR)))

(SETQ S '(* (+ A B) (  C (/ D E))))

;;  Ejecucion


(compile s)

((MOVE 1 A)
 (MOVE 2 B)
 (ADD 1 2)
 (MOVE 2 C)
 (MOVE 3 D)
 (MOVE 4 E)
 (DIV 3 4)
 (SUB 2 3)
 (MUL 1 2))
```

Discos de Banerji

Se tienen cuatro discos concentricos, cada uno dividido en ocho partes y en cada parte un numero. Se deben acomodar los discos de tal forma que la suma de los numeros alineados sea la misma para cada segmento.

Para resolver el problema se uso una lista para representar cada disco, de manera que el problema se reduce a acomodar las cuatro listas de tal forma que la suma de todas las columnas sea la misma.

Este programa fue realizado por SALVADOR LOPEZ MENDOZA, para su curso de Inteligencia Artificial.

```
(SETQ L1 '( 2 1 3 4 2 5 1 3 ) )       ; Definicion de los discos
(SETQ L2 '( 3 2 3 4 1 3 4 5 ) )
(SETQ L3 '( 3 4 5 3 3 2 2 1 ) )
(SETQ L4 '( 5 1 5 3 4 3 2 4 ) )

(SETQ LIST1 (LIST L2 L3 L4 ) )        ; Definicion de anidamiento
(SETQ LISTA2 (LIST L1) )              ; El 1er disco esta fijo

(DEFUN IMPRIME (LISTA SUMA)           ; Imprimo la posicion actual
                                      ;de los discos y la suma de cada columna
   (PRINT '(UNA SOLUCION AL PROBLEMA ES) )
   (PRINT (CAR LISTA) )
   (PRINT (CADR LISTA) )
   (PRINT (CADDR LISTA) )
   (PRINT (CAR (LAST LISTA) ))
   (PRINT SUMA)
   (TERPRI)  )

(DEFUN VERIFICA (LISTA)               ; Determina si la posicion actual de los
                                      ; discos es una solucion al problema
   (SETQ S1 (MAPCAR '+ (CAR LISTA) (CADR LISTA) ) )
   (SETQ S2 (MAPCAR '+ (CADDR LISTA) (CAR (LAST LISTA) ) ) )
   (SETQ SUM (MAPCAR '+ S1 S2) )
   (SETQ SUMA (LAST SUM) )
   (SETQ SUMV (APPEND SUMA SUMA SUMA SUMA SUMA SUMA SUMA SUMA) )
   (SETQ RES (MAPCAR '= SUM SUMV) )
   (COND ( (APPLY 'AND RES) (IMPRIME LISTA SUM) ) )   ; SOLUCION => imprime.
                                                )

(DEFUN COLOCA ( LLISTA )              ; Toma el siguiente disco y lo coloca
   (LET ( (LACT (CAR LLISTA) ) )
      (COND ( (NULL LACT) (VERIFICA LISTA2 ) )  ; Si no hay mas verifica si
                                                ; es solucion al problema
            (T (DO ( (CONT 0 (+ CONT 1) ))      ; Coloca el disco actual en
                   ( (ZEROP (- CONT 8)) T )     ; todas sus posiciones.
                   (SETQ LACT (APPEND (CDR LACT) (LIST (CAR LACT)))) ;Giralo
                   (SETQ LISTA2 (CONS LACT LISTA2) )  ; Lo coloca para rev.
                   (COLOCA (CDR LLISTA ) )            ; Verifica los restantes
                   (SETQ LISTA2 (CDR LISTA2)))))      ; Elimina para revisar
                                                      ; otra posibilidad.
                                                )))))
```

```
1: Ejecucion

(COLOCA LIST1)


(UNA SOLUCION AL PROBLEMA ES)
(4 5 2 4 5 1 5 3)
(4 5 3 3 2 2 1 3)
(2 3 4 1 3 4 5 3)
(2 1 3 4 2 5 1 3)
(12 12 12 12 12 12 12 12)
```

El programa que se lista a continuacion es un interprete para el lenguaje de las expresiones sobre el tipo de datos de las pilas binarias (referencia de Wand) ---pilas formadas por ceros y unos--- y esta construido utilizando como fundamento las especificaciones formales del tipo de datos y del lenguaje. El dise\~no original se debe a Gloria Quintanilla y alumnos (referencia de Quintanilla) quienes escribieron el interprete utilizando una variante de LISP conocida como muLISP. La version que se presenta aqui es una traduccion a CommonLISP de ese mismo interprete y fue realizada por : RAFAEL MORALES GAMBOA, en un curso de Teoria Matematica de la Computacion.

El programa esta compuesto por cuatro tipos de expresiones simbolicas:

Definiciones de constantes y variables que contienen informacion 'ambiental', como los nombres de los elementos del conjunto de pilas binarias, de los procedimientos y funciones intrinsecos, y de los predicados y funciones definidos por el programador;

Funciones que analizan la estructura de expresiones simbolicas para determinar su pertenencia o no al lenguaje de las expresiones;

Funciones que proporcionan el significado de los distintos expresiones que conforman el lenguaje; y

Funciones auxiliares para el manejo de las expresiones simbolicas.

Referencia de Wand:
Wand, M.:"Induction, Recursion and Programming". North Holland, 1980, caps. 3 y 4.

Quintanilla, G. y otros :"Dise\~no del interprete de un lenguaje de programacion a partir de su especificacion formal". Comunicaciones Internas del IIMAS

```
; 0. Funcion auxiliar

; IGUALES(<param1>,<param2>)
;       Verifica si param1 = param2
;
(DEFUN IGUALES (PARAM1 PARAM2)
   (COND ((AND (NUMBERP PARAM1) (NUMBERP PARAM2) )
          (= PARAM1 PARAM2) )
         (T (EQUAL PARAM1 PARAM2) ) ) )
```

```
; 1. Definicion de los ambientes del ADT.

; Nombres y numero de parametros de las funciones intrinsecas.
;
(SETQ A-FIS '((POP 1) (PUSH1 1) (PUSH0 1) ) )


; Nombres y numero de parametros de los predicados intrinsecos.
;
(SETQ A-PIS '((IS0P 1) (IS1P 1) (EMPTYP 1) ) )



; 2. Definicion de las Funciones de Aceptacion de las Estructuras del ADT.

; Acepta o rechaza una expresion simbolica, dependiendo si es o no
; una expresion del lenguaje.
;
(DEFUN ES-EXP (FENV ENV S-EXP)
   (COND ((OR (EN-CIS S-EXP)
              (AND (EN-IVS FENV S-EXP)
                   (ES-IVS ENV S-EXP) )
                (ES-FUN-INT FENV ENV S-EXP)
                (ES-FUN-USUA FENV ENV S-EXP)
                (ES-COND FENV ENV S-EXP) ) T)
         (T NIL) ) )



; Verifica si una expresion simbolica es una constante valida para
; el ADT de los pilas binarios.
;
(DEFUN EN-CIS (S-EXP)
   (COND ((NULL S-EXP) T)
         ((ATOM S-EXP) (EQUAL S-EXP 'EMPTY) )
         ((NOT (NUMBERP (CAR S-EXP) ) ) NIL)
         ((OR (= (CAR S-EXP) 1)
              (= (CAR S-EXP) 0) )
          (EN-CIS (CDR S-EXP) ) ) ) )

                                                              /

; Dada una expresion simbolica "s_exp" la lista de asociacion
; "a-fis" del ADT entrega "nil" si "s-exp" no esta en "a-fis" y en
; otro caso entregara el numero de argumentos que tiene asociado.
;
(DEFUN EN-FIS (S-EXP)
   (LET ((LST A-FIS) )
     (DO ((RESULTADO NIL) )
         ((NULL LST) RESULTADO)
       (COND ((EQUAL (CAAR LST) S-EXP)
              (SETQ RESULTADO (CADAR LST) )
              (SETQ LST NIL) )
             (T (SETQ LST (CDR LST) ) ) ) ) ) )
```

```
; Verifica si "s-exp" es una lista de expresiones validas del lenguaje.
;
(DEFUN ES-LISTA-EXP (FENV ENV S-EXP)
   (COND ((NULL S-EXP) T)
         (T (AND (ES-EXP FENV ENV (CAR S-EXP) )
                 (ES-LISTA-EXP FENV ENV (CDR S-EXP) ) ) ) ) )


; Revisa si "s-exp" es una llamada correcta para una funcion intrinseca.
;
(DEFUN ES-FUN-INT (FENV ENV S-EXP)
   (COND ((ATOM S-EXP) NIL)
         ((IGUALES (EN-FIS (CAR S-EXP) ) (LENGTH (CDR S-EXP) ) )
            (ES-LISTA-EXP FENV ENV (CDR S-EXP) ) ) ) )


; Verifica si "s-exp" es la llamada de un predicado intrinseco.
;
(DEFUN ES-PRED (FENV ENV S-EXP)
   (COND ((ATOM S-EXP) NIL)
         ((IGUALES (EN-PIS (CAR S-EXP) ) (LENGTH (CDR S-EXP) ) )
             (ES-LISTA-EXP FENV ENV (CDR S-EXP) ) ) ) )


; Dada "s-exp" y la lista de asociacion "a-pis" del ADT entrego "nil"
; si "s-exp" no esta en "a-pis" y en otro caso entregara el numero
; de argumentos que tiene asociados.
;
(DEFUN EN-PIS (S-EXP)
   (LET ((LST A-PIS) )
      (DO ((RESULTADO NIL) )
          ((NULL LST) RESULTADO)
            (COND ((EQUAL (CAAR LST) S-EXP)
                     (SETQ RESULTADO (CADAR LST) )
                     (SETQ LST NIL) )
                  (T (SETQ LST (CDR LST) ) ) ) ) ) )


; Revisa que este realmente "s-exp" en el ambiente de variables
; "env" del ADT.
;
(DEFUN ES-IVS (ENV S-EXP)
   (COND ((NULL ENV) NIL)
         ((EQUAL (CAAR ENV) S-EXP) T)
         (T (ES-IVS (CDR ENV) S-EXP) ) ) )


; Determino si "s-exp" es una variable sintacticamente valida.
;
(DEFUN EN-IVS (FENV S-EXP)
   (AND (AND (ATOM S-EXP) (NOT (NUMBERP S-EXP) ) )
        (NOT (OR (EN-FIS S-EXP)
                 (EN-PIS S-EXP)
                 (EN-CIS S-EXP)
                 (EN-FENV FENV S-EXP) ) ) ) )
```

```
; Revisa si sintacticamente esta bien escrita una expresion
; condicional: la sintaxis debe ser
;
; (if <predicado> then <expresion> else <expresion>)
;
(DEFUN ES-COND (FENV ENV S-EXP)
   (AND (LISTP S-EXP)
        (EQUAL (NTH 1 S-EXP) 'IF)
        (ES-PRED FENV ENV (NTH 2 S-EXP) )
        (EQUAL (NTH 3 S-EXP) 'THEN)
        (ES-EXP FENV ENV (NTH 4 S-EXP) )
        (EQUAL (NTH 5 S-EXP) 'ELSE)
        (ES-EXP FENV ENV (NTH 6 S-EXP) )
        (NULL (NTH S-EXP 7) ) ) )


; Devuelve nil si "fus" no esta definido dentro del conjunto de
; funciones del usuario "fenv".  En caso contrario, devuelve el
; numero de parametros formales que tiene la funcion de usuario.
;
(DEFUN EN-FENV (FENV FUS)
   (COND ((OR (NULL FUS) (NULL FENV) ) NIL)
         ((EQUAL (CAAR FENV) FUS) (LENGTH (CADAR FENV) ) )
         (T (EN-FENV (CDR FENV) FUS) ) ) )


; Revisa que "s-exp" sea una llamada correcta para una funcion del
; usuario.
(DEFUN ES-FUN-USUA (FENV ENV S-EXP)
   (COND ((AND (LISTP S-EXP)
               (NOT (ES-IVS ENV (CAR S-EXP) ) )
               (IGUALES (EN-FENV FENV (CAR S-EXP) ) (LENGTH (CDR S-EXP) ) ) )
          (ES-LISTA-EXP FENV ENV (CDR S-EXP) ) ) ) )


; Escribe un mensaje de error si es que las funciones identificadoras
; del interprete no reconocieron una expresion simbolica.
;
(DEFUN ERROR (N-ERROR)
   (TERPRI)
   (TERPRI)
   (BLANKS 10)
   (COND ((= N-ERROR 1) '!--> EXPRESION IRRECONOCIBLE!)
         ((= N-ERROR 2) '!--> FUNCION  MAL ESCRITA!)
         (T '!--> COMANDO INV'ALIDO!) ) )



; 3. Definicion de las Funciones de Significado

; Entrega el valor que representa "cis", solo en caso de que sea
; identico a la constante "empty" devolvera el valor "nil".
;
(DEFUN M-CIS (CIS)
   (COND ((EQUAL CIS 'EMPTY) NIL)
         (T CIS) ) )
```

```
; Entrega una lista que contiene los valores que resultaron despues
; de evaluar cada uno de los argumentos de una funcion en particular.
;
(DEFUN M-LISTA-EXP (FENV ENV LST-EXP LST-CIS)
   (COND ((NULL LST-EXP) (REVERSE LST-CIS) )
         (T (M-LISTA-EXP FENV
                         ENV
                         (CDR LST-EXP)
                         (CONS (M-EXP FENV ENV (CAR LST-EXP) )
                               LST-CIS) ) ) ) )


; Concatena el simbolo "m" con el nombre de una funcion o predicado
; intrinsecos.
;
(DEFUN M-FIS (FIS)
   (PACK (LIST 'M FIS) ) )


; Las operaciones de 'push0', 'push1' y 'pop', y los predicados
; 'is0p', 'is1p' y 'emptyp' del ADT se efectuan por medio de las
; las funciones asociadas a una "m" concatenada con una expresion
; simbolica que esta en el conjunto de constantes del ADT.
;
(DEFUN MPUSH0 (CIS)
   (CONS 0 CIS) )


(DEFUN MPUSH1 (CIS)
   (CONS 1 CIS) )


(DEFUN MPOP (CIS)
   (CDR CIS) )


(DEFUN MIS0P (CIS)
   (= (CAR CIS) 0) )


(DEFUN MIS1P (CIS)
   (= (CAR CIS) 1) )


(DEFUN MEMPTYP (CIS)
   (NULL CIS) )


; Evalua una funcion intrinseca del lenguaje y entrega el valor
; calculado.
;
(DEFUN M-FUN-INT (FENV ENV S-EXP)
   (APPLY (M-FIS (CAR S-EXP) )
          (M-LISTA-EXP FENV ENV (CDR S-EXP) NIL) ) )
```

```
; Entrega el valor asociado con el nombre de una variable "ivs" en el
; ambiente de variables "env".
;
(DEFUN M-IVS (ENV IVS)
   (VALOR ENV IVS) )


; Obtiene el valor de una variable dada.
;
(DEFUN VALOR (ENV IVS)
   (COND ((NULL ENV) 'NO-DEFINIDA)
         ((EQUAL (CAAR ENV) IVS) (CADAR ENV) )
         (T (M-IVS (CDR ENV) IVS) ) ) )


; Devuelve el significado de una expresion condicional.
;
(DEFUN M-COND (FENV ENV COND)
   (COND ((M-EXP FENV ENV (NTH 1 COND) )
          (M-EXP FENV ENV (NTH 3 COND) ) )
         (T (M-EXP FENV ENV (NTH 5 COND) ) ) ) )


; Entrega una lista que contiene los parametros formales de la
; funcion del usuario que sera evaluada.
;
(DEFUN TOMA-FORMALES (FENV FUS)
   (COND ((EQUAL (CAAR FENV) FUS) (CADAR FENV) )
         (T (TOMA-FORMALES (CDR FENV) FUS) ) ) )


; Dado el nombre de una funcion del usuario "fus" y el conjunto de
; funciones del usuario "fenv", entrega la expresion que sera
; evaluada asociada a dicha funcion.
;
(DEFUN TOMA-CUERPO (FENV FUS)
   (COND ((EQUAL (CAAR FENV) FUS) (NTH 2 (CAR FENV) ) )
         (T (TOMA-CUERPO (CDR FENV) FUS) ) ) )


; Evalua una funcion de usuario evaluando el cuerpo de la funcion
; del usuario sobre un nuevo ambiente dado por los valores de los
; argumentos actuales.
;
(DEFUN M-FUN-USUA (FENV ENV S-EXP)
   (M-EXP FENV (CREA-AMBIENTE (TOMA-FORMALES FENV (CAR S-EXP) )
                              (M-LISTA-EXP FENV ENV (CDR S-EXP) NIL) )
                              (TOMA-CUERPO FENV (CAR S-EXP) ) ) )
```

```
; Da significado a toda aquella expresion simbolica "s-exp" que
; que tenga sentido en el lenguaje de las expresiones.
;
(DEFUN M-EXP (FENV ENV S-EXP)
   (COND ((EN-CIS S-EXP) (M-CIS S-EXP) )
         ((ES-IVS ENV S-EXP) (M-IVS ENV S-EXP) )
         ((OR (EN-FIS (CAR S-EXP) )
              (EN-PIS (CAR S-EXP) ) )
          (M-FUN-INT FENV ENV S-EXP) )
         ((ES-COND FENV ENV S-EXP) (M-COND FENV ENV S-EXP) )
         (T (M-FUN-USUA FENV ENV S-EXP) ) ) )



; 4. Definicion de las Funciones Constructoras de las Funciones de
;    Usuario y de los Ambientes ENV.

; Mezcla dos listas, una con los nombres de las variables que conforman
; el ambiente y otra con los valores asociados a las variables.
;
(DEFUN CREA-AMBIENTE (VARS VALS)
   (COND ((NULL VARS) NIL)
         (T (CONS (LIST (CAR VARS) (CAR VALS) )
                  (CREA-AMBIENTE (CDR VARS) (CDR VALS) ) ) ) ) )


; Construye y agrega nuevas funciones del usuario a las ya existentes en
; el conjunto "fenv", revisando que no se defina una ya existente, y que
; el cuerpo dela nueva funcion de usuario sea coherente con os
; parametros formales, siendo una expresion del lenguaje bien construida.
;
(DEFUN DEC-FUS (FENV NOMBRE PARAMETROS EXPRESION)
   (COND ((NOT (ES-EXP FENV (CREA-AMBIENTE PARAMETROS NIL)
                            EXPRESION) )
          (ERROR 2) )
         ((NULL (EN-FENV FENV NOMBRE) )
          (CONS (LIST NOMBRE PARAMETROS EXPRESION) FENV) )
         (T (REEMP-FUS FENV NOMBRE PARAMETROS EXPRESION NIL) ) ) )


; Reemplaza una funcion ya existente dentro del conjunto "fenv", pero
; no revisa que la expresion corresponda a los parametros formales
; y que sea una expresion valida del lenguaje.
;
(DEFUN REEMP-FUS (FENV NOMBRE PARAMETROS EXPRESION LST-RES)
   (COND ((EQUAL (CAAR FENV) NOMBRE)
          (APPEND (REVERSE LST-RES)
                  (CONS (LIST NOMBRE PARAMETROS EXPRESION)
                        (CDR FENV) ) ) )
         (T (REEMP-FUS (CDR FENV)
                       NOMBRE
                       PARAMETROS
                       EXPRESION
                       (CONS (CAR FENV) LST-RES) ) ) ) )
```

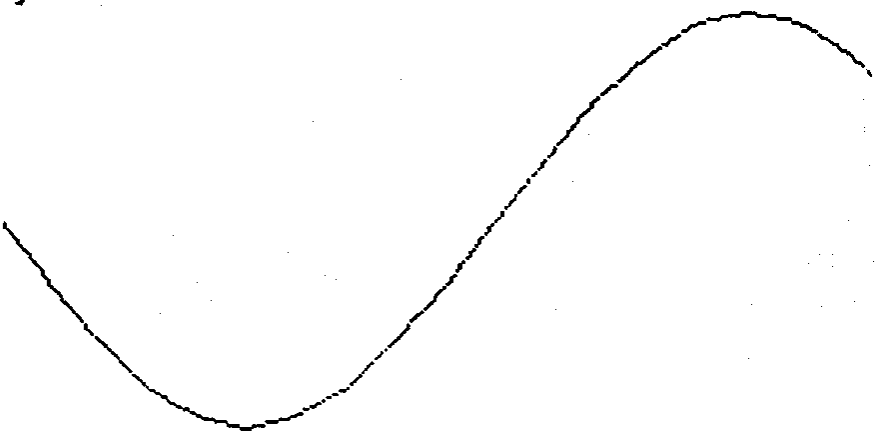
```
; 5. Definicion del Interprete de Expresiones

; Funcion evaluadora del lenguaje de expresiones sobre un Tipo de Datos
; Abstracto (ADT), de acuerdo a las funciones de significado dadas en
; el libro de Wand.
(DEFUN INTERPRETE (FENV ENV S-EXP)
   (COND ((NOT (ES-EXP FENV ENV S-EXP) ) (ERROR 1) )
         (T (M-EXP FENV ENV S-EXP) ) ) )
```

```
; Funciones de graficacion utilizando los conceptos de puerto de graficacion
; y ventana de vision, realizadas por RAFAEL MORALES GAMBOA.


(SETQ P-MIN '(-159 -99) )          ; Extremos inferior izquierdo y
(SETQ P-MAX '(159 99) )            ; superior derecho del puerto de graficacion.

(SETQ PCURSOR '(0 0) )             ; Coordenadas del cursor grafico
                                   ; (en el puerto de graficacion).


; Calcula el centro del puerto con extremos opuestos "p1" y "p2"

(DEFUN P-BISEC (P1 P2)
    (LIST (TRUNCATE (+ (COORD-X P1) (COORD-X P2) ) 2)
          (TRUNCATE (+ (COORD-Y P1) (COORD-Y P2) ) 2) ) )


; Define el puerto de vision  con extremos inferior izquierdo "p1" y
; superior derecho "p2". Actualiza la escala para la transformacion de
; coordenadas y la posicion de los cursores al centro del puerto y
; de la ventana.
(DEFUN PUERTO (P1 P2)
   (SETQ P-MIN P1)
   (SETQ P-MAX P2)
   (ESCALA)
   (SETQ VCURSOR (V-BISEC V-MIN V-MAX) )
   (SETQ PCURSOR (P-BISEC P1 P2) ) )


(SETQ V-MIN '(-159.0 -99.0) )      ; Extremos inferior izquierdo y
(SETQ V-MAX '(159.0 99.0) )        ; superior derecho de la ventana de vision.

(SETQ VCURSOR '(0.0 0.0) )         ; Coordenadas del cursor real
                                   ; (en la ventana de vision).


; Calcula el centro de la ventana con extremos opuestos "v1" y "v2".
(DEFUN V-BISEC (V1 V2)
   (LIST (/ (+ (COORD-X V1) (COORD-X V2) ) 2)
         (/ (+ (COORD-Y V1) (COORD-Y V2) ) 2) ) )


; Define la ventana de vision  con extremos inferior izquierdo "v1" y
; superior derecho "v2". Actualiza la escala para la transformacion de
; coordenadas y la posicion de los cursores al centro del puerto y
; de la ventana.
(DEFUN VENTANA (V1 V2)
   (SETQ Vu-MIN V1)
   (SETQ V-MAX V2)
   (ESCALA)
   (SETQ PCURSOR (P-BISEC P-MIN P-MAX) )
   (SETQ VCURSOR (V-BISEC V1 V2) ) )
```

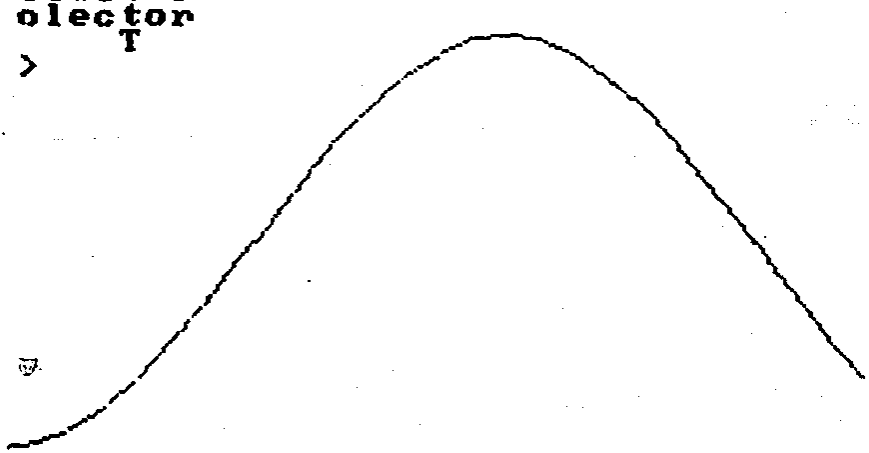
```
(DEFUN COORD-X (P-O-V)          ; Devuelve la primera coordenada (abscisa)
   (CAR P-O-V) )                ; de un punto (real o grafico).

(DEFUN COORD-Y (P-O-V)          ; devuelve la segunda coordenada (ordenada)
   (CADR P-O-V) )               ; de un punto (real o grafico).

(DEFUN PCURSOR-X ()             ; Devuelve la abscisa del cursor grafico.
   (COORD-X PCURSOR) )

(DEFUN PCURSOR-Y ()             ; Devuelve la ordenada del cursor grafico.
   (COORD-Y PCURSOR) )

(DEFUN VCURSOR-X ()             ; Devuelve la abscisa del cursor real.
   (COORD-X VCURSOR) )

(DEFUN VCURSOR-Y ()             ; Devuelve la ordenada del cursor real.
   (COORD-Y VCURSOR) )


; Calcula la escala para la transformacion de coordenadas entre la ventana
; y el puerto.
(DEFUN ESCALA ()
   (SETQ X-ESC (/ (- (COORD-X P-MAX) (COORD-X P-MIN) )
                  (- (COORD-X V-MAX) (COORD-X V-MIN) ) ) )
   (SETQ Y-ESC (/ (- (COORD-Y P-MAX) (COORD-Y P-MIN) )
                  (- (COORD-Y V-MAX) (COORD-Y V-MIN) ) ) ) )


(SETQ X-ESC 1.0)                ; Escala horizontal.
(SETQ Y-ESC 1.0)                ; Escala vertical.


; Calcula la escala horizontal
(DEFUN ESC-X (X)
   (ROUND (+ (* X-ESC (- X (COORD-X V-MIN) ) ) (COORD-X P-MIN) ) ) )


; Calcula la escala vertical.
(DEFUN ESC-Y (Y)
   (ROUND (+ (* Y-ESC (- Y (COORD-Y V-MIN) ) ) (COORD-Y P-MIN) ) ) )


; Calcula el cambio de coordenadas entre la ventana y el puerto.
(DEFUN ESC-XY (VPUNTO)
   (LIST (ESC-X (COORD-X VPUNTO) ) (ESC-Y (COORD-Y VPUNTO) ) ) )


(DEFUN MOV-ABS-P (PPUNTO)               ; Mueve el cursor grafico en
   (SETQ PCURSOR PPUNTO) )              ; coordenadas absolutas.

(DEFUN MOV-ABS-V (VPUNTO)               ; Mueve el cursor real en
   (SETQ VCURSOR VPUNTO) )              ; coordenadas absolutas.
```

```
; Dibuja un segmento de recta desde la posicion del cursor hasta un punto
; dado en coordenadas absolutas y actualiza el cursor. Utiliza coordenadas
; graficas, en el puerto.
(DEFUN LINE-ABS-P (PPUNTO)
   (LINE PCURSOR PPUNTO)
   (SETQ PCURSOR PPUNTO) )


; dibuja un segmento de recta desde la posicion del cursor hasta un punto
; dado en coordenadas absolutas y actualiza el cursor. Utiliza coordenadas
; reales, en la ventana.
(DEFUN LINE-ABS-V (VPUNTO)
   (LINE (ESC-XY VCURSOR) (ESC-XY VPUNTO) )
   (SETQ VCURSOR VPUNTO) )


; Mueve el cursor grafico en coordenadas relativas a la posicion actual
; del cursor.
(DEFUN MOV-REL-P (PPUNTO)
   (SETQ PCURSOR (LIST (+ (COORD-X PCURSOR) (COORD-X PPUNTO) )
                       (+ (COORD-Y PCURSOR) (COORD-Y PPUNTO) ) ) ) )


; Mueve el cursor real en coordenadas relativas a la posicion actual del
; cursor.
(DEFUN MOV-REL-V (VPUNTO)
   (SETQ VCURSOR (LIST (+ (COORD-X VCURSOR) (COORD-X VPUNTO) )
                       (+ (COORD-Y VCURSOR) (COORD-Y VPUNTO) ) ) ) )


; Dibuja un segmento de recta desde la posicion del cursor hasta un punto
; dado en coordenadas relativas y actualiza el cursor. Utiliza coordenadas
; graficas, en el puerto.
(DEFUN LINE-REL-P (PPUNTO)
   (LET ((PUNTO-AUX PCURSOR) )
     (LINE PUNTO-AUX (MOV-REL-P PPUNTO) ) ) )


; Dibuja un segmento de recta desde la posicion del cursor hasta un punto
; dado en coordenadas relativas y actualiza el cursor. Utiliza coordenadas
; reales, en la ventana.
(DEFUN LINE-REL-V (VPUNTO)
   (LET ((PUNTO-AUX VCURSOR) )
     (LINE (ESC-XY PUNTO-AUX) (ESC-XY (MOV-REL-V VPUNTO) ) ) ) )


; dibuja la grafica de una "funcion" en el intervalo ["vmin","vmax"]
; utilizando "npuntos" en el intervalo.
(DEFUN DIBGRAF (FUNCION VMIN VMAX NPUNTOS)
   (LET ((INC (/ (- VMAX VMIN) NPUNTOS) ) )
     (MOV-ABS-V (LIST VMIN (FUNCALL FUNCION VMIN) ) )
     (DO ((X VMIN (+ X INC) ) )
         ((> X VMAX) T)
       (LINE-ABS-V (LIST X (FUNCALL FUNCION X) ) ) ) ) )
```

```
> (dibgraf 'sin (- pi) pi 20)
    T
>
```



```
> (dibgraf 'cos (-pi) pi 30)
olector
olector
    T
>
```



El objetivo de este programa es resolver el problema de asignar valores numericos a las letras de una lista de palabras de tal manera que al sumar los valores respectivos de como resultado el asociado a la lista del ultimo renglon.Por ejemplo:

```
  S E N D
+
  M O R E
---------
M O N E Y
```

Este programa fue realizado por JOSE ANTONIO LOPEZ SAUCEDO, durante su curso de Inteligencia Artificial.

```
;
;
;       Definicion de Funciones
;
;
;   Con la siguiente funcion se obtiene el i-esimo elemento de la lista LISTA
;

(DEFUN TOMAI (LISTA I1)
   (COND ((= I1 1) (CAR LISTA) )
         (T (TOMAI (CDR LISTA) (- I1 1) ) ) ) )



;    La siguiente funcion regresa el elemento que se encuentra en el i-esimo
;    renglon y j-esima columna de la lista LISTA

(DEFUN TOMAIJ (LISTA I2 J2)
   (COND ((ATOM (CAR LISTA) ) (TOMAI LISTA I2) )
         (T (TOMAI (TOMAI LISTA I2) J2) ) ) )



;    La siguiente funcion regresa T o NIL dependiendo si el numero DIGITO
;    esta contenido o no, en la lista DISP .

(DEFUN DISPON (DIGITO DISP)
   (COND ((NULL DISP) NIL )
         ((= DIGITO (CAR DISP) ) T )
         (T (DISPON DIGITO (CDR DISP) ) ) ) )



;    La siguiente funcion regresa T o NIL dependiendo si la letra ITEM esta
;    o no la izquierda de la lista de palabras que forman a LISTA .

(DEFUN IZQUIERDA (ITEM LISTA)
   (COND ((NULL LISTA) NIL )
         ((EQUAL ITEM (CAR (CAR LISTA) ) ) T )
         (T (IZQUIERDA ITEM (CDR LISTA) ) ) ) )
```

```
;    La siguiente funcion regresa T o NIL dependiendo si la letra AUXILIAR
;    esta o no contenida en la lista LETRA2.

(DEFUN MIEMBRO (AUXILIAR LETRA2)
   (COND ((NULL LETRA2) NIL)
         ((NULL (MEMBER AUXILIAR LETRA2) ) NIL ) ) )


;    La siguientes dos funciones construyen una lista que contiene todas las
;    letras que forman a LISTA en orden,conforme van apareciendo.

(DEFUN CONSTR (LISTA)
   (DO-S ((LETR NIL)
          (AUX NIL)
          (J3 N1 ) )
         ((= J3 0) 'LETR )
       (DO-S ((I3 1) )
             ((= I3 (+ M1 2) ) T )
          (COND ((AND (= I3 (+ M1 1) ) (NULL CONDICION) )
                 (SETQ AUX (TOMAIJ LISTA I3 (+ J3 1) ) ) )
                (T (SETQ AUX (TOMAIJ LISTA I3 J3) ) ) )
          (COND ((NOT (MIEMBRO AUX LETR) )
                 (SETQ LETR (CONS AUX LETR) ) ) )
          (SETQ I3 (+ I3 1) ) )
          (SETQ J3 (- J3 1) ) ) )

(DEFUN CONSTRUYE (
   (SETQ LETRA (CONSTR LISTA) )
   (COND ((EQUAL CONDICION T) LETRA)
         (T (COND ((NOT (MIEMBRO (TOMAIJ LISTA  (+ M1 1) 1) LETRA) )
                        (CONS (TOMAIJ LISTA (+ M1 1) 1) LETRA) )
                  (T LETRA) ) ) ) )



;    La siguiente funcion regresa la suma de los valores asociados a las
;    letras que se encuentras en la j-esima columna de LISTA

(DEFUN SUMA (LISTA COLUMNA)
   (COND ((NULL (CDR LISTA) ) 0 )
         (T (+ (EVAL (TOMAI (CAR LISTA) COLUMNA) )
               (SUMA (CDR LISTA) COLUMNA) ) ) ) )



;    La siguiente funcion regresa la lista resultante de borrar el numero VAL
;    de la lista de numeros DISP.

(DEFUN DELETE (VAL DISP)
   (COND ((NULL DISP) NIL )
         ((NOT (= VAL (CAR DISP) ) ) (CONS (CAR DISP)
                                           (DELETE VAL (CDR DISP) ) ) )
         (T (CDR DISP) ) ) )
```

```
;     La siguiente funcion recorre la LISTA desde el renglon r-esimo columna
;     j-esima hasta llegar al final de la LISTA o hasta encontrar una letra
;     que no tenga un numero valido asociado.

(DEFUN RECORRE-R (LISTA R1 C1)
   (COND ((= R1 (+  M1 1) ) (+M1 1) )
         ((= (- 1) (EVAL (TOMAIJ LISTA R1 C1) ) ) R1 )
         (T (RECORRE-R LISTA (+ R1 1) C1) ) ) )



;     La siguiente funcion asocia a cada letra de la LISTA el valor de -1 como
;     marca para denotar que esa letra no tiene valor numero asociado valido.

(DEFUN DAVALOR ()
   (DO ((I1 1) )
       ((= I1 (+ M1 2) ) T )
     (DO  ((J1 N1) )
          ((ZEROP J1) T )
       (SET (TOMAIJ LISTA
                    I1
                    (COND ((= I1 (+ M1 1) ) (VALOR-C1 J1) )
                          (T J1 ) ) ) (- 1) )
       (SETQ J1 (- J1 1) ) )
       (SETQ I1 (+ I1 1) ) )
       (SET EXTRA  (- 1) ) )



;     Esta funcion regresa el valor correspondiente al valor correcto de la
;     columna en que se encuentra una letra en el ultimo renglon,ya que puede
;     suceder que el ultimo renglon tenga una letra mas que los anteriores.

(DEFUN VALOR-C1 (C1)
  (COND ((EQUAL CONDICION T) C1 )
        (T (+ C1 1) ) ) )


;     Esta funcion imprime los guiones necesarios para separar el ultimo renglon

(DEFUN ESCRIBE-LINEA ()
   (COND ((NULL CONDICION) (PRIN1 '-) (PRIN1 '-) (PRIN1 '-) ) )
   (DO ((I1 1) )
       ((= I1 (+ N1 1) ) T )
     (PRIN1 '- )
     (PRIN1 '- )
     (PRIN1 '- )
     (SETQ I1 (+ I1 1) ) )
     (TERPRI ) )
```

```
;     La siguiente funcion salta tantos renglones como lo indica NUM

(DEFUN SALTA-LINEAS (NUM)
   (COND ((ZEROP NUM) T )
         (T (TERPRI)
   (SALTA-LINEAS (- NUM 1) ) ) ) )



;     La siguiente funcion escribe la LISTA con todas sus letras y el valor
;     asociado a cada una de ellas.Extra es la letra que puede tener de mas el
;     ultimo renglon respecto a los demas.

(DEFUN ESCRIBE-LA-SOLUCION (LISTA EXTRA)
   (PRIN1 'UNA )
   (PRIN1 'SOLUCION )
   (PRIN1 'ES )
   (TERPRI )
   (DO ((AUX2 NIL)
        (I1 1 ) )
       ((> I1 (+ M1 2) ) T )
     (COND ((AND (NOT (EQUAL CONDICION T) )
                 (< I1 (+ M1 1) ) )
            (BLANKS 6) ) )
     (DO ((J1 1) )
         ((> J1 (+ N1 1) ) T )
       (SETQ AUX2 (TOMAIJ LISTA
                          I1
                          (COND ((> I1 (+ M1 1) ) (VALOR-C1 J1) )
                                (T J1) ) ) )
       (PRIN1 AUX2 )
       (PRIN1 '= )
       (PRIN1 (EVAL AUX2) )
       (SETQ J1 (+ J1 1) ) )
       (TERPRI )
       (COND ((= I1 M1) (ESCRIBE-LINEA) ) )
       (COND ((AND (EQUAL CONDICION NIL) (= I1 M1) )
                (PRIN1 EXTRA)
                (PRIN1 '=)
                (PRIN1 (EVAL EXTRA) ) ) )
       (SETQ I1 (+ I1 1) ) )
       (TERPRI ) )



;     La siguiente funcion define si una letra aparece antes que otra en la
;     lista de letras obtenidas ordenadamente.

(DEFUN ESTA-ANTES (ITEM LETT LETRAS)
   (COND ((EQUAL ITEM LETT) T )
         ((EQUAL LETT (CAR LETRAS) ) NIL )
         ((EQUAL ITEM (CAR LETRAS) ) T )
         (T (ESTA-ANTES ITEM LETT (CDR LETRAS) ) ) ) )
```

```
;       La siguiente funcion es lo mas importante ya que es lo que intenta
;       encontrar la solucion al problema dado,ademas de buscar una solucion
;       una vez que lo encuentra regresa al llamado anterior para buscar mas
;       soluciones y asi hasta encontrar todas,si es que las hay.)

(DEFUN INTENTA (LISTA DISPONIBLES R1 C1 SOBRE EXTRA)
   (COND ((AND (ZEROP C1)
               (OR (AND (EQUAL CONDICION T) (ZEROP SOBRE) )
                   (AND (NOT (EQUAL CONDICION T) )
                        (= SOBRE (EVAL EXTRA) ) ) ) )
          (ESCRIBE-LA-SOLUCION LISTA EXTRA) )
         ((AND (ZEROP C1)
               (NOT (EQUAL CONDICION T) )
               (= (EVAL EXTRA) (- 1) )
               (NOT (ZEROP SOBRE) )
               (DISPON SOBRE DISPONIBLES) )
          (SET EXTRA SOBRE)
          (ESCRIBE-LA-SOLUCION LISTA EXTRA) )
         ((NOT (ZEROP C1) ) (LET ((M2 0)
                                  (AUX2 NIL) )
                              (SETQ AUX2 (TOMAIJ LISTA R1 C1) )
                              (SETQ M2 (COND ((IZQUIERDA AUX2 LISTA) 1 )
                                             (T 0 ) ) )
                              (DO ((K1 M2) )
                                  ((= K1 10) AUX2 )
                                (COND ((DISPON K1 DISPONIBLES)
                                       (SET AUX2 K1)
                                       (BUSCA LISTA
                                              (DELETE K1 DISPONIBLES)
                                              R1
                                              C1
                                              SOBRE
                                              EXTRA)
                                       (SET AUX2 (- 1) )
                                       (DO ((J2 1)
                                            (VAR NIL) )
                                           ((= J2 (+ C1 1) ) T )
                                         (SETQ VAR
                                               (TOMAI (CAR (LAST LISTA) )
                                                      (VALOR-C1 J2) ) )
                                         (COND ((NOT (ESTA-ANTES VAR
                                                                 AUX2
                                                                 LETRAS) )
                                                (SET VAR (- 1) ) ) )
                                         (SETQ J2 (+ J2 1) ) )
                                       (SET EXTRA (- 1) ) ) )
                                (SETQ K1 (+ K1 1) ) ) ) ) ) )
```

```
;       La siguiente funcion es auxiliar de la anterior ya que si INTENTA
;       encontro,lo que puede ser parte de la solucion,entonces BUSCA se
;       encarga de buscar la siguiente letra que le falta asociar valor y
;       a la vez verifica que se vayan cumpliendo las condiciones del problema
;       si es asi continua intentando,de lo contrario regresa a INTENTA para
;       continuar con otros valores.

(DEFUN BUSCA (LISTA DISPONIBLES R1 C1 SOBRE EXTRA)
   (DO ((AUX NIL)
        (VALOR 0)
        (SALIR NIL) )
       ((EQUAL SALIR T) T )
     (COND ((NOT (ZEROP C1) ) (SETQ R1 (RECORRE-R LISTA R1 C1) ) ) )
     (COND ((= R1 (+ M1 1) )
            (SETQ AUX (TOMAIJ LISTA R1 (VALOR-C1 C1) ) )
            (SETQ VALOR (REM (+ (SUMA LISTA C1) SOBRE) 10) )
            (COND ((AND (= (- 1) (EVAL AUX) )
                        (DISPON VALOR DISPONIBLES) )
                   (SET AUX VALOR)
                   (SETQ DISPONIBLES
                         (DELETE VALOR DISPONIBLES) )
                   (SETQ SOBRE
                         (TRUNCATE (+ (SUMA LISTA C1) SOBRE) 10) )
                   (SETQ R1 1)
                   (SETQ C1 (- C1 1) ) )
                  ((= (EVAL AUX) VALOR) (SETQ R1 1)
                    (SETQ SOBRE
                          (TRUNCATE (+ (SUMA LISTA C1) SOBRE) 10) )
                    (SETQ C1 (- C1 1) ) )
                  (T (SETQ SALIR T) ) ) )
           (T (SETQ SALIR T)
              (INTENTA LISTA DISPONIBLES R1 C1 SOBRE EXTRA) ) ) ) )




;       La siguiente funcion sirve de paso intermedio para preparar todo para
;       que INTENTA busque la solucion.Entre otras cosas lee la LISTA a tratar,
;       la mide verifica si el ultimo renglon tiene un caracter mas que los
;       anteriores si es este el caso entonces a EXTRA le asocia tal letra,
;       les da valores iniciales a cada letra de la LISTA y define la lista de
;       los posibles valores que pueden tomar las letras .

(DEFUN RESUELVE ()
   (PRINT 'ESCRIBE-LA-LISTA)
   (SETQ LISTA (READ) )
   (SETQ M1 (- (LENGTH LISTA) 1) )
   (SETQ N1 (LENGTH (CAR LISTA) ) )
   (SETQ CONDICION (+ N1 (LENGTH (CAR (LAST LISTA) ) ) ) )
   (SETQ EXTRA (COND ((EQUAL CONDICION 1) 'NADA )
                     (T (CAR (CAR (LAST LISTA) ) ) ) ) )
   (DAVALOR)
   (SETQ DISPONIBLES '(0 1 2 3 4 5 6 7 8 9) )
   (SETQ LETRAS (REVERSE (CONSTRUYE LISTA) ) ) )
```

```
;    La siguiente funcion imprime la LINEA que pasa como parametro

(DEFUN IMPRIME-LINEA (LINEA)
   (COND ((NULL LINEA) T )
         (T (PRIN1 (CAR LINEA) )
   (IMPRIME-LINEA (CDR LINEA) ) ) ) )

;      La siguiente funcion imprime la presentacion cuando se empieza
;          a utilizar el programa.

(DEFUN PRESENTACION ()
   (SALTA-LINEAS 5)
   (IMPRIME-LINEA '(CON ESTE PROGRAMA SE PUEDEN RESOLVER PROBLEMAS COMO EL SIGUIENTE) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(SI TENEMOS LAS PALABRAS) )
   (SALTA-LINEAS 3)
   (BLANKS 22)
   (IMPRIME-LINEA '(S E N D) )
   (SALTA-LINEAS 1)
   (BLANKS 22)
   (IMPRIME-LINEA '(M O R E) )
   (SALTA-LINEAS 1)
   (BLANKS 20)
   (IMPRIME-LINEA '(M O N E Y) )
   (SALTA-LINEAS 5)
   (IMPRIME-LINEA '(DONDE LO QUE INTERESA CONOCER ES EL VALOR NUMERICO QUE LE PODEMOS) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(ASOCIAR A LAS LETRAS PARA QUE SUMANDO LOS VALORES RESPECTIVOS DE ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(CADA UNA DE ESTAS S E N D  Y  M O R E  NOS DE COMO RESULTADO EL ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(NUMERO QUE SE FORMA EN EL ULTIMO RENGLON M O N E Y) )
   (SALTA-LINEAS 5)
   (IMPRIME-LINEA '(PARA RESOLVER EL PROBLEMA PARTICULAR QUE USTED DESEE SOLAMENTE ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(QUE ESCRIBIR LAS PALABRAS QUE FORMAN DICHO PROBLEMA) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(POR EJEMPLO SI DESEASE RESOLVER EL PROBLEMA DEL EJEMPLO) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(TENDRIA QUE TECLEARLO CUANDO SE LO PIDA DE LA SIGUIENTE MANERA) )
   (SALTA-LINEAS 2)
   (PRIN1 '((S E N D )  (M O R E )  (M O N E Y ) ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(ES DECIR LA FORMA DE ESCRIBIR EL PROBLEMA ES DANDO CADA RENGLON) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(DEL MISMO ENTRE PARENTESIS Y A SU VEZ TODO EL PROBLEMA EN UN ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(PARENTESIS QUE LO CONTENGA DESPUES OPRIMA LA TECLA DE RETURN ) )
   (SALTA-LINEAS 2)
   (IMPRIME-LINEA '(Y ESPERE UN TIEMPO RAZONABLE PARA OBTENER LOS POSIBLES RESULTADOS) )
   (SALTA-LINEAS 20) )
```

```
:       Esta funcion llama a la presentacion,ejecuta la funcion resuelve
:       y comienza la busqueda de la o las soluciones, si es que existen.
:       al problema utilizando a la funcion intenta.

(DEFUN SEND ()
   (PRESENTACION)
   (RESUELVE)
   (INTENTA LISTA DISPONIBLES 1 N1 0 EXTRA7) )


:: ejecucion


(send)
```

CON ESTE PROGRAMA SE PUEDEN RESOLVER PROBLEMAS COMO EL SIGUIENTE

SI TENEMOS LAS PALABRAS

```
      S E N D
      M O R E
  M O N E Y
```

DONDE LO QUE INTERESA CONOCER ES EL VALOR NUMERICO QUE LE PODEMOS ASOCIAR A LAS LETRAS PARA QUE SUMANDO LOS VALORES RESPECTIVOS DE CADA UNA DE ESTAS S E N D Y M O R E NOS DE COMO RESULTADO EL NUMERO QUE SE FORMA EN EL ULTIMO RENGLON M O N E Y

PARA RESOLVER EL PROBLEMA PARTICULAR QUE USTED DESEE SOLAMENTE QUE ESCRIBIR LAS PALABRAS QUE FORMAN DICHO PROBLEMA
POR EJEMPLO SI DESEASE RESOLVER EL PROBLEMA DEL EJEMPLO
TENDRIA QUE TECLEARLO CUANDO SE LO PIDA DE LA SIGUIENTE MANERA
((S E N D) (M O R E) (MONEY))
ES DECIR LA FORMA DE ESCRIBIR EL PROBLEMA ES DANDO CADA RENGLON DEL MISMO ENTRE PARENTESIS Y A SU VEZ TODO EL PROBLEMA EN UN PARENTESIS QUE LO CONTENGA DESPUES OPRIMA LA TECLA DE RETURN
Y ESPERE UN TIEMPO RAZONABLE PARA OBTENER LOS POSIBLES RESULTADOS

```
ESCRIBE-LA-LISTA  > ((S E N D) (M O R E) (M O N E Y) )

UNA SOLUCION ES
      S = 9 E = 5 N = 6 D = 7
      M = 1 0 = 0 R = 8 E = 5
- - - - - - - - - - - - - -
M = 1 0 = 0 N = 6 E = 5 Y = 2

UNA SOLUCION ES
      S = 9 E = 6 N = 7 D = 5
      M = 1 0 = 0 R = 8 E = 6
- - - - - - - - - - - - - -
M = 1 0 = 0 N = 7 E = 6 Y = 1

UNA SOLUCION ES
      S = 5 E = 7 N = 3 D = 1
      M = 0 0 = 6 R = 4 E = 7
- - - - - - - - - - - - - -
M = 0 0 = 6 N = 3 E = 7 Y = 8

UNA SOLUCION ES
      S = 5 E = 7 N = 3 D = 2
      M = 0 0 = 6 R = 4 E = 7
- - - - - - - - - - - - - -
M = 0 0 = 6 N = 3 E = 7 Y = 9
```