UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

# UN INTERPRETE DE LISP PARA LA HEWLETT-PACKARD 1000

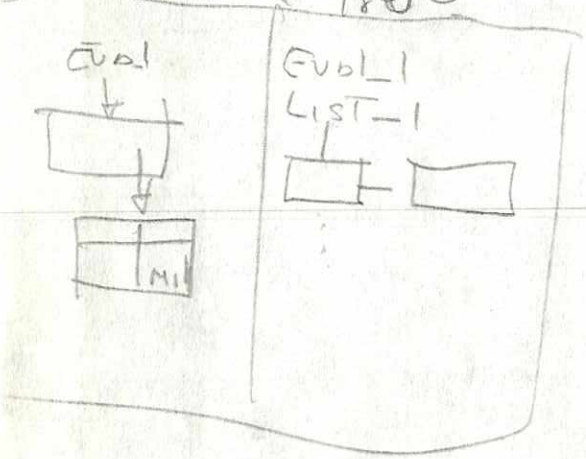
GABRIEL BASTERRECHEA DE LA VEGA

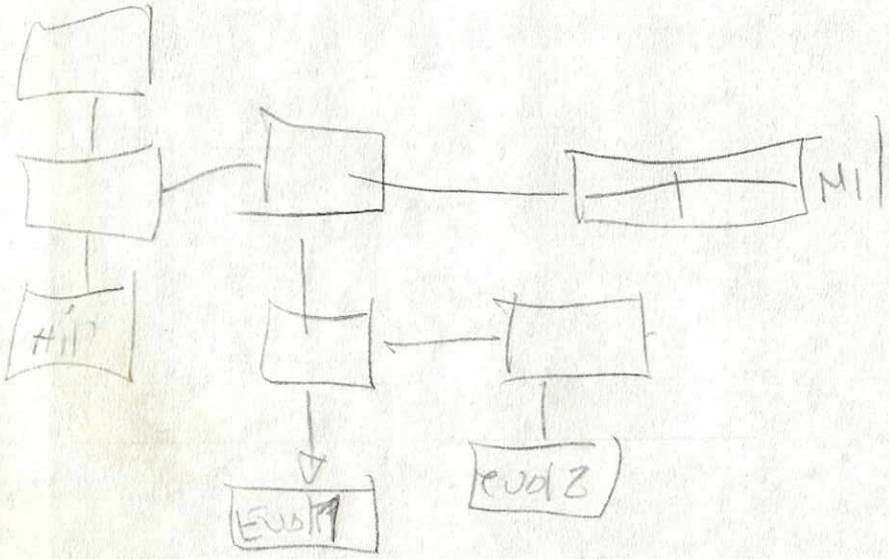


Guatemala

1,984

(COND ((EQUAL LIST-1 LIST-2) LIST-1)
(true LIST-2))



# UN INTERPRETE DE LISP PARA LA HEWLLETT-PACKARD 1000

UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

# UN INTERPRETE DE LISP PARA LA HEWLETT-PACKARD 1000

GABRIEL BASTERRECHEA DE LA VEGA

Trabajo de investigación presentado para optar el grado académico de

LICENCIADO EN CIENCIAS DE LA COMPUTACION

Guatemala

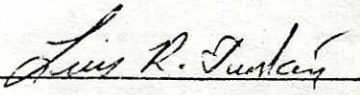
1,984

Vo. Bo. :

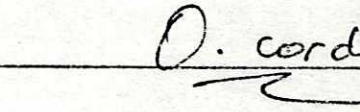
(f) 

Licenciado Fabian Pira
Asesor

Tribunal :

(f) 

(f) 

(f) 

Fecha de Aprobación :

A mis Padres

# PREFACIO

La inquietud de la realización de este proyecto, se remonta a la fecha en que por primera vez tuve conocimiento del lenguaje LISP (3er. Ciclo del año 1981), en virtud de que en aquella época, este conocimiento se basó únicamente en aspectos teóricos, sin tener la oportunidad de ponerlos en práctica.

En esa fecha recibimos en la Universidad el curso Organización de Lenguajes de Programación, curso en el que se hace un análisis a fondo de los principales lenguajes de programación existentes, así como el tipo de aplicaciones especiales para los cuales fueron diseñados. Entre estos lenguajes se encontraba LISP.

En esta oportunidad me llamó la atención la gran diferencia, en muchos aspectos ( los cuales mencionaré en detalle más adelante), que existía entre LISP y el resto de los lenguajes de programación antes estudiados. Prácticamente se salía de los patrones tradicionales que observé en los otros lenguajes, ya que, por medio de una sintaxis diferente facilitaba que el programador expresara sus ideas al computador de una forma más lógica, más parecida a la forma del pensamiento del hombre y no tan atada a la máquina.

Lastimosamente en aquella época no contabamos en la Universidad con un Intérprete de LISP para poder realizar prácticas directamente en el computador. La forma en que lo estudiamos consistía en que hacíamos nuestros programas de LISP en papel y el profesor se encargaba de hacer de intérprete de LISP para chequear si la lógica estaba correcta.

Fue en ese curso cuando pensé lo interesante que sería diseñar un Intérprete de LISP para la computadora de la Universidad.

Aunque no necesariamente fuera un LISP completo, sino que bastara con que se pudiera poner en práctica los conceptos de LISP que se estudian en el curso Organización de Lenguajes de Programación. (Quiero hacer notar que el objetivo de ese curso no es que el alumno salga hecho un experto en LISP, sino con los conocimientos suficientes en LISP para poder evaluarlo y poder así conocer en que tipo de aplicaciones y en que momento de su carrera profesional a él le convendrá utilizar este lenguaje.)

Luego en 1982, al tomar el curso Investigación en Ciencias de la Computación, decidí por que el Intérprete de LISP fuera el tema a desarrollar en este curso, así como mi Modelo de Trabajo Profesional previo a optar el título de Licenciado en Ciencias de la Computación.

A grandes rasgos las fases principales en el desarrollo de éste proyecto son las siguientes :

1. Recopilación y estudio de bibliografía sobre el diseño y construcción de compiladores e intérpretadores.
2. Recopilación y estudio de bibliografía sobre el lenguaje de programación LISP.
3. Análisis y estudio de los distintos algoritmos y estructuras de datos (máquina virtual) más convenientes para la implementación de la relación entre fases 1 y 2 o sea el Interpretador de LISP.
4. Implementación de los algoritmos y estructuras de

datos obtenidos en la fase 3 por medio de un lenguaje de programación conveniente.

Después de haber considerado los distintos lenguajes de programación con que contaba el computador de la Universidad Hewlett-Packard 1000 decidí que el que más convenía era el ALGOL, y fue por medio de éste con el que empecé y con el que hice gran parte del trabajo, sin embargo a medida que el trabajo iba creciendo, el ALGOL llegó a su límite (ya que es muy pequeño) y sus tablas se llenaron. Por lo que tuve que escoger otro lenguaje lo suficientemente estructurado para que el paso del código que había hecho en ALGOL resultara más fácil, y fue así como escogí el RATFOR el cual es un precompilador de FORTRAN que contiene la mayoría de intrucciones de un lenguaje bien estructurado. Y es en este lenguaje en el que está escrito el Interpretador de LISP, con algunas rutinas de FORTRAN.

5. Implementar las funciones básicas de LISP.
6. Pruebas hasta que el Interprete estuviera bien afinado.
7. Documentación del proyecto.

Creo que mi objetivo fue alcanzado, este Intérprete puede ser de gran utilidad para alguna persona que desée empezar a dar sus primeros pasos en el interesante mundo de LISP. Al mismo tiempo fue diseñado de una forma para que pueda ser complementado en un futuro por otro estudiante y así lograr tener un LISP bastante poderoso en la Universidad.

En vista que existen términos en Inglés que al traducirlos al Español pierden su significado. Y ya que estos

términos son comunes para la gente que tiene relación en el campo de la Computación, he decidido poner entre comillas y subrayar estos términos y agregar al final un Glosario que los explican.

Si alguna persona desea obtener los fuentes de los programas, éstos se encuentran con el Jefe del Departamento de Ciencias de la Computación en la Universidad, en este caso con el Ing. Luis Furlán.

Por último, quiero mencionar que este trabajo no lo hubiera podido haber terminado con éxito, de no haber tenido la colaboración tanto por sus conocimientos, como la visión de haberme podido guiar en el desarrollo de todas las fases, desde la fase de recopilación de bibliografía pasando por las de análisis y desarrollo hasta llegar a la de escribir este informe, al Lic. Fabián Pira. Al mismo tiempo de no haber tenido el apoyo para el uso del computador del Ing. Luis Furlán.

Para ellos, mi sincero agradecimiento.

Grabiel Basterrechea de la Vega

CONTENIDO

LISTA DE FIGURAS

## RESUMEN

Este escrito es la culminación del desarrollo de un trabajo cuyo principal objetivo fue la realización de un Modelo de Trabajo Profesional como lo es la creación de un Intérprete.

En ningún momento se pretendió, debido a la cantidad de horas/hombre que involucraría así como la experiencia profesional necesaria, que éste fuera un Intérprete de LISP completo, sino simplemente, un modelo lo suficientemente poderoso para que pudiera ser de utilidad en la educación de este maravilloso lenguaje.

Por los resultados obtenidos considero que este objetivo fue alcanzado. A continuación describiré en forma resumida el desarrollo de este proyecto.

En el Capítulo I he tratado de explicar las características principales que hacen de LISP un lenguaje completamente distinto a los demás, y las ventajas del uso de éstas en los campos de: Inteligencia Artificial, manipulación simbólica, programación funcional y otros.

En el Capítulo II ya se habla con más detalle de lo que consiste el lenguaje, sus diversos tipos de instrucciones y se

da una descripción del lenguaje en forma teórica.

Este capítulo es de gran ayuda para aquellas personas que no conocen el lenguaje y empiezan a familiarizarse con la terminología de LISP.

En los Capítulos III y IV se mencionan las estructuras de datos que fueron utilizadas para la creación de la Máquina Virtual sobre la cual se ejecutaría el Intérprete, así como los distintos algoritmos para la manipulación de las instrucciones en LISP, las cuales son los datos de entrada al sistema.

Por último en el Capítulo V se muestran los resultados obtenidos en el desarrollo del trabajo.

# I. INTRODUCCION

Desde los años en que las primeras computadoras empezaron a surgir hasta la fecha, han habido muchos cambios en el campo de la Computación tomando en consideración el corto tiempo que ha transcurrido, aproximadamente 40 años. Estos cambios son muy notorios tanto en los dispositivos físicos -HARDWARE- como en los distintos programas que facilitan al hombre que estos dispositivos físicos hagan lo que nosotros querramos que hagan -SOFTWARE-.

En la década de los cuarentas la forma en que el hombre se hacía entender por la computadora era bastante complicada y difícil ya que el programador tenía que conocer de una manera exacta y detallada la interrelación de los distintos componentes físicos del computador ya que la forma en que se le daban instrucciones era por la conexión o desconexión de alambres y otros dispositivos eléctricos.

Ante esta dificultad a finales de la década de los cuarentas y principios de la década de los cincuenta se fueron creando programas con el fin de que ésta comunicación no implicara el conocimiento profundo de dispositivos eléctricos, y que ésta se adaptara más al lenguaje humano. (Lenguaje de Programación) Así, la función de estos programas consiste en traducir las instrucciones de un lenguaje de programación a un lenguaje que pudiera ser reconocido por la máquina (Lenguaje de Máquinas).

A estos programas se les llama Traductores y es así como se han ido creando una variedad de éstos, entre otros,COBOL, FORTRAN, ALGOL, PL/1, RPG II, BASIC. Cada lenguaje de éstos fue creado para aplicaciones específicas ya sea comerciales como COBOL, científicas como FORTRAN, comerciales y científicas como PL/1 de simulación como SIMULA y GPSS, en educación como PILOT, etc.

A principios de la década de los sesentas John McCarthy del Massachusetts Institute of Technology (M.I.T.) creó el lenguaje de Programación llamado LISP (LISt Processing). Su objetivo era crear una notación poderosa para definir y transformar funciones, así como, para manipular símbolos de una manera más fácil y eficiente de la que hasta ese momento le brindaban los otros lenguajes de programación.

El poder expresivo del lenguaje fué reconocido por un pequeño número de investigadores quienes estaban principalmente interesados con la dificultad de los problemas para la manipulación simbólica en el campo de Inteligencia Artificial. Simultáneamente, el número de estudiantes que realizaba trabajos avanzados en la ciencia de la computación concentró su interés académico sobre LISP. Cursos sobre lenguajes de programación (como I2 en el Curriculum 68 de la ACM) se convirtió en el curso principal que escogían los estudiantes graduados y no graduados en ciencias de la computación.

En 1972 en una junta para el diseño de un curriculum en una Universidad de los Estados Unidos, un profesor de Análisis Numérico, Dr. Richard Bortels, expresó su opinión:"Un estudiante con el grado de Licenciado en Ciencias de la Computación que no haya aprendido LISP está culturalmente perdido" (The Little LISPer. D.P. Friedman Introducción).

Lo anterior se debe a que si bien es cierto, LISP no es un lenguaje que se use en aplicaciones comerciales, que son las más comunes, si es usado y es el más eficiente para las aplicaciones en el campo de la Inteligencia Artificial, más específicamente en: Operaciones simbólicas en el cálculo diferencial e integral, teoría de circuitos eléctricos, lógica matemáticas, teoría de juegos y en otros.

Hoy en día se ha notado un crecimiento bastante grande de programas que muestran lo que la mayoría de la gente llama comportamiento inteligente. Lo interesante es que la mayoría de todos estos programas inteligentes o que parecen ser inteligentes son escritos en LISP. Consecuentemente, una persona que quiera entender algo sobre la inteligencia de las computadoras, necesita en algún punto conocer LISP, si desea que su conocimiento sea completo.

Por otro lado, hoy en día existen muchos lenguajes de programación. Afortunadamente, sin embargo, solamente unos pocos son para la manipulación de símbolos, y de éstos LISP es el más usado. Después que LISP es aprendido la mayoría de los otros lenguajes para el manejo de símbolos son fácilmente comprendidos. Felizmente, LISP es un lenguaje fácil de aprender, no siendo necesaria una previa comprensión de otros lenguajes. Por el contrario algo de esta experiencia puede ser de ayuda para el desarrollo de un "Mal acento" en LISP ya que otros lenguajes hacen cosas muy distintas y una traducción función por función deja construcciones muy anticuadas.

A continuación explicaré tres de las características principales que posee LISP:

## A. LISP un lenguaje puramente funcional.

En programación existen varios estilos para escribir algoritmos, entre estos estilos se encuentra uno que se basa en el uso de funciones como operandos. Hay varios lenguajes de programación que facilitan el desarrollo de este estilo siendo LISP el prototipo de todos éstos.

La mayoría de los lenguajes de programación convencionales (podríamos decir no funcionales)tienen un nivel de atadura entre los requerimientos de la máquina y los requerimientos del usuario bastante marcado.

Es decir tienen cierto "compromiso" con la máquina para que los programas corran más rápido y cierto "compromiso" con el usuario para que sean mas fáciles de escribir.

Los lenguajes de programación funcionales tienen la característica de que hacen un "compromiso" menor con la máquina proveyendo un alto nivel de interfase con el usuario.

A estos lenguajes se les conoce como puramente funcionales porque el bloque principal en la construcción de programas es la noción de función. Estos lenguajes son los más estudiados cuando se ven cursos de lenguajes de muy alto nivel. Los programas que podrían ser largos y difíciles de escribir en lenguajes convencionales de alto nivel pueden a menudo ser escritos mas cortos y más claros en una forma funcional. El precio que se paga es que la implementación de los programas hace un uso menos eficiente del hardware, sin embargo con los adelantos de la tecnología, ésta pérdida de eficiencias va teniendo menos importancia, impulsando así un uso mayor en el futuro.

A lo largo del trabajo se irán mostrando varios programas en LISP lo que dará una idea bastante clara de la estructura de un lenguaje de programación funcional, sin embargo a continuación mostraré algunos ejemplos (no en LISP) de la estructura de un lenguaje funcional.

Empezando por las funciones aritméticas tenemos que estamos acostumbrados en matemáticas y en programación, a escribir expresiones algebráicas que involucran operadores aritméticos, tales como:

1. A + B x C
2. ( 2 x X + 3 x Y) / (X - Y)

Por supuesto, los operadores +, -, X, Y / son funciones, cada uno tiene como dominio parejas ordenadas de números reales. Nosotros podemos hacer esta observación explicitamente definiéndolas como funciones:

SUMA (X, Y) = X + Y
REST (X, Y) = X - Y
MULT (X, Y) = X x Y
DIVI (X, Y) = X / Y

Usando estos nombres de funciones en vez de operadores las expresiones algebráicas anteriores se convierten en:

SUMA (A, MULT (B,C))
DIVI (ADD (MULT (2,X), MULT (3,Y)), REST (X,Y))

Las expresiones escritas en esta forma están en un estilo funcional.

B. LISP, un lenguaje diseñado fundamentalmente para la manipulación de datos simbólicos.

Para una mejor comprensión de la característica principal de LISP, la manipulación de datos simbólicos, es necesario que quede muy claro el concepto de este tipo de datos.

## 1. Datos simbólicos.

Se identifica a los tipos de expresiones simbólicas llamándolas Expresiones-S y, hablando en notación funcional, forman los datos del dominio de los programas funcionales como LISP. Cada una de las siguientes tres líneas contienen Expresiones-S:

```
(JUAN PEREZ TIENE 33)
(( DAVID 17) (MARIA 24) (ELIZABETH 6))
((MI CASA) TIENE (GRANDES (PUERTAS CAFES)))
```

La propiedad común obvia que se nota en estos ejemplos es el uso de paréntesis. Realmente, el uso de paréntesis en Expresiones-S es absolutamente fundamental. Si la posición, o el número de paréntesis es alterada en cualquier forma, entonces la estructura y consecuentemente el significado de las Expresiones-S será cambiado.

Aparte de los paréntesis en los ejemplos anteriores, cada Expresión-S está construída de partes elementales llamados Atomos. Los siguientes son ejemplos de átomos:

| | |
|---|---|
| JUAN | -127 |
| PEREZ | MICATALOGO |
| 33 | MANZANA 13 |
| MI | |

Los átomos son de dos tipos: numéricos o simbólicos.

Un átomo simbólico es usualmente una secuencia de letras, aunque puede contener otros caracteres, incluyendo dígitos, con la condición de que el primer caracter sea una letra. Un átomo simbólico se considera que es indivisible. Más bien la única operación que se puede ejecutar sobre átomos simbólicos es comparar dos para saber si son iguales o diferentes.

Un átomo numérico es una secuencia de dígitos, posiblemente precedidos por un signo. En general las Expresiones-S contienen una mezcla de átomos simbólicos y numéricos.

La forma más simple de una Expresión-S es un Atomo. Una forma más elaborada de Expresión-S es una Lista de Atomos. Esta es formada escribiendo los átomos en una secuencia y circundando la secuencia en un conjunto de paréntesis. Asi:

```
(JUAN PEREZ TIENE 33)
```

es una secuencia de cuatro átomos

```
(JUAN PEREZ COME
   PAN Y MANTEQUILLA
     MUY A MENUDO)
```

es una lista compuesta de 9 átomos, todos simbólicos

```
(14 GRADOS Y 50 MINUTOS)
```

es una lista de 5 átomos simbólicos y numéricos

```
(BANANO)
```

es una lista de 1 átomo

( B A N A N O )

es una lista de 6 átomos simbólicos.

Expresiones-S significativamente más elaboradas pueden ser construídas cuando nosotros hacemos que los elementos de una lista no sean únicamente átomos, sino Expresiones-S en sí.

Así:

((DAVID 17) (MARIA 24) (ELIZABETH 6))

es una Expresión-S porque es una lista de tres elementos, cada uno de los cuales es en sí una Expresión-S.

Las reglas generales para construir Expresiones-S pueden ser sumarizadas en la siguiente definición recursiva:

1. Un átomo es una Expresión-S
2. Una secuencia de Expresiones-S encerradas en paréntesis es una Expresión-S (una lista).

Cuando escribimos un programa probablemente la primera consideración que se debe hacer es como nuestros datos se acomodan al formato de las Expresiones-S. Considere estos ejemplos:

Suponga que nosotros queremos escoger una representación de Expresiones-S para fórmulas algebráicas simples. Aquí nosotros reconocemos que las fórmulas estan hechas de constantes, variables y operadores binarios. Vamos a representar las constantes y variables por átomos, y colocar un operador binario en la primera posición en una Lista seguida por sus operandos.

En el futuro, escogeremos nombres simbólicos particulares para los operadores.

Entonces, la forma en que las fórmulas quedan representadas por Expresiones-S son las siguientes:

| Fórmula | Expresión-S |
|---|---|
| Constante | Número |
| Variable | Símbolo |
| p + q | (SUM P Q) |
| p - q | (RES P Q) |
| p X q | (MULT P Q) |
| p / q | (DIV P Q) |
| $p^q$ | (EXP P Q) |

Así en el caso de:

2 x Y + 2

tiene la forma p1 + q1 donde p1 =2xY y q 1=2, y p1 tiene la forma p2 x q2 donde p2=2 y q2=y, entonces representamos la fórmula por:

(SUM(MUL 2 Y) 2)

En una forma similar podemos ver que,

$X^2$ + 2X - 31

puede ser representado por:

(SUM(EXP X 2) (RES(MUL 2 X) 31))

También por,

(RES(SUM(EXP X2) (MUL 2X)) 31)

y muchas otras.

Como un segundo ejemplo, considere un programa para un juego de ajedrez. Primero, necesitamos denotar cada pieza por símbolos de su valor y color, por ejemplo (REY BLANCO) y (PEON NEGRO). Después, necesitamos denotar la posición de cada pieza. Podríamos escoger representar las coordenadas por parejas de enteros (asumiendo las filas y columnas del tablero). Entonces varios cuadros pueden ser representados por (4 7), ( 2 1) etc. Así el tablero puede ser representado haciendo una Lista de parejas por pieza y posición, así:

```
( ( (REY BLANCO)    (4 7) )
  (REY NEGRO)       (2 1) )
  (PEON BLANCO)     (3 3) )
  (ALFIL NEGRO)     (8 7) ) )
```

Estos datos simbólicos son los tipos de datos que maneja LISP. Más adelante veremos que tipo de funciones son construídas en LISP para la manipulación de estos datos.

## C. LISP, un lenguaje usado en el campo de la Inteligencia Artificial

En vista de que la principal aplicación de LISP se usa en el campo de la Inteligencia Artificial, es necesario comentar algunos aspectos importantes en este campo para una mejor apreciación del lenguaje.

La idea de que las computadoras digitales algun día igualarán o exederán las habilidades intelectuales del hombre ha sido comentada repetidamente desde que la computadora fue inventada, aunque no se ha contado con bases firmes para poder confirmarlo.

Sin embargo en los últimos 20 ó 25 años la nueva disciplina de la Inteligencia Artificial ha dado lugar a algunas de las más ingenuas analogías entre el computador y el cerebro humano y ha empezado la tarea de colocar el concepto de inteligencia sobre bases teóricas.

Por ejemplo, considerando el aprendizaje en las personas, este parecería estar íntimamente relacionado al crecimiento y cambios en la estructura física del cerebro. Pero el "Hardware" de un computador no pasa por estos cambios, sin embargo sus programas pueden cambiar en una variedad de formas; pueden acumular datos, organizarlos y aún modificarse a sí mismos (características muy importantes en LISP) Así trabajadores en Inteligencia Artificial han sugerido que programas complejos y particularmente aquellos que pueden modificar sus propias operaciones, pueden ser buenos modelos del aprendizaje humano.

Tales modelos, como otros modelos en la ciencia, no duplican el fenómeno que ellos están representando en todo su detalle. Por ejemplo, las operaciones del computador que involucran un programa de "aprendizaje" no mantiene una relacion en la forma como las células del cerebro y de los nervios lo hacen.

Sin embargo el modelo de computador de la inteligencia es lo suficientemente flexible para ser programado a un nivel de detalle considerado apropiado para realzar las funciones esenciales del pensamiento humano.

En vista de la gran cantidad de aplicaciones que se pueden mencionar en el campo de la Inteligencia Artificial, únicamente se hará mención de una de ellas, pero que dará una idea bastante clara de la forma como el computador es usado en este campo.

Uno de los temas constantes en la Inteligencia Artificial trata de como explotar un rango de posibles acciones en la persecusión de metas bien definidas. En general, para cada acción tomada, nuevas posibles acciones se vuelven disponibles, y así el planeamiento de una secuencia de acciones debe ser considerada como una estructura ramificada de estados posibles llamada en computación como Búsqueda en Arbol. La raíz es la situación actual, las ramas son las posibles acciones y los extremos de las ramas son las posibles salidas. A esto se le conoce con el nombre de "Backtracking".

En muchos casos sin embargo, la búsqueda en el árbol es tan larga que un rastreo completo no es posible. Por eso la mayoría de los programas en la Inteligencia Artificial utilizan los principios heurísticos, los cuales dan una solución que descansa en reglas empíricas o intitutivas, permitiendo la determinación de una solución mejorada (sin ser la óptima) Es decir, los procedimientos de búsqueda heurística se mueven inteligentemente de un punto solución a otro, con el objetivo de mejorar el valor de la solución.

Cuando mejoras futuras no pueden ser alcanzadas, la mejor solución hasta ese momento es la solución "aproximada" al modelo, sin garantizar que la solución óptima haya sido alcanzada.

Programas que ejecutan juegos tales como el ajedrez, damas y "backgammon" son ejemplos de búsquedas guiadas heurísticamente. Muchos de los juegos que ahora se realizan a nivel de expertos o jugadores maestros han sido derrotados por el computador; un programa llamado Mighty Bee, escrito por Hans Berliner de Carnegie Mellon University derrotó al campeón del mundo de "backgammon" en 1979. Probablemente el mayor interés se centra en los programas que juegan ajedrez y virtualmente todos los programas que juegan ajedrez desde los cincuentas se basan en modelos de búsqueda heurística desarrollado por Claude E. Shannon en los laboratorios Bell.

En una forma breve y sin entrar en detalles, pero por medio de la cual el lector comprenderá como el computador elige sus movimientos, se explicará como el programa revisa sus opciones heurísticamente.

Cuando un programa de ajedrez está escogiendo un movimiento, el programa debe de alguna forma, evaluar la posición que resultaría de cada movimiento posible. La evaluación puede ser hecha por medio de un sistema de punteo basado en dos factores principales, primero, dándole un valor a cada pieza a favor y segundo, pesando la gravedad de amenaza a cualquier pieza que pueda ser capturada en el movimiento siguiente.

Como en la figura siguiente:

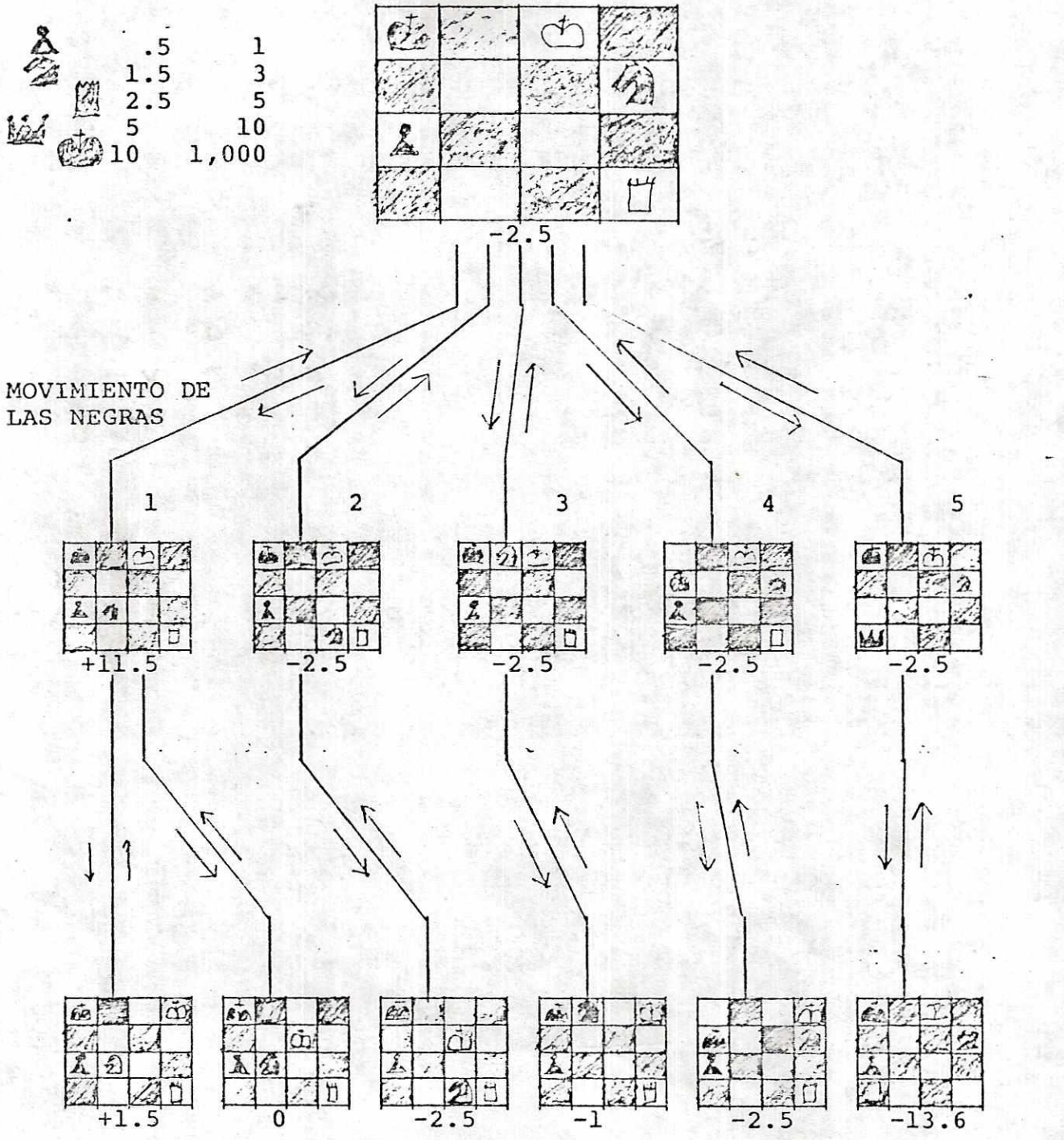



Fig. 1.1 Forma en que un programa evalúa las posiciones en Ajedrez

Programas más sofisticados también incorporan factores como la resistencia de los peones, el grado de control sobre el centro del tablero y el número de piezas movilizadas. La meta del programa es entonces maximizar el puntaje.

Ahora bien, ¿Hasta qué nivel de profundidad el programa dejará de analizar todas las posibles opciones? Es aquí donde se ponen en práctica los principios heurísticos.

Los principios heurísticos incorporados en el programa de ajedrez controlan la amplitud y la profundidad de la búsqueda en el árbol, el programa llega siempre a un nivel específico en busca del mejor movimiento. Generalmente calcula el puntaje para cada movimiento legal de una posición dada. El programa puede entonces considerar la respuesta del oponente para cada uno de sus movimientos.

Sin embargo, al principio del proceso, es preciso definir el nivel que alcanzará la evaluación o el número de posiciones a ser evaluadas se convertirán en imprácticos. Esto se logra haciendo uso de los principios heurísticos. Estos principios son definidos por el programador al inicio del programa.

Así,las Negras hacen el movimiento a la posición que les de el puntaje más alto, asumiendo que las Blancas hacen el mejor movimiento disponible en cada turno.

En el ajedrez el promedio de número de movimientos que pueden ser hechos en una posición es de 35; una exaustiva búsqueda únicamente de tres niveles de profundidad por cada jugador, podría requerir el chequeo de más de 1,800 millones de movimientos. Actualmente, rigurosos métodos para retener la búsqueda en el árbol son utilizados.

El actual campeón del mundo entre programas se llama Belle, desarrollado por Ken Thompson y Joe Condon de los Laboratorios Bell. El programa se ejecuta en un computador con "Hardware" específicamente diseñado para hacer cálculos de ajedrez; revisa a un promedio de 160,000 posiciones por segundo. Belle juega en torneos con un promedio de 2,160 (un promedio entre 2,000 a 2,199 califica a un jugador como experto).

Este programa muestra como la computadora simula el comportamiento de la inteligencia humana al nivel de los jugadores expertos de ajedrez. Existen muchos otros programas que hacen este tipo de simulaciones en el campo de la Inteligencia Artificial. Lo importante aquí, es que el lenguaje más usado para este tipo de aplicaciones es el LISP.

# II. DESCRIPCION DEL LENGUAJE

Cualquier programa, sin tomar en consideración el lenguaje que se use, puede ser examinado especificando un conjunto de operaciones que van ha ser aplicadas a ciertos datos en alguna secuencia. Las diferencias básicas entre los lenguajes está en los tipos de datos permitidos, en los tipos de operaciones disponible, y en los mecanismos provistos para controlar la secuencia en la cual las operaciones son aplicadas a los datos. Estas tres áreas -datos, operaciones y control definen las características principales de cada lenguaje.

En el caso de LISP existen varios dialectos, este trabajo se basó en el dialecto conocido con el nombre de INTERLISP (Mac Carthy, John, 1962).

Jean Sammet dijo: "Los lenguajes de programación pueden ser divididos en dos categorías. En una está LISP y en la otra los demás lenguajes de programación." LISP se encuentra en otra categoría por las grandes diferencias que tiene con los otros lenguajes en las tres áreas antes mencionadas, y por las siguentes carcaterísticas especiales:

-.La equivalencia en la forma entre programas y datos, lo cual permite a las estructuras de datos ser ejecutadas como programas y a los programas se modificados como datos.

-.La gran utilización como estructura de control, de la recursión, en vez de la iteración, la cual es común en la mayoría de los otros lenguajes.

-. El uso de listas encadenadas como la estructura de datos básica, junto con las operaciones de modificación de las listas.

El procesamiento de listas es la base de la mayoría de los algoritmos en LISP.

A. Los datos en LISP

Una de las formas en las que LISP difiere de la mayoría de los demás lenguajes de programación, está en la naturaleza del dato. En LISP, todos los datos están en la forma de expresiones simbólicas usualmente referidas como Expresiones-S. Las Expresiones-S son de un tamaño indefinido teniendo un tipo de estructura en árbol, de tal forma que las sub-expresiones pueden ser fácilmente aisladas. En el sistema de programación de LISP, la mayor parte de memoria disponible es usada para almacenar Expresiones-S en la forma de estructuras de lista. Este tipo de organización de la memoria libera al programador de distribuir almacenamiento para las diferentes secciones de su programa.

1. Expresiones Simbólicas. El tipo más elemental de expresiones simbólicas (Expresiones-S) es el Símbolo Atómico (mejor conocido como Atomo). Definición: Un símbolo Atómico es una agrupación de no más de 30 letras y números; en donde el primer caracter debe ser una letra.

Ejemplos:

a) A
b) MANZANA
c) PARTE2
d) STRINGEXTRALARGODEFLETE
e) A4B66XYZ2

Estos símbolos son llamados atómicos ya que son tomados como un todo y no pueden ser divididos en caracteres individuales.

Todas las Expresiones-S son construídas por Símbolos Atómicos y los signos de puntuacion "(",")". La operación básica para formar Expresiones-S es combinando dos de ellas para producir una más grande. Así, de los dos Símbolos Atómicos A1 y A2 se puede formar la Expresión-S (A1 A2).

Definición: Una Expresión-S puede ser un Símbolo Atómico o la composición de estos elementos, en el siguiente orden: Un paréntesis izquierdo, una Expresión-S y un paréntesis derecho:

Note que la definición es recursiva.

Algunos ejemplos son:

a) ATOM
b) (A B)
c) (A (B C)
d) ((A1 A2) B)
e) ((U V) (X Y))
f) ((U V) (X (Y Z)))
g) ( (ESTÀ ES UNA)
   ( ( EXPRESION ) (S) ))

La forma como son representadas las Expresiones-S en el computador se conocen como Listas Encadenadas. Cada elemento de estas listas tiene dos señaladores, uno que señala al elemento de dato (Un elemento de dato puede ser: un átomo o una lista) y otro que señala al elemento que sigue en la lista. El último elemento de la lista señala a un átomo especial llamado NIL, como su sucesor. A estos dos señaladores se les llama CAR y CDR. El CDR señala al sucedor del elemento en la lista y el CAR señala al elemento de datos. (Los términos CAR y CDR se originan de la organización del "Hardware" de la primera computadora en la cual se implementó el Lisp).

Un ejemplo de una Expresión-S representada por una lista es:

Sea la Expresión-S: (A B C D)

Esta Expresión-S es representada en la memoria del computador en la forma siguiente:

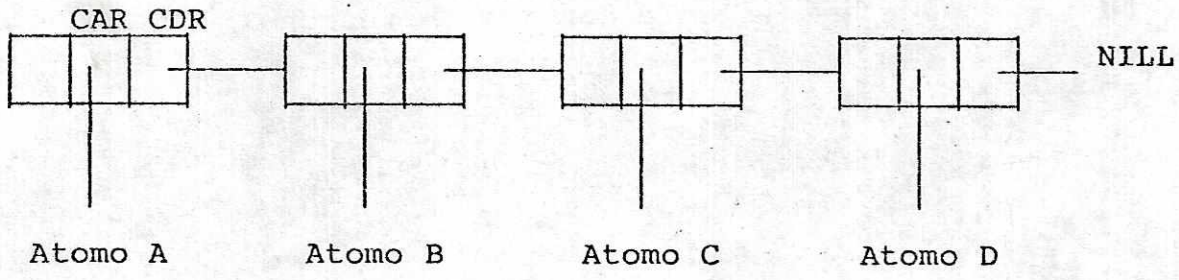



Fig. 2.1 Representación de la Expresión-S (A B C D)

Ya que el señalador CAR puede señalar a otra lista, es posible construir estructuras de listas de una complejidad arbitraria. Ordinariamente estas son estructuras de árbol.

Por ejemplo, la Expresión-S siguiente:

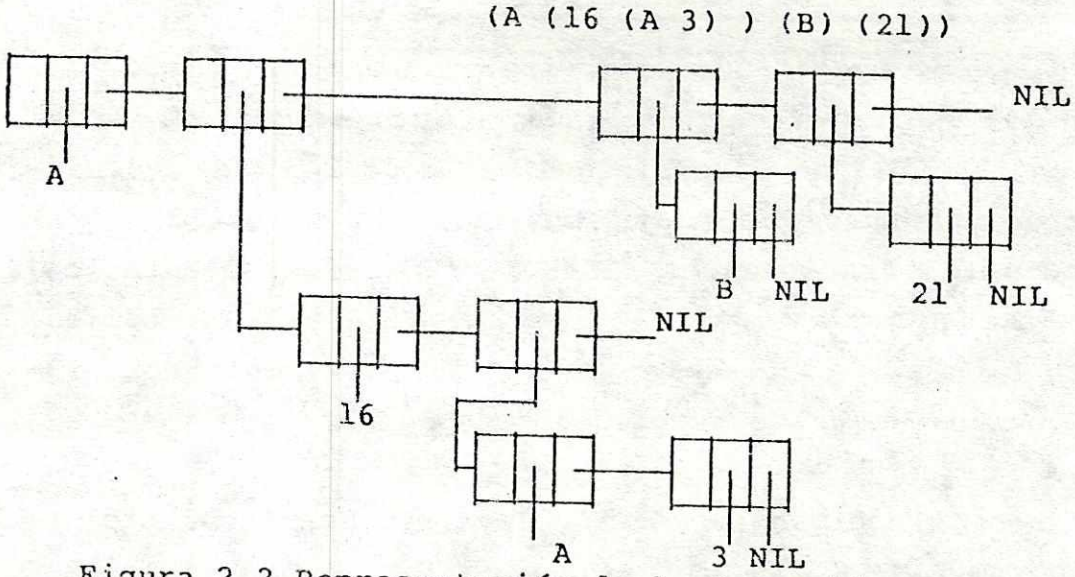



Figura 2.2 Representación de la Expresión-S

(A (16(A3)) (B) (21))

Cada átomo tiene una Lista de Propiedades asociada y accesible a través de un señalador almacenado en la dirección de la memoria que representa al átomo (Tabla simbólica). Una lista de propiedades es una lista ordinaria de LISP, diferenciándose únicamente en que sus elementos están apareados lógicamente es una secuencia alterna de Valor de Propiedad-Nombre de Propiedad. Cada Lista de Propiedades de un átomo contiene por lo menos la propiedad conocida como PNAME cuyo valor asociado es un apuntador a una lista que contiene el nombre de impresión del átomo en forma de una cadena de caracteres. Si un átomo es el nombre de una función su Lista de Propiedades contiene el Nombre de Propiedad, EXPR, y un señalador a la definición de la función.

2. Programas. Las rutinas de entrada de LISP no hacen distinción entre programas y listas de datos, sino que simplemente trasladan todas las listas a una representación interna de listas encadenadas.

Cada átomo encontrado se revisa en una tabla simbólica, durante la ejecución del programa, llamada OB-LIST, si ya existe en la tabla se direcciona a esa posición, si no existe se crea una nueva celda que contendrá el nombre del átomo. Todas las funciones en LISP tienen su representación, como si fueran un átomo cualquiera, en la tabla simbólica. Estas funciones se definen en esta tabla al inicio del programa, con una lista de propiedades especial.

Esta simple organización hace posible trasladar programas y datos a la misma representación interna de listas encadenadas. El siguiente ejemplo muestra una simple definición de función en LISP con su correspondiente representación interna.

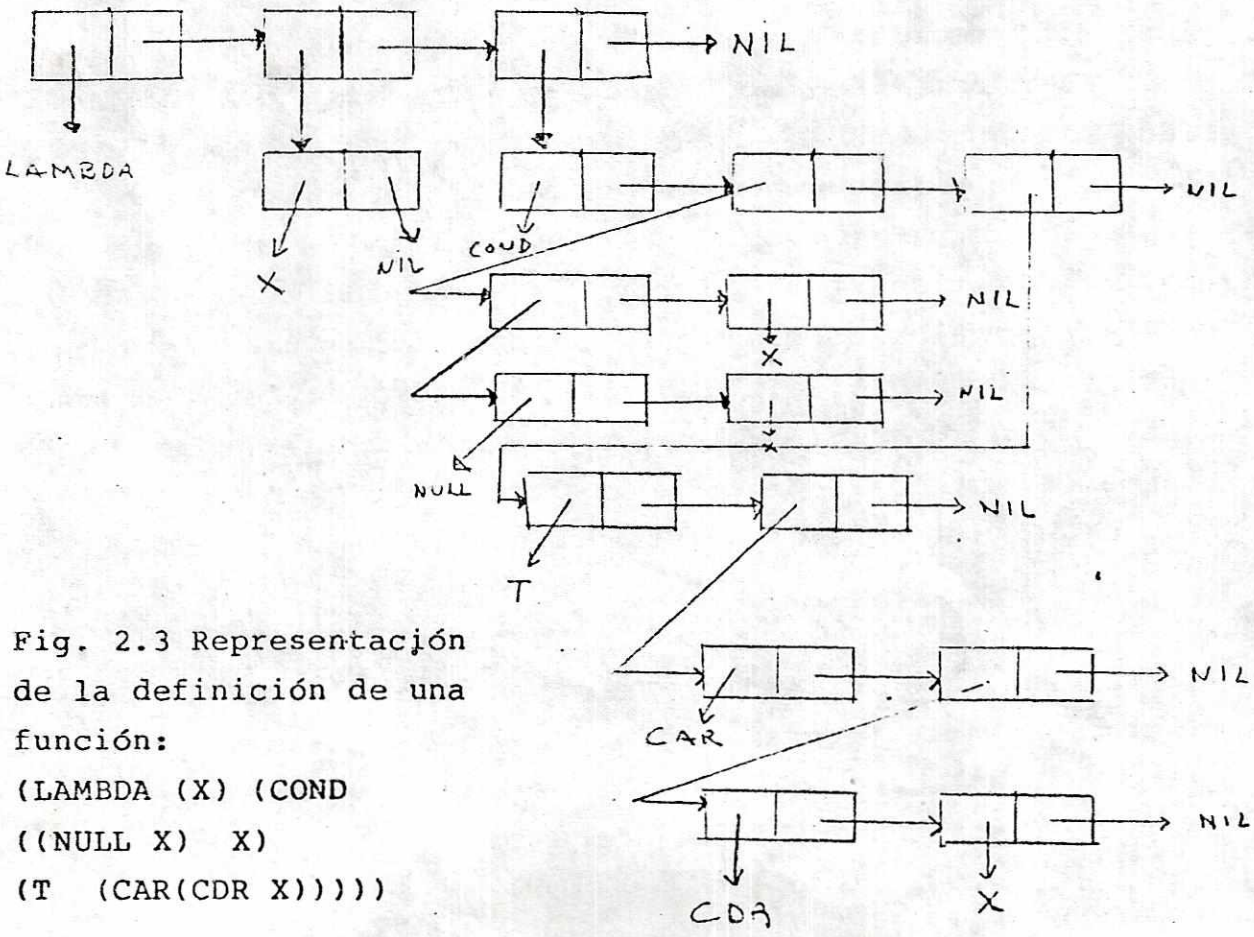



Fig. 2.3 Representación de la definición de una función:
(LAMBDA (X) (COND
((NULL X) X)
(T (CAR(CDR X)))))

Es de hacer notar que esta representación interna común entre programas y estructuras de datos permite que programas en LISP sean creados dinámicamente por otros programas durante la ejecución y más tarde sean corridos. Así, una lista de datos puede convertirse en una lista de programas.

B. Operaciones en LISP.

Las operaciones forman el complemento de los datos en programación; los datos representan el componente pasivo, la información almacenada; las operaciones representan el componente activo, la cual crea, destruye y transforma los datos.

1. Manejo de listas. CAR Y CDR. Las operaciones CAR y CDR obtienen el señalador CAR de un elemento de lista dañado, respectivamente.

Así dada una lista L: (( A B ) C )

Representada así:

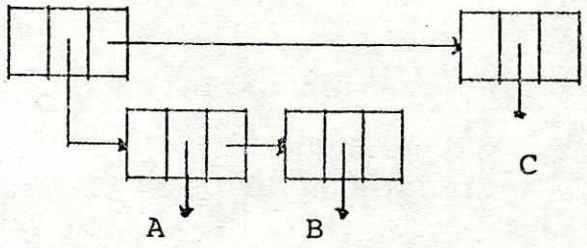



Fig. 2.4 Representación de la lista ((A B) C)

Si aplicamos la función (CAR L), el resultado es la sublista: ( A B ) = (CAR L)

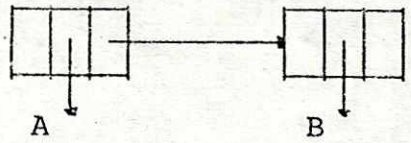



Fig. 2.5 Representación de la lista (A B)

Ya que se toma el señalador CAR del primer elemento de lista L; el cual apunta a la sublista (A B).

Si aplicamos la función (CDR L) a la misma lista L, el resultado sería la sublista (C).

(C) = (CDR L)

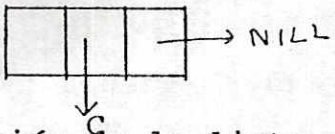



Fig. 2.6 Representación de la lista (C)

Ya que se toma el CDR del primer elemento de lista L; el cual señala a la sublista (C).

CONS. La función CONS toma dos señaladores como operandos y crea una nueva unidad de memoria, incorporándola como un nuevo elemento de la lista. Almacena los dos señaladores en los campos CAR y CDR de la nueva unidad, y regresa un señalador a la dirección de esta unidad. El segundo elemento debe ser una lista agregándosele el primer elemento, el cual encabezará la nueva lista formada. Por ejemplo:

Sean las listas+ L1 = (A B C) y L2 = (X Y)

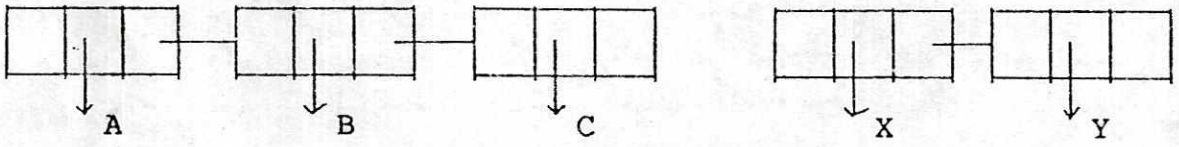



Fig. 2.7 Representación de las listas (A B C) y (X Y)

La operación (CONS L1 L2) construirá la lista:

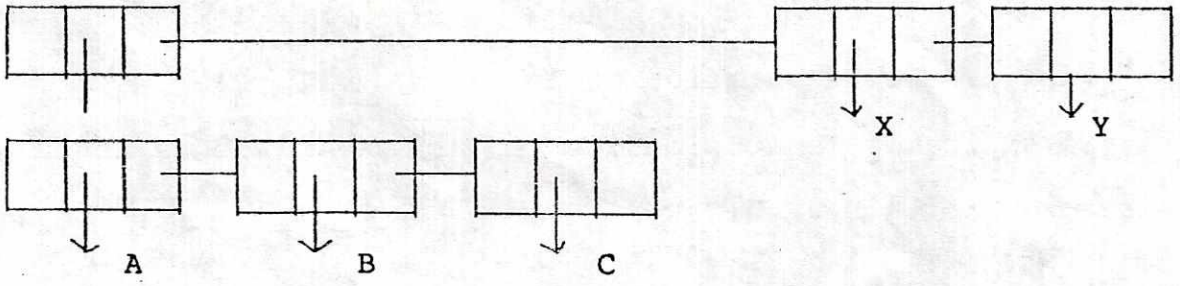



Fig. 2.8 Representación de la lista ((A B C) X Y)

CAR, CDR y CONS son las operaciones básicas para el procesamiento y construcción de listas. Usándolas apropiadamente cualquier lista puede ser dividida en sus elementos constituyentes y nuevas listas pueden ser construídas de estos o de otros elementos.

LIST y QUOTE. LIST puede ser usada para reemplazar una larga secuencia de operaciones CONS. LIST toma cualquier número de argumentos y construye una lista de estos argumentos, regresando un señalador a la lista resultante.

QUOTE permite que cualquier lista o átomo sea escrito como una literal en un programa. Por ejemplo, si L=(B C) es una lista, entonces (CONS (QUOTE A) L) produce la lista (A B C).

Otras funciones de manejo de listas son: APPEND utilizada para concatenar dos listas, COPY copia una lista, EFFACE elimina un elemento de una lista y SUBST substituye un elemento de una lista por otro. La mayoría de implementaciones de LISP extiende este conjunto básico para la inclusión de ctras operaciones que manejan listas.

2. Aritméticas. LISP contiene las operaciones aritméticas básicas PLUS, DIFFERENCE, TIMES y DIVIDE así como algunas otras. Si tenemos la operación aritmética A + B x C se escribe (PLUS A (TIMES B C)).

3. Relacionales, Lógicas y Otros Predicados. Las operaciones lógicas básicas son AND, Or, y NOT. Junto con los predicados aritméticos LESSP (menor-que), GREATERP (mayor que), ZEROP (igual a cero), y MINUSP (negativo).

Dos predicados importantes distingue el tipo de dato de un elemento. ATOM, regresa verdadero (el atomo T) si su argumento es un atomo asi:

(ATOM (QUOTE A)) retorna T, ya que A es un átomo.

Sin embargo si A tiene el valor (B C)

(ATOM A) retorna F, ya que A tiene el valor de una lista.

El otro predicado es NUMBERP retorna verdadero si su argumento es un número (real o entero).

El predicado NULL retorna verdadero si su argumento es el átomo especial NIL (NULL representa la lista vacía, entonces si (NULL L) regresa verdadero L es una lista vacía).

EQ y EQUAL son usados para revisar igualdad. EQ se aplica solamente a los atomos (no incluye números), mientras que EQUAL se aplica arbitrariamente a estructuras de listas (incluyendo átomos y números). MEMBER revisar si un elemento dado se encuentra en una lista.

4. Manejo de Lista de Propiedades. Todo átomo posee un conjunto de atributos que le definen sus características. Por ejemplo, el átomo JUAN podría tener los siguientes atributos: nombre de impresión, que es JUAN, su edad, su sexo, quienes son sus padres, quienes son sus hijos, sus hermanos etc. No necesariamente debe tener todos estos atributos, pero sí es necesario que tenga por lo menos el nombre de impresión. Las operaciones para el manejo de estas Listas de Propiedad son:

ATTRIB y DEFLIST son usadas para agregar las parejas de nombre y valor a la Lista de Propiedad; GET regresa el valor actual asociado con un nombre; REMPROP elimina el nombre y el valor de un atributo que tenga la Lista de Propiedad.

Por ejemplo para agregar la pareja nombre-valor AGE, 26 a la Lista de Propiedades de un átomo JOE, se escribe

(DEFLIST(((QUOTE JOE) 26))(QUOTE AGE))

Más tarde en el programa la propiedad AGE del átomo JOE puede ser obtenida a través de la función

(GET (QUOTE JOE) (QUOTE AGE))

La eliminación de la AGE de JOE puede ser llevada a cabo por

(REMPROP (QUOTE JOE) (QUOTE AGE))

Las operaciones DEFINE, CSET Y CSETQ también modifican las Listas de Propiedades por nombres de propiedad especiales.

5. Asignaciones. Las asignaciones directas suelen tener un papel central en la programación de LISP, como en otros lenguajes. Muchos programas en LISP son escritos por entero sin contener ni una sóla función de asignación. Lo que más se utiliza es la recursión y la transmisión de parámetros para obtener el mismo efecto indirectamente.

El operador de asignación es SETQ. La expresión (SETQ X VAL) asigna el valor de VAL como el nuevo valor de la variable X.

La operación SET es idéntica a la SETQ excepto que la variable a la cual se le hará la asignación puede ser evaluada. Por ejemplo, (SET(CAR L)VAL) es equivalente a (SETQ X VAL) si el átomo X es el primer elemento de la lista L. SETQ y SET son utilizados para cambiar los valores de las variables en el ambiente actual de referencia.

A las variables globales se le asignan valores usando las operaciones CSETQ y CSET.

Otras dos operaciones, RPLACA y RPLACD, permiten asignaciones al CAR y CDR respectivamente, de cualquier elemento de una lista.

6. Entrada y Salida. La salida de la mayoría de los programas LISP es automática.

Para cada llamada de una función en el programa, el sistema automáticamente imprime el valor calculado por la función.

Datos de entrada pueden ser incluídos directamente en la forma de parametros actuales, en las llamadas de las funciones. Así entonces, muchos programas en LISP se ejecutan enteramente sin ningún comando explícito de entrada y salida.

Operaciones simples son provistas para leer y escribir archivos externos secuenciales. READ, lee la lista de Expresiones-S que se encuentren en un archivo de entrada; PRINT imprime una lista; y PRINTPROP imprime una Lista de Propiedades.

Otro conjunto de funciones permiten que un archivo de entrada lea caracter por caracter, y una línea de salida se construya elemento por elemento bajo el control del programador.

7. Subprogramas. Los subprogramas en LISP son siempre definidos como funciones. Una definición de función tiene la forma de:

fn-name(LAMBDA(lista-de-parámetros)(cuerpo-de-la-fn)))

El "cuerpo-de-la-fn", es una expresión arbitraria.

Como parámetros actuales, se le pueden dar a las funciones estructuras de datos especiales o arbitraria, y así cualquier función puede regresar como valor otra estructura de dato. Así, las funciones LISP no están restringidas en su habilidad para crear, destruir o modificar estructuras de datos. Desde que estas estructuras de datos pueden ser definiciones de subprogramas, es posible escribir programas LISP, los cuales crean o modifican otros programas LISP.

C. Control de secuencias en LISP.

Un programa en LISP está compuesto de una simple secuencia de llamadas de funciones, cada una consistiendo de un nombre de función seguido por una lista de parámetros actuales. Estas llamadas de funciones son ejecutadas en secuencia, hasta que se alcanza el paréntesis derecho que coincida con el primer paréntesis izquierdo, definiendo así, una Expresión-S completa.

1. Expresiones. Las expresiones en LISP están escritas en notación estricta, llamada Cambridge Polish (la cual consiste en un nombre de función seguida de los parámetros actuales) con una completa parentización. Esto hace una sintáxis muy simple y regular.

La función COND permite la ramificación dentro de las expresiones.

Como todas las operaciones en LISP, COND es escrita como una función seguida de sus parámetros actuales, COND invoca una especial regla de evaluación la cual da el efecto de una estructura de control ramificada. La forma es:

```
(COND  ( test_1     result_expr_1)
       ( test_2     result_expr_2)
       ( test_3     result_expr_3)
          "               "
          "               "
          "               "
       ( test_k     result_expr_k)   )
```

Cada test es una expresión que debe ser evaluada a verdadero o falso. Las expresiones test son evaluadas en secuencia y cuando una evaluación resulta verdadera la correspondiente result_expression es evaluada y el valor regresado es el valor con el que regresa la función COND.

2. Instrucciones especiales y PROG. Cualquier programa LISP puede ser escrito usando solamente expresiones simples, condicionales y llamadas a funciones recursivas. Sin embargo, para muchos algoritmos los cuales podrían ser codificados usando instrucciones iterativas ya que la codificación no iterativa requiere un conocimiento profundo de la recursión, es necesario tener la facilidad de poder utilizar este tipo de instrucciones para dar el efecto interactivo. La instruccion PROG permite que un ciclo sea codificado directamente. PROG toma la forma de:

```
(PROG  (lista de variables locales)
       (expr_1)
       (expr_2)
          "
          "
          "
       (expr_n)   )
```

Un átomo puede ser colocado entre cualquier par de expresiones en un PROG, para servir como etiqueta a las siguientes expresiones. Un "goto" es provisto en la forma (GO label), la cual transfiere el control a la expresión siguiente de la etiqueta designada.

Un PROG puede tener salida, ya sea completando la evaluación de la última expresión (en este caso PROG tendrá el valor de NIL) o por una llamada a la operación RETURN. El argumento de RETURN es el valor a ser regresado como valor de PROG.

3. Generadores de elementos de listas. Un simple conjunto de generadores son llamados funcionales en la terminología de LISP. El funcional MAPCAR es típico. La función llamada, (MAPCAR"list" "fn-name") aplica la función fn-name para cada elemento de la list "list" en secuencia. El valor de MAPCAR es la lista compuesta de los valores producidos por la "fn-name" duante este proceso. MAPCAR es usada para reemplazar el uso de ciclos explícitos o la recursión en el proceso secuencial de una lista. Por ejemplo:

(MAPCAR FRUTAS FUNMANZANA)

En donde FRUTAS es un átomo que tiene una lista de frutas, por ejemplo FRUTAS= (NARANJA MANZANA MANZANA MANZANA PERA UVA)

Y FUNMANZANA es una función que reecibe un átomo como parámetro y regresa T (verdadero) si el atomo es MANZANA y F si no lo es.

Entonces por medio de MAPCAR se evalúa una lista de átomos, en este caso FRUTAS, dando el valor para cada átomo de la lista, asi:

(MAPCAR FRUTAS FUNMANZANA)

regresa (F T T T F F)

Ya que sólo los elementos 2, 3 y 4 de la lista FRUTAS tienen el valor MANZANA.

D. Control de Datos:

En la mayoría de las implementaciones de lenguajes el ambiente de referencia es mantenido como una estructura de datos definida por el sistema, oculta al programador. LISP es el único en hacer que el ambiente de referecia se una estructura de datos explícita, la cual es visible y accesible al programador. La exacta representación del ambiente más simple es la que usa una explícita lista de asociación, la lista-A.

1. Listas-A y ambientes de referencia. Una lista-A (lista de Asociación) representa una ambiente de referencia. La lista-A es una lista ordinaria de LISP, cada uno de cuyos elementos es un señalador a una unidad de memoria representando un identificador (átomo) y su asociación actual.

Estas asociaciones de parejas son unidades de memoria cuyo CAR, apunta al átomo y cuyo CDR apunta al valor (ya sea otra lista, u otro átomo) asociado con el átomo. La figura siguiente ilustra la estructura de la Lista-A.

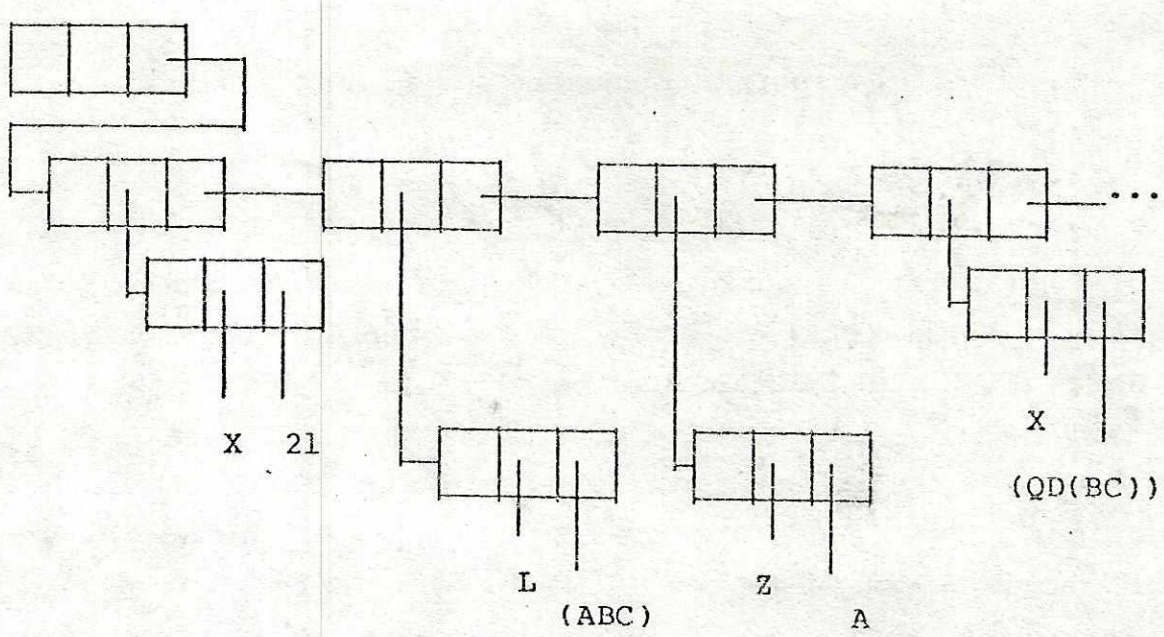



Fig. 2.9 Ejemplo de la estructura Lista-A

Las referencias en LISP están estrictamente de acuerdo a la regla de la más reciente asociación la cual es implementada por una simple búsqueda en la lista-A desde el inicio (las asociaciones más recientes) al final (las asociaciones menos recientes), hasta que una asociación apropiada es encontrada.

La lista-A es modificada durante la ejecución de un programa en tres formas básica:

Primero, cuando una llamada de una función ocurre, el interpretador de LISP (actualmente la función LAMBDA) aparea los átomos, representando los parámetros formales con sus correspondientes valores de los parámetros actuales y agrega las parejas resultantes a la lista A.

Segundo, lo mismo sucede cuando se usa la función PROG.

Tercero, el programador puede modificar la más reciente asociación en la Lista-A usando SETQ o SET.

2. Referencias Globales. LISP es único en proveer una facilidad para que cualquier átomo le sea dada una asociación global, la cual es usada en preferencia si una asociación en la lista-A esta presente.

Un átomo tiene una asociación global Nombre de Propiedad APVAL que aparece en su Lista de Propiedad. El valor asociado con APVAL es el valor global del átomo. Las operaciones CSETA y CSET pueden ser usadas para dar a un átomo una asociación global. Por ejemplo (CSET Q PI 3.1416) pone el valor global del átomo PI a 3.1416 insertando el Nombre de Propiedad APVAL y el Valor de Propiedad 3.1416 en la Lista de Propiedad de PI. Cada vez que PI es referenciado en cualquier parte en el programa de ejecución, el valor 3.1416, puede ser obtenido (sin tomar en consideración si PI se encuentra en la Lista-A. La regla de referencia completa para átomos usados como variables es la siguiente: Busca primero un APVAL en la Lista de Propiedad del átomo y luego busca en la Lista-A.

Una técnica diferente es usada cuando un átomo se utiliza como nombre de una función. En vez de buscar un APVAL en la Lista de Propiedad de átomo un especial nombre de Propiedad es buscado como: EXPR, SUBR, FEXPR, o FSUBR. Estos difieren entre sí como sigue:

EXPR: la definición de la función está en forma de estructura de lista LISP, y los parámetros son transmitidos por valor.

SUBR: Igual a EXPR, excepto que la función está en len-

guaje de máquina o una función compilada definida por el programador.

FEXPR: Igual a EXPR, sólo que los parámetros son transmitidos por nombre (no evaluados).

FSUBR: Igual a SUBR sólo que los parámetros son transmitidos sin evaluar.

Si el nombre de una función no tiene Nombre de Propiedad en su Lista de Propiedad, entonces la búsqueda se hace en la Lista-A en la forma usual, y la función encontrada es tratada como EXPR.

## III. MAQUINA EN LA QUE SE DESARROLLA EL INTERPRETE

Un computador puede definirse como: Un conjunto integrado de algoritmos y estructuras de datos capaces de almacenar y ejecutar programas. Un computador puede ser construído como un dispositivo físico usando alambres, transistores, memorias magnéticas, etc. En este caso se le llama Computador "Hardware", pero igualmente puede ser construído usando "Software", ejecutando programas en algún otro tipo de computador, en este caso se le llama Computador Simulado por "Software".

Un lenguaje de programación es implementado construyendo un Traductor, el cual traduce programas de un lenguaje específico a programas de Lenguaje de Máquina capaces de ser ejecutados directamente por el computador. El computador que ejecuta el lenguaje traducido puede ocacionalmente ser un computador "Hardware", pero ordinariamente es un Computador Virtual compuesto parcialmente de "Hardware" y parcialmente de "Software".

### A. Computador Virtual.

Un computador puede ser construído así:

a) A través de la realización de "HARDWARE", representando las estructuras de datos y algoritmos directamente con los dispositivos físicos.

b) A través de una realización "FIRMWARE", representando las estructuras de datos y algoritmos por microprogramación, en un "Hardware" adecuado.

c) A tráves de simulación de "Software", representando las estructuras de datos y algoritmos por programas y estructuras de datos en algun otro lenguaje de programación.

d) A través de alguna combinación de estas técnicas, representando varias partes del Computador directamente en "Hardware" y en Microprogramas por la simulación de "Software" apropiado.

Un computador "Hardware" es conocido como Computador Real. Un computador que está parcialmente o por completo simulado por "Software" o microprogramas es llamado Computador Virtual.

B. El Computador Virtual LISP.

La organización de la memoria durante la ejecución del programa en una implementación típica de LISP, es por medio del tipo de memoria conocido como "HEAP", la cual contiene las estructuras de lista de datos y las estructuras de lista de funciones.

Un "HEAP" es un bloque de almacenamiento dentro del cual las piezas son colocadas y liberadas en una forma relativamente, no estructurada. Es por esto que aquí, los problemas de colocación de almacenamiento, recuperación, compactación y reuso deben ser muy severos. No existe una técnica definida en el manejo de este almacenamiento. (En este trabajo se útilizó una que será explicada más adelante).

La necesidad del almacenamiento "HEAP" y su manejo aumenta cuando el lenguaje requiere que el almacenamiento sea colocado y liberado en puntos arbitrarios durante la ejecu-

ción del programa, y cuando el lenguaje permite la creación, destrucción o extensión de las estructuras de datos del programador es puntos distintos del programa. En LISP, un nuevo elemento puede ser agregado a una existente estructura de lista en cualquier punto, requeriendo que un nuevo almacenamiento sea creado.

Es conveniente dividir el manejo del almacenamiento en un "HEAP", en dos categorías dependiendo de si los elementos colocados son simpre del mismo tamaño o de tamaño variable.

En caso de que se use un "HEAP" de tamaño fijo, este consiste en una secuencia de K elementos, cada una de N palabras de largo, donde K x N = el tamaño del "HEAP BLOCK".

La figura siguiente muestra a grandes rasgos una organización típica de la memoria LISP en tiempos de ejecución.

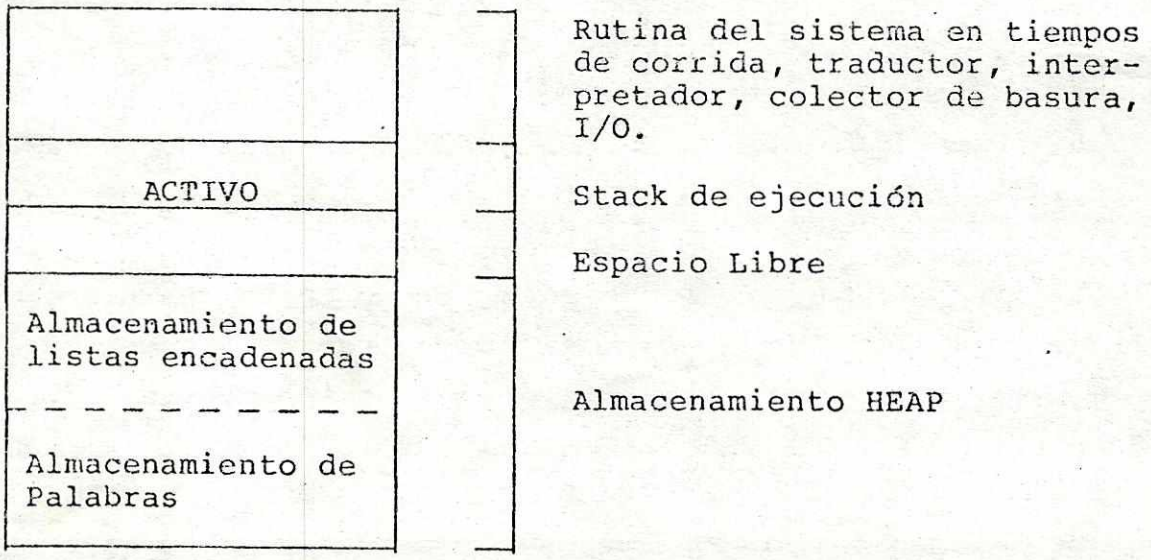



Fig. 3.1 Organización de la memoria LISP (EL HEAP)

C. Almacenamiento inicial y reuso de celdas.

El almacenamiento del "Heap" puede ser llevada a cabo con un simple señalador "Heap", el cual señala al siguiente elemento libre disponible en el "Heap". Cada vez que un nuevo elemento se necesita, el señalador "Heap" se direcciona al punto siguiente de un elemento libre. Sin embargo, hay un problema cuando se libera un elemento que ya ha sido usado y quede a disposición de ser usado otra vez. Ya que el señalador del "Heap" busca elementos libres hacia adelante y le es difícil reconocer posiciones libres que le quedan atras.

Entonces estamos restringidos a "avanzar" el señalador durante la colocación, con algún mecanismo para mantener la pista del almacenamiento que haya sido liberado.

Eventualmente el señalador "Heap" alcanza el fin del bloque asignado al "Heap". En este punto necsitamos hacer disponibles aquellos elementos de atrás que hayan sido liberados. ¿Como mantener la pista de los elementos del "Heap" que han sido liberados? La técnica más común es mantener una lista encadenada de estos elementos libres, llamada Lista con Espacio Libre. Una celda de almacenamiento fuera del "Heap", es acogida como celda que contendrá un puntero a otro elemento libre, el cual se encadena con algún otro, así sucesivamente.

La colocación de elementos de esta lista de Espacio Libre es simple. Cuando un elemento es necesario, se accesa la celda que contiene el puntero al primer elemento libre disponible. Ya que es utilizado hay que eliminarlos de la Lista de Espacio Libre, esto se hace tomando el puntero al segundo elemento libre (que se encuentra en el primer elemento) y se almacena en la celda que apunta siempre al primer elemento libre, haciendo que este segundo sea ahora el primer elemento libre.

Cuando un elemento en uso es liberado, el proceso se invierte, agregándolo a la cabeza del Espacio Libre disponible. Haciendo que su contenido señale al que antes era el primer elemento libre disponible y haciendo que la celda que siempre señale al primer espacio libre diponible le señale a él. La figura ilustra lo anterior:

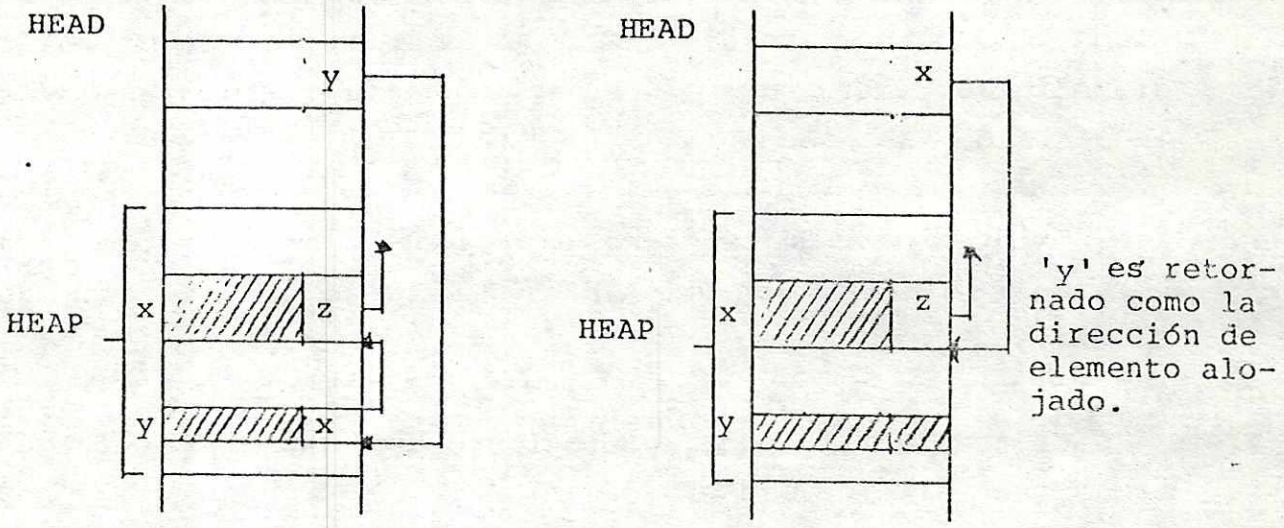



Fig. 3.2 Forma de asignación de celdas activas

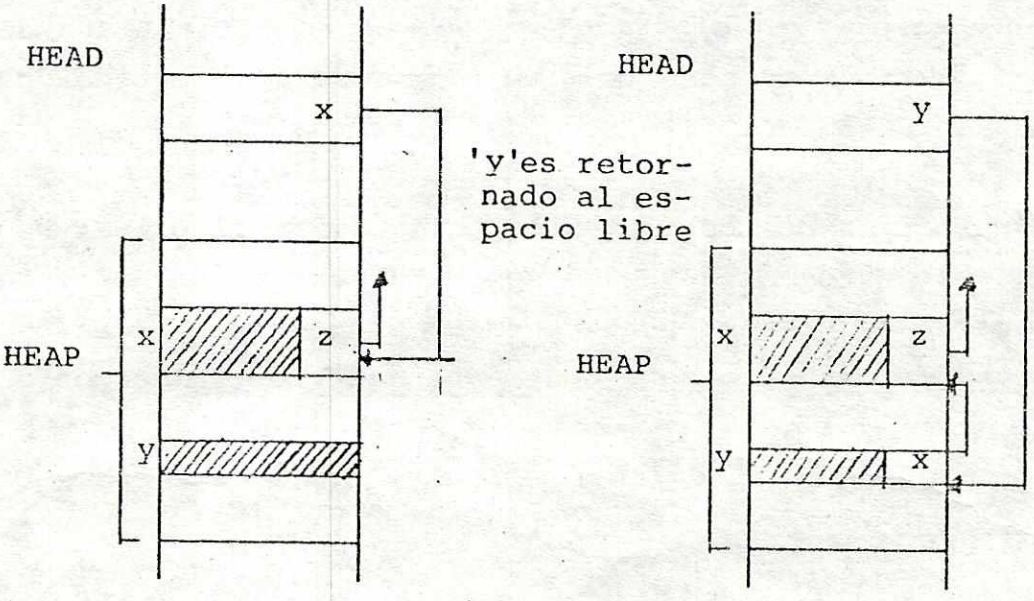



Fig. 3.3 Haciendo disponibles celdas inactivas

D. Recuperación de Celdas Inactivas

1. Colector de basura. El retorno de almacenamiento libre a la Lista de Espacio Libre simple, si se provee que tal almacenamiento sea identificado y recuperado. Pero la identificación y recuperación puede ser muy difícil. El problema está en determinar qué elementos en el "Heap" están disponibles para re-uso y entonces puedan ser regresados a la lista de Espacio Libre.

Existen varias técnicas para lograr lo anterior, sin embargo, la más común en LISP es la llamada: Colector de Basura.

Cuando la lista de espacio libre está por completo agotada y necesita de más almacenamiento, el proceso es suspendido temporalmente y un procedimiento extraordinario es iniciado: "Una colección de basura", el cual identifica los elementos "basura" en el "Heap" y los regresa a la Lista de Espacio Libre. El proceso original es entonces continuado. La colección de Basura involucra dos estados:

a) Marcación de los elementos activos. En el primer estado cada elemento en el "Heap" que está activo, por ejemplo que sea parte de una estructura de datos accesible, debe ser marcado. Cada elemento debe contener un Bit de Colección de Basura puesto inicialmente encendido. El algoritmo de marcación, pone el Bit de Colección de Basura de cada elemento activo apagado.

b) Colectar los elementos de basura. Una vez el algoritmo anterior ha marcado los elementos activos, el resto cuyo Bit de Colección de Basura está encendido, es basura y puede ser regresado a la lista de Espacio Libre. Una simple barrida secuencial del "Heap" es suficiente en este estado.

El primer estado, el de marcación del "Bit" de Basura es el más difícil. Ya que cuando la lista de espacio libre se agota, la colección de basura se inicia; cada elemento en el "Heap" está, ya sea activo (p. ej. que aún esté en uso) o bien es basura. Desafortunadamente, la inspección de un elemento no puede indicar su estado ya que no hay nada intrínsico en el elemento "basura" que indique que es "basura". Más aún, la presencia de un puntero a un elemento señalado por otro del "Heap" no necesariamente indica, que el elemento señalado esté activo; ya que puede ser que ambos elementos sean "basura". Entonces una simple barrida del "Heap" la cual revisa señaladores y marca los elementos apuntados, como activos, no es suficiente.

¿Cuándo esta activo un elemento "Heap"? Un elemento está activo si hay señalador a él que provenga de afuera del "Heap" o de otro elemento "Heap" activo. Si es posible identificar estos punteros de afuera, y marcar los elementos "heap" apropiados, entonces puede ser iniciado un proceso interactivo de marcación el cual busca otros elementos activos tomando las direcciones a las que apuntan aquellos señaladores. Estos nuevos elementos son entonces marcados y se toma su señalador para identificar otros elementos activos, así sucesivamente.

Hay que tomar en cuenta que se necesitan tres suposiciones para llevar a cabo el proceso de marcación, explicado anteriormente:

a) Cualquier elemento activo debe ser accesible a través de una cadena de señaladores que principie fuera del "Heap".

b) Debe ser posible identificar cada señalador fuera del "Heap" el cual señale a un elemento dentro del "Heap".

c) Debe ser posible identificar dentro de cualquier elemento activo del "Heap" los campos que contienen señaladores a otros elementos dentro del Heap.

Si cualquiera de estas tres asumciones no es satisfecha, entonces el proceso de marcación fallará al marcar algunos elementos como activos.

La manera como éstas asumciones son satisfechas en una implementación típica de LISP es:

Primero, cada elemento "Heap" es formateado idénticamente, usualmente con dos campos de punteros y un conjunto de "Bits" como data del sistema (incluyendo el "Bit" de Colección de Basura). Como cada elemento "Heap" contiene exactamente dos señaladores, y estos dos señaladores están siempre en las mismas posiciones dentro del elemento, la asumción c) es satisfecha.

Segundo, existe solamente un pequeño conjunto de estructuras de datos del sistema, las cuales tienen señaladores al "Heap" (la Lista-A, la OB-LISTA etc.,). Marcando el inicio de estas estructuras de datos del sistema se garantiza la identificación de todos los señaladores externos dentro del "Heap", como requiere la asumción b).

Finalmente es imposible alcanzar un elemento "Heap" por otra cadena de señaladores que empiece fuera del "Heap". La asumción a) queda satisfecha.

## IV. DESARROLLO DEL INTERPRETE

El desarrollo del intérprete consiste de dos fases fundamentales que son:

a) La creación de una Máquina Virtual especialmente diseñada para la evaluación de expresiones simbólicas, las cuales forman el lenguaje LISP.

b) La creación de los algoritmos que manejan las expresiones simbólicas en la Máquina Virtual creada en la primera fase.

### A. Estructuras de Datos que forman la Máquina Virtual.

Las distintas partes que constituyen una Máquina Virtual se conocen como Estructuras de Datos. Una estructura de datos es simplemente un conjunto estructurado de elementos de datos. Entonces la simulación de esta Máquina Virtual sobre el computador define la organización lógica de los datos y el lenguaje de programación provee la representación sintáctica necesaria para estos datos. Este conjunto estructurado tiene las siguientes características:

-.La forma en que puede ser accesada.
-.El tipo de datos que puede almacenar.
-.El formato de sus elementos.
-.Las formas de comunicación entre estos elementos.

Las principales estructuras de datos diseñadas para la creación de la Máquina Virtual LISP son:

-.Una Tabla Simbólica (Lista-OB)
-.Un método para asociar los símbolos con sus valores (Lista-A)

-. Un "Stack" de ejecución.

-. Un área de almacenamiento principal (Heap).

1. La Tabla Simbólica (Lista-Ob). Los átomos en LISP son simplemente símbolos abstractos que necesitan ser representados por un conjunto de caracteres conocido como el PNAME (De Print NAME en inglés). Se podría decir que el PNAME es el nombre del átomo utilizado en las operaciones de entrada y salida.

Este conjunto de caracteres representando los símbolos son almacenados en una tabla Simbólica. De esta forma todo átomo tiene un señalador a esta tablapara las operaciones de entrada y salida.

La forma como se implementó en este Intérprete esta tabla simbólica fue usando la técnica "Desmenuzamiento" en donde las colisiones son manejadas por "rebalse", permitiendo un rápido acceso a cualquier PNAME en particular, así como la eliminación o adición de alguno de ellos. La técnica de "Desmenuzamiento" es la forma tradicional en LISP de representar átomos.

Para ello se diseñaron dos tipos de formatos:

El primero, el A, tiene la forma siguiente:

| ONOB | ATOMPT | NEXTPT | STRGPT |
|---|---|---|---|
| On/Off | Ptr | Ptr | Ptr |

Fig.4.1 Formato "A" de la tabla simbólica.

El campo ONOB indica si la celda esta activa o no, para uso del Colector de Basura.

El ATOMPT es un señalador al átomo que se encuentra en la memoria principal del Interprete, el "Heap".

NEXTPT, en caso de que hubieran colisiones, este elemento señala a una celda de desbordé.

STRGPT, elemento que señala al segundo tipo de formato, que es donde se encuentra el nombre del átomo.

El segundo formato, el B, tiene la forma siguiente:

| CHARPT | CHARS |
|---|---|
| Ptr | Seis Caracteres pòsibles |

Fig. 4.2 Formato "B" de la tabla simbólica.

CHARS, Tiene una longitud de tres palabras para poder así contener el nombre del átomo. Ahora bien si este nombre tiene más de 6 caracteres se utiliza el campo CHARPT, que es un señalador a una celda con el mismo formato con la disposición de poder almacenar otros 6 caracteres, así sucesivamente se van creando celdas hasta que se logre poder almacenar el nombre. La última de estas celdas tiene el valor -1 en el fampo CHARPT lo que indica que el nombre ya no necesita de más celdas.

Un vistazo general a esta tabla simbólica es representado en la gráfica de la página siguiente. Una variable es leída y pasa por una función de "Desmenuzamiento" que convierte esa variable en un valor numérico entre 1 y 49, dependiendo de los caracteres de que consta esta variable. Luego se toma la dirección de la siguiente celda disponible, almacenando este valor en la casilla entre 1 y 49 segun sea el valor de "Desmenuzamiento" que haya recibido la variable. Después se crea otra celda que contendrá el nombre de esta variable.

En caso de que una variable tome algun valor entre 1 y 49 ya asignado se seguirá la cadena de desborde direccionada por el campo NEXTPT del formato de la celda A, para colocarse al final de esta cadena.

Por último, se hace señalar al campo ATOMPT a la posición del átomo en el "Heap".

2. La Lista de Asociación o Lista-A. La Lista-A es una estructura representada como un "push-down", cuyos elementos son parejas de átomos y valores. Esta estructura permite 'La regla de asociación más reciente' a ser aplicada, recorriendo el "stack" secuencialmente desde el tope; usando la primera asociación encontrada. La Lista-A es una Expresión-S.

3. El "Stack" de ejecución. Esta estructura contiene las direcciones de las celdas que se recorren al ir evaluando una Expresión-S. Es utilizado el "stack" ya que es muy útil para guardar las direcciones de retorno en los momentos en que se evalúan sub-expresiones a niveles más bajos, y así poder regresar a los niveles de arriba por medio de estas direcciones.

4. Memoria principal del Interprete. El "heap" es el área de almacenamiento principal de LISP.

El "Heap" es un bloque de almacenamiento constituído por elementos llamados celdas los cuales son colocados y liberados de una manera relativamente no estructurada.

Esta estructura de datos es necesaria en LISP ya que se utiliza en situaciones donde el almacenamiento es muy dinámico, es decir, donde se agregan y se liberan celdas en puntos arbitrarios de la ejecución de un programa.

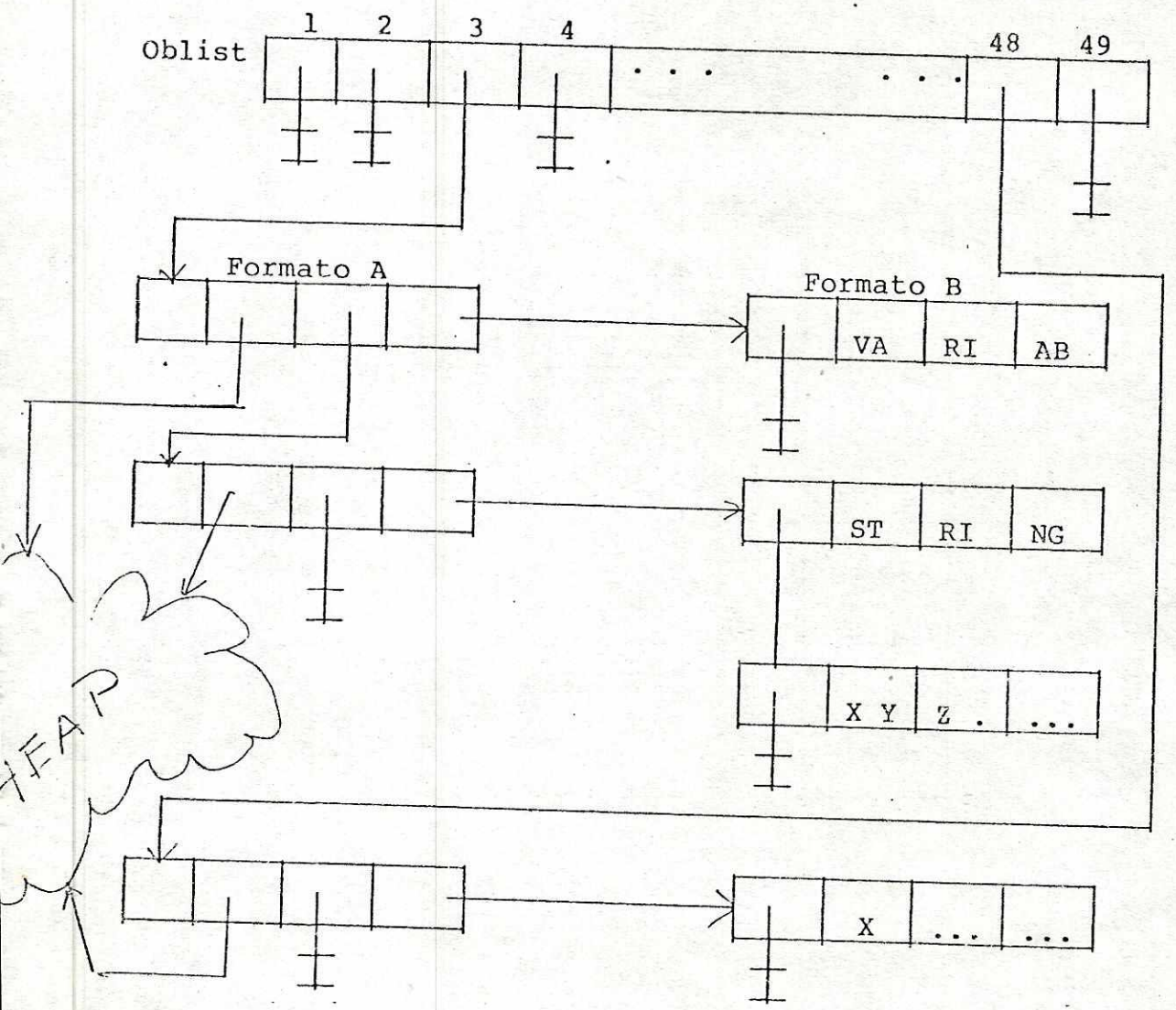



Fig 4.3 Ejemplo de Encadenamiento de la tabla Simbolica

Dentro de esta área de almacenamiento fueron creados cuatro tipos de formatos, que son:

| TYPE | CAR | CDR |
|---|---|---|
| CONS | Ptr | Ptr |

| TYPE | PNAME | VAL | PLIST |
|---|---|---|---|
| SATOM | Ptr | Ptr | Ptr |

| TYPE | IVAL |
|---|---|
| IATOM | # Entero |

| TYPE | ADDRS |
|---|---|
| SUBR | Ptr |

Fig.4.4 Formato de tipos de celdas del "Heap"

En las celdas de tipo CONS se utilizan en la representación de listas en las Expresiones-S. Posee dos señaladores; el CAR y el CDR. El señalador CAR señala al primer elemento de una lista, y el CDR señala a lo que queda de la lista después de haber quitado el primer elemento.

La celda tipo SATOM contiene información de los átomos, así: el campo PNAME señala a la celda que contiene el nombre del átomo o "Print Name". El campo VAL contiene un elemento que señala al valor del átomo ya sea una lista u otro átomo. Y el campo PLIST que señala al área donde se encuentra la Lista de Propiedades del átomo.

La celda tipo IATOM contiene a los átomos numéricos enteros. En donde el campo IVAL es el valor que tiene el átomo.

Por último, la celda tipo SUBR es utilizada para definir subrutinas por medio de la instrucción LAMBDA. Entonces, cuando se define una subrutina LAMBDA, se crea una celda tipo SUBR almacenando la dirección del comienzo de esta subrutina en el campo ADDRS.

De esta forma el "Heap" contiene todas las celdas necesarias, para construír las Expresiones-S. Estas Expresiones-S son construídas con una estructura de Arbol Binario en donde se usan las celdas tipo CONS como nodos, no terminales, y las celdaS SATOM y IATOM (que corresponden a átomos) como nodos terminales.

B. Algoritmos principales

Ya que las estructuras de datos de la Máquina Virtual LISP han sido creadas, es necesario construir los algoritmos adecuados a estas estructuras para poder así evaluar y ejecutar el código fuente escrito en LISP.

Los principales algoritmos que han sido creados son:

-. Analizador de Léxico
-. Representación de las Expresiones-S como Listas Encadenadas
-. Evaluación de las Expresiones-S.
-. Ejecución de las funciones evaluadas en EVAL.
-. Generador de la salida del programa.

1. Analizador de léxico. El problema del reconocimiento del código fuente puede ser muy difícil dependiendo de la complejidad del lenguaje. Afortunadamente LISP es un lenguaje sencilllo de reconocer debido a su representación de listas en forma de paréntesis. Así, en LISP se trata del reconocimiento de expresiones simbólicas. Practicamente, entonces, la única dificultad se centra en el análisis de léxico de los átomos. Esto implica simplemente la revisión correcta de la sintáxis del átomo, así como su identificación.

El funcionamiento del algoritmo es simple: Recibe el primer caracter de un "token" por medio de la rutina NEXTC, que regresa el caracter siguiente a evaluar. Luego es llamada la rutina CONVE que convierte este caracter a un código, 1 si es dígito, 2 si es letra, 3 si es paréntesis y 4 si es un caracter no permitido.

Ya que identificó al primer caracter, este algoritmo hace una revisión hasta que encuentra un delimitador en el "token" que puede ser un espacio en blanco o un paréntesis. Dependiendo de la revisión genera la siguiente salida:

-. El símbolo 1 si el átomo es numérico.
-. El símbolo 2 si el átomo es simbólico.
-. El símbolo 3 si es paréntesis izquierdo.
-. El símbolo 4 si es paréntesis derecho.

Entonces la salida de este algoritmo es el "token" identificado y su símbolo correspondiente.

2. Representación de las Expresiones-S como listas encadenadas.

La rutina que crea las estructuras de listas encadenadas se llama CNSL, la forma en que opera es la siguiente:

Recibe la salida de la rutina anterior que es el símbolo y el "token". Si el símbolo es 3 (paréntesis izquierdo) crea una celda en el "Heap" tipo CONS (vista en un inciso anterior) ya que la aparición de un paréntesis abierto implica que una lista será representada. Esta celda es liberada por medio de la rutina CFREE que regresa la dirección de la siguiente celda en el "Heap" disponible.

Si el símbolo es 4 indica que se cierra un paréntesis, lo que nos indica que una lista termina, por lo que el CDR de la última celda tipo CONS debe señalar a NILL.

Si el símbolo es 1 o 2 se crea una celda tipo IVAL o SATOM respectivamente. Al mismo tiempo se debe establecer el encadenamiento entre las celdas tipo CONS (no-terminales en el árbol) con estas celdas terminales en el árbol.

En el caso de que el átomo sea simbólico es necesario crearle una entrada en la tabla simbólica y Lista-Ob. Esto se hace por medio de la rutina de desmenuzamiento la cual es un algoritmo que, dependiendo de la secuencia de caracteres que contiene el átomo, le asigna una casilla en la Lista Ob. Esta casilla contendrá la dirección de una celda con el formato A de la tabla simbólica (visto en un inciso anterior) y ésta a su vez contendrá un señalador a la celda con formato B que contiene el PNAME del átomo.

La CRHIP también llama a la GETPN que se encarga de localizar en que posición deben ser creadas las celdas en el área Lista-Ob. Esto es, en el caso que hayan colisiones GETPN dará la posición recorriendo todos los elementos que hayan colisionado hasta llegar al último en la cadena que es donde debe colocarse en nuevo elemento.

Al mismo tiempo revisa si ese átomo ya se encuentra en la tabla simbólica utilizando la rutina COMPA la cual compara dos secuencias de caracteres. Entonces GETPN regresa dos valores, por medio de una variable lógica, indicando si el átomo ya existía en la tabla o nó y la dirección en donde se debe colocar las celdas nuevas en caso de que el átomo no estuviera en la tabla.

Luego CRHIP llama a SETPN para que cree los encadenamientos necesarios para poner las celdas correspondientes al PNAME, en caso que el átomo no exista.

Otra estructura importante en el algoritmo CONSL, usada en la generación de las listas encadenadas representando las Expresiones-S, es el uso de un "Stack" el cual se hace necesario para ir almacenando direcciones para poder bajar de nivel en el árbol y así poder subir otra vez a los niveles originales.

Hasta el momento la lógica es la siguiente: Se llama a la rutina de análisis de sintáxis e identificación de "tokens" que pueden ser números símbolos o paréntesis. Si la sintáxis está correcta, dependiendo de cada "token" se van construyendo las listas encadenadas por medio de la rutina CONSL. Así sucesivamente se identifica otro "token", se representa físicamente en memoria hasta que los paréntesis se completen, es decir que se llegue al mismo número de paréntesis derechos e izquierdos.

3. Evaluación de las Expresiones-S. En vista que las Expresiones-S han sido representadas físicamente en la memoria, con todos sus encadenamientos el paso siguiente es evaluar las distintas subexpresiones.

La rutina EVAL recibe como parámetro la dirección de la celda, que puede ser el inicio de la Expresión-S principal (o programa) o el inicio de una subexpresión dentro del programa; recorriendo por medio de un "stack" (igual a como se construyeron las listas) las distintas celdas que corresponden a esa Expresión Simbólica.

Estos "Stacks" fueron implementados en vista de que el lenguaje utilizado (así como todos los que se encontraban en la Universidad) no eran recursivos. De serlo se hubiera facilitado grandemente el algoritmo. La lógica a grandes rasgos de este algoritmo es el siguiente:

Es S un Atomo? SI SI ENTONCES regresar el valor S .

SINO es QUOTE el ler. elemento de S? SI SI ENTONCES regresa el 2o elemento de S.

SINO es LAMBDA el ler. elemento de S. SI SI ENTONCES identificar como subrutina la expresión anterior.

SINO es función el ler. ele. de S? SI SI ENTONCES meter en "stack" de funciones y CALL EVAL, regresando el 2o. ele.

FIN

Para una mejor comprensión de la forma como se implementó la rutina (que podríamos decir es la clave del intérprete) mostraré unos ejemplos de Expresiones-S:

(SET A (QUOTE (X Y Z)))

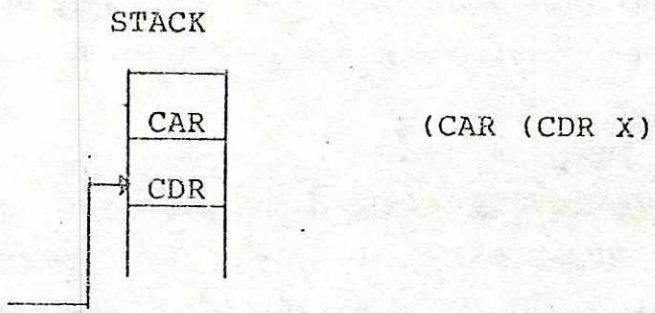
EVAL recibe la dirección donde empieza la expresión anterior, lee el átomo SETQ y revisa en la tabla de funciones reconociendo que es una función definida y la almacena en el "stack" de funciones llamado STACKF. Luego identifica al átomo A regresando un señalador a éste átomo, después sigue QUOTE en donde se regresa la dirección que señala a la expresión siguiente que es el último elemento, terminando así la evaluación. Luego se llama a la rutina EXECU que hará un recorrido del "stack" de funciones ejecutando las funciones que encuentre. Almacenando un señalador a la expresión resultante.

( CAR (CDR A))

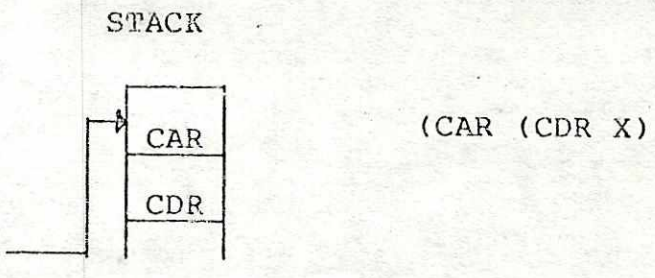
EVAL, identifica a CAR como función definida y la almacena en el "stack" de funciones, luego identifica a CDR y hace lo mismo. Llega con A y regresa la dirección que señala a la lista (X Y Z), que es el valor de A. Luego se alcanza el último paréntesis y se llama a la rutina (EXECU la cual accesa el "stack" de funciones y las ejecuta sobre la lista (X Y Z). Regresando un señalador a la expresión resultante, en este caso Y.

En resumen, EVAL evalúa las Expresiones-S poniendo las funciones en el "stack" de funciones, que serán aplicadas sobre la expresión a la que señala la variable de trabajo, para que cuando se completen los paréntesis llamar a EXECU para ejecutarlas.

El control de evaluación de las distintas subexpresiones se lleva a cabo por medio de un "stack" que opera más o menos de la siguiente forma: El "stack" almacena en cada una de sus posiciones el comienzo de una sub-Expresión-S y es eliminada cuando se cierra un paréntesis, así si miramos en algun momento la evaluación de una Expresión-S, podría estar asi:




Si continuamos y cerramos un paréntesis el señalador al tope del "stack" disminuye en 1 quedando así:




Hasta que el stack se vacía por la paridad en los paréntesis.

En la recursividad se controla la dirección de retorno por medio de una etiqueta común y una variable lógica.

La etiqueta se encuntra en la línea siguiente donde se simuló el CALL EVAL (esta simulación se hace por medio de un "go to" al inicio de EVAL).

La variable lógica indicará cual es la etiqueta que le corresponde a una llamada recursiva.

Ahora bien hay que controlar en qué momento se terminó de evaluar la expresión para poder regresar a la línea siguiente. Esto se hace almacenando el señalador -del-Tope-del-stack en una variable X y cuando el nivel de evaluación haga que el señalador-del-tope-del-stack sea igual a X (ya que al ir evaluando la nueva subexpresión el señalador-del-Tope-del-stack subirá en relación a X un número de posiciones dependiendo de las subexpresiones que tenga) indica que la expresión ha sido evaluada.

Así aparte de evaluar cual es la etiqueta de la dirección de retorno por medio de las variables lógicas, sabemos cual es la expresión 'local' a la simulación de la llamada.

4. Ejecución de las funciones evaluadas. Esta rutina accesa dos estructuras: el "stack" de funciones llamado STACKF y la posición que contiene el señalador a la expresión simbólica sobre la cual trabajarán las funciones en STACKF. Entonces toma el señalador que señala a la expresión simbólica y le aplica la función que en esos momentos está al tope del "stack" de funciones, el resultado de esta operación es una nueva expresión simbólica que es a la que señala ahora el señalador mencionado anteriormente, se toma la función que sigue en el "stack" de funciones y trabaja sobre la nueva expresión dando otro resultado, asi sucesivamente hasta que el "stack" de funciones queda vacío.

El valor al que señala ahora el señalador, es el resultado de la Expresión-S.

Esta rutina contiene la definición de todas las funciones que han sido implementadas en el interprete como CAR, CDR, CONS, SETQ, COND etc. y son las que encargan de todo el manejo de señaladores que cada función implica.

5. Generador de la salida del programa. Esta rutina toma como único parámetro el señalador a la expresión resultante que se obtuvo por medio de la rutina anterior. Su tarea es la de imprimir la expresión simbólica resultante.

Su función es la inversa a la de CONSL o sea de contruir una lista a partir de una expresión simbólica, ya que aquí recibe como parámetro una lista y se tiene que generar la expresión simbólica equivalente.

Se recorre la lista con la ayuda de un "stack" y dependiendo del tipo de celda se coloca en un 'almacenamiento' de salida. Así si se llega a una celda tipo CONS y su CDR no señala a NILL se inserta en el almacenamiento el paréntesis izquierdo "(". Si se llega a una celda tipo SATOM, se utiliza la rutina GPNAM y se obtiene el nombre de impresión del átomo buscando en la tabla simbólica e insertando este nombre en el almacenamiento. Si se llega a una celda tipo IVAL se usa la rutina ENCOD para representar en código ASCII el valor numérico de este átomo. Por último se llega a una celda tipo CONS y su CDR señala a NILL, indica que es el fin de una lista y hay que insertar en el 'almacenamiento' el paréntesis derecho")".

El final se alcanza cuando se vacía el "stack", dicho en otras palabras cuando los paréntesis derechos igualan en numero a los paréntesis izquierdos.

Eso es todo en lo que se refiere a los algoritmos principales del interprete.

Sin embargo, existen muchos módulos que no han sido mencionados y que fueron de gran utilidad en la implementación de los algoritmos anteriores.

A continuación haré una descripción, módulo por módulo, de la funcionalidad de todas las rutinas de las cuales está constituído el interprete de LISP. Encontré apropiado proveer rutinas en el nivel bajo, en la implementación de las estructuras de datos.

La secuencia de ejecución del intérprete está diseñada en un estilo vertical en donde es posible reconocer los siguientes módulos:

-. Inicialización de las estructuras de datos.
-. Lectura de las expresiones simbólicas.
-. Conversión y compactación de los datos.
-. Construcción de listas.
-. Evaluación de las expresiones simbólicas.
-. Ejecución.
-. Impresión.

Cada uno de estos módulos utiliza distintas rutinas que se encuentran en los niveles más bajos del Intérprete. Son los algoritmos principales que se encuentran en el nivel mas alto los que hacen uso de varios subprogramas que se encuentran a un nivel intermedio, los cuales a su vez hacen uso de las rutinas de más bajo nivel en lo que se refiere al acceso directo a las estructuras de datos.

Ahora específicamente se mencionarán brevemente todas las rutinas diseñadas, catalogandolas en el nivel que ocupan en el Intérprete:

-. Rutinas de Alto Nivel.

INIT - Llamada una vez para inicializar el Intérprete

GETOK - Revisar la sintáxis y reconoce los "tokens" catalogándolos dependiendo de su tipo.

CMPCT - Para facilitar el manejo de caracteres de los "tokens", cada caracter al leerse se colocó en na palabra (formato A1), por medio de esta rutina se pasan a formato A2, o sea dos caracteres por palabra.

CONSL - Construye listas encadenadas a partir de las expresiones simbólicas leídas.

EVALS - Controla la evaluación de las expresiones simbólicas.

PRINT - Imprime la expresión simbólica resultante.

-. Rutinas a nivel Intermedio

De estas rutinas hacen uso los algoritmos de Nivel Alto para su implementación modular. Son las que accesan las rutinas de nivel Bajo, para la interacción con las estructuras de datos.

CRHIP - Crea las celdas SATOM e IVAL en el momento en que se leen átomos numéricos o simbólicos.

GETPN - Revisa si ya se encuentra el átomo en la tabla simbólica, si no, determina la posición donde las celdas que contienen el PNAME del átomo deben ser creadas, principalmente en el caso de colisiones.

SETPN - Crea los encadenamientos entre la tabla simbólica y el "Heap", una vez se ha determinado la posición de la celda que contiene el PNAME.

GPNAM - Obtiene el PNAME de un átomo.

SPNAM - Pone el PNAME de un átomo tomando en cuenta la longitud de este nombre.

RWORD - Determina si un átomo se encuentra en la tabla de palabras reservadas, es decir si es una función definida.

NEXTC - Regresa el siguiente caracter a evaluar.

NBLAN - Remueve los blancos.

PUTBF - Pone un "token" en el almacenamiento de salida.

-. Rutinas de Utilidad.

HASH - Tiene la función de desmenuzamiento utilizada para encontrar la posición en la tabla simbólica del nombre de impresión de un átomo.

DECOD - Transforma una cadena de caracteres ASCTI a su representación binaria.

ENCOD - Transforma un número binario a su representación ASCII.

COMPA - Compara dos cadenas de caracteres regresando True si son iguales y False si no lo son.

LENGT - Determina la cantidad de caracteres de una cadena de caracteres.

JUSTI - Justifica una cadena de caracteres a la izquierda.

CONVE - Convierte un caracter a un código en especial, ya sea dígito, letra, paréntesis izquierdo y paréntesis derecho.

-. Rutinas de Nivel Bajo.

Estas rutinas son las únicas que accesan las estructuras de datos directamente, siendo esa su función.

GCAR - Obtiene el segundo elemento de una celda "Heap".

SCAR - Actualiza el segundo elemento de una celda "Heap".

GCDR - Obtiene el Tercer elemento de una celda "Heap".

SCDR - Actualiza el tercer elemento de una celda "Heap".

GTYP - Obtiene el primer elemento de una celda "Heap", conociendo así el tipo de la celda.

STYP - Pone el tipo que le corresponde a una celda.

GPTR - Obtiene el cuarto elemento de una celda "Heap".

SPTR - Actualiza el cuarto elemento de una celda.

PUSHC - Hace una adición en el "stack" de ejecución.

POPC - Hace una eliminación al "stack" de ejecución.

PUSHF - Hace una adición al "stack" de funciones.

POPF - Hace una eliminación al "stack" de funciones.

CFRE - Regresa un señalador a la celda siguiente disponible.

INTERPRETADOR DE LISP

NIVELES DE DISEÑO

INIT -...
LECTURA -...
CONVERSION -...
CONSL -...

INTERMEDIO

SOPORTE - CRHIP, GETPN, SETPN, GPNAM, SPNAM, RWORD, NEXTC, NBLAN, PUTBF.

UTILIDAD - HASH, DECOD, ENCOD, COMPA, LENGT, JUSTI, CONVE.

BAJO

MANHEAP - HCAR, SCAR, GCDR, SCDR, GTYPE, STYPE, GPTRV, CFREE.

MANSTACK - PUSHC, POPC, PUSHF, POPF.

EVALS -....
PRINT -...

Finalmente, el "colector de basura" no ha sido implementado pero han sido creadas las estructuras necesarias para que pueda hacerse sin ningún problema. Cada bloque de memoria puede ser requerido por el "colector de basura" para su uso. La primera posición de cada bloque es usado para almacenar la indicación de sí o no la celda es "basura". Esto se puede determinar revisando el signo del valor en esa posición. Un valor negativo en la primera posición podría indicar que el bloque no es "basura" mientras que un valor positivo podría indicar que sí lo es.

# V RESULTADOS

En esta sección me referiré a las funciones que fueron diseñadas y que funcionan en el Intérprete, así como la forma como se interactúa con él. No mencionaré en esta sección lo que se refiere a las estructuras de datos (como resultado) porque los resultados de éstas se pueden ver en el capítulo anterior de Implementación del Intérprete.

Existen tres caracteres usados como prefijos para señal de respuesta del Intérprete que son:

'*' Significa que el Intérprete está esperando a que el usuario ingrese su expresión simbólica a evaluar.

'>' Significa que lo que sigue es la salida de la expresión simbólica previamente ingresada.

'?' Significa que lo que sigue es la salida de un error en la expresión simbólica previamente ingresada.

## 1. Manipulación de listas

Notación:

Sean: N1, N2.... - Atomos numéricos
S1, S2.... - Expresiones-S
L1, L2.... - Listas
A1, A2.... - Atomos

1. (CAR Sl)  -Regresa el Car de Sl.

```
*(CAR (QUOTE)  (A B C)))
>A
```

```
*(CAR  (QUOTE  (  (A B C)  X Y Z)))
>(A B C)
```

```
*(CAR (QUOTE (((  A ))  ( B )  ( C ) D )))
>(( A ))
```

```
*(CAR (CAR(QUOTE(  (( A )) ( B )  ( C )  D)))
>( A )
```

Sea que el átomo L tiene el valor ( (EN VISTA) (DE LO))

```
*(CAR L)
>(EN VISTA)
```

```
*(CAR(CAR L))
>EN
```

2. (CDR Sl)  -Regresa el CDR de Sl

```
*(CDR(QUOTE(A B C)))
>(B C)
```

```
*(CDR (QUOTE(   (A B C) X Y Z)))

>(X Y Z)

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

>( ( B )   ( C )   D )

*(CDR(CDR   (QUOTE( (( A ))   ( B )   ( C )   D )))

>(( C ) D )
```

Sea que L tenga el valor (  (EN VISTA)  (DE LO) SUCEDIDO)

```
*(CDR L)

>(   (DE LO)   SUCEDIDO)

*(CDR (CDR L))

>(SUCEDIDO)
```

3. (CONS S1 S2)  Regresa el CONS de S1  y S2

```
* (CONS   (QUOTE A)   (QUOTE   (B C)))

> (A B C)

* (CONS (QUOTE(A B))    (QUOTE(C D)))

>((A B) C D)
```

Sea que L1 tiene el valor JUAN y L2 (LUIS PEREZ)

```
*(CONS  L1 L2)

>(JUAN LUIS PEREZ)
```

4. (QUOTE S1) - Regresa S1 sin evaluar.

```
*(QUOTE(A B C))

>(A B C)

*(QUOTE X)

>X

*(QUOTE ( (A B)  (C D)))

>(  (A B)  (C D))
```

2. Aritméticas

1. (ADD1 N1) -Regresa el valor N1 + 1

```
*(ADD1  80)

>81

*(ADD1  -65)

>-64

*(ADD1  7)

>8
```

2.(SUB1 N1) -Regresa el valor N1-1

```
*(SUB1 80)
>79

*(SUB1 -65)
>-66

*(SUB1 7)
>6
```

3. Relaciones y lógicas

1 (ATOM S1) -Regresa T si S1 es un átomo, F si es otra cosa.

```
*(ATOM A)
>F
```

Ya que a A no se le ha asignado valor

Si a A le asignamos un valor MARIA

```
*(ATOM A)
>T
```

Si a A le asignamos el valor (JUAN PEREZ)

```
*(ATOM A)
>F

*(ATOM QUOTE A))
>T
```

2.(EQ A1 A2) -Regresa T si A1 es igual al valor A2, F si no lo son.

Sea que A1 tiene el valor de PERA y A2 el valor de PIÑA

```
*(EQ A1 A2)
>F
```

Si a A2 le asignamos ahora el valor PERA

```
*(EQ A1 A2)
>T
```

Si a A2 le asignamos (PERA)

```
*(EQ A1 A2)
>F          Ya que A2 no es átomo
```

Si A1 tiene el valor LECHE y L1 el valor (LA LECHE ES BUENA)

```
*(EQ A1 (CAR(CDR L1)))
>T
```

3. (NULL S1) - Regresa T si S1 es nula, F si no.

Si L1 tiene el valor (ESTA LISTA NO ES NULA)

```
*(NULL L1)
>F
```

Si L1 tiene el valor ()

```
*(NULL L1)
>T
```

Si L1 tiene el valor NULO

```
*(NULL L1)
>F
```

4. Operaciones de Asignación.

1. (SETQ A1 S1) -Pone el valor de A1 (sin evaluar) el de S1 regresando S1.

```
*(SETQ A (QUOTE(X Y Z)))
>(X Y Z)

*(SETQ  X (QUOTE A))
>A

*(SETQ X A)
>(X Y Z)

*(SETQ X Y)
>NILL          Ya que a Y no se le ha asignado ningun valor.
```

## 6.Control de Secuencia

1. (COND (S1 S2)
Evalúa las S-impares hasta
(S3 S4)
encontrar alguna que regrese
' '
el valor T, entonces ejecuta
' '
(Sn Sn) la S-par que le corresponde

```
*(COND
*((NULL L) ())
*(T  (CONS(CAR(CAR L))  (FIRSTS(CDR L)))) )
>(X Y Z)

*(COND
*((NULL LAT) ())
*((EQ (CAR LAT) OLD) (CONS OLD (CONS NEW (CDR LAT))))
*(T   (CONS  ( CAR LAT)  (INSERT OLD NEW (CDR LAT)))) )
>(A B C)

*(COND
*((EQ'A B)   T)
8(T  F))
>T
```

EQUAL
SETF
AREF
FIRST
DESFTYOC
REST
MAKE_ARRAY
MAPCAR

## 7. Otras

1. (LISTP S1) - regresa T si S1 es una lista, F si no.
Sea que L1 tiene el valor (ESTO ES UNA LISTA)

```
*(LISTP Ll)
>T
```

Sea que Ll tiene el siguiente valor (ESTO NO ES LISTA)

```
*(LISTP Ll)
>T
```

Sea que Ll tiene el siguiente valor LISTA

```
*(LISTP Ll)
>F
```

Sea que Ll tiene el valor ()

```
*(LISTP Ll)
>T
```

Hasta aquí, todas las funciones que han sído implementadas.

Todas estas funciones son las que forman el subconjunto de LISP creado.

Es posible dependiendo de las necesidades del sistema poder crear otras funciones de éstas, para agrandar el subconjunto, pero considero que con las que han sido implementadas ya se tiene una base sólida para poner en práctica los principales conceptos de LISP.

# VI. C O N C L U S I O N E S

- El computador Hewlett-Packard 1000 ya cuenta con un intérprete de LISP para aquellos alumnos que deseen iniciarse en el aprendizaje de este lenguaje.

- Este trabajo demuestra la factibilidad de la realización de proyectos de graduación a nivel alto como es la implementación de un intérprete.

- El proyecto puede ser completado por otro alumno para así tener el conjunto completo de LISP de la documentación y diseño modular contemplada en el trabajo.

- Podría considerarse como un programa que podría estar en la biblioteca de programas hechos por alumnos de la Universidad y en algún momento pueda intercambiarse con otras universidades nacionales o extranjeras.

- Puede ser utilizado en el laboratorio del curso Organización de Lenguajes de programación, que es el curso en el que generalmente se estudia este interesante lenguaje.

- Por el nivel que requiere la implementación de un Intérprete considero que llena los requerimientos como Modelo de Trabajo Profesional.


J
V
S

# VI. D I S C U S I O N

Si bien es cierto este trabajo no tiene implementadas algunas de las facilidades de LISP y posiblemente la eficiencia en la ejecución no sea la óptima, considero que el subconjunto creado es muy poderoso y su eficiencia es buena. Ya que este subconjunto forma el núcleo de cualquier interprete de LISP conteniendo las funciones básicas que hacen ver en LISP un lenguaje en la forma de manejo de datos simbólicos y en su estilo funcional.

La calificación como Modelo de trabajo Profesional se adapta perfectamente al trabajo realizado, ya que este proyecto es una réplica de una tarea que es ejecutada por profesionales, la implementación de un intérprete. Este proyecto no llega a ser una réplica exacta del intérprete (simplemente es un modelo) en vista que la implementación completa es una tarea bastante difícil, en la que generalmente intervienen varias personas especializadas (profesionales) y el tiempo que utilizan es considerable.

Sin embargo el intérprete esta diseñado para que pueda ser completado con facilidad por otro estudiante.

Las partes que faltarían de implementar en esta versión y que de ser construídas harán un LISP completo son:

-. Ampliar el subconjunto de funciones creadas hasta completar el conjunto total.

-. Implementar la estructura de datos Listas de Propiedad.

-. Implementar el "Colector de Basura"
-. Creación de la Instrucción PROG.

Estas cuatro partes pueden ser implementadas en el Intérprete ya que se encuentran las estructuras de datos necesarios para el diseño de sus algoritmos.

Existen dos formas para aumentar las funciones del Intérprete, una consiste en crear funciones escritas en LISP y la otra en el lenguaje que se implementó el Intérprete o sea en RATFOR.

La primera es bastante sencilla, únicamente bastaría con poner en un archivo las nuevas funciones escritas en LISP y en el momento en que el Intérprete da inicio la rutina INIT leería automaticamente el archivo y crearía las celda tipo SUBR en las funciones que fueron generadas y luego podrían ser invocadas por el usuario.

La segunda consistiría en los pasos siguientes:

1. Poner en la rutina RWORD (palabras reservadas) las funciones necesarias y asignarle un código de salida.

2. Alterar la rutina EXECUT que lo que hace es accesar el "stack" de ejecución tomando el código de cada función y dependiendo de este código hace uso de las funciones de bajo nivel como GCAR, GCDR, SCAR, SCDR, etc. para ejecturar la función requerida retornando el puntero a la Expresion-S resultante.

La parte que se complicaría un poco es la de la implemen

tación del "Colector de Basura". En la sección II (Descripción del Lenguaje) se hace mención de ciertas estructuras especiales así como de el reconocimiento de celdas activas que deben ser considerados en la implementación del "Colector de Basura", que deben ser leídas. Así también habría que leer alguna bibliografía sobre las distintas técnicas de implementación de este algoritmo. Sugerencias: Pratt, T. W. 1975.

# VIII. G L O S A R I O

1. Backgamon- Juego de mesa.

2. Backtracking- Proceso mediante el cual un programa recorre los nodos de un árbol hasta encontrar la solución óptima.

3. Bucketting- técnica utilizada en el "Hashing" para manejar colisiones por medio de un 'cubo o balde' donde se almacenan.

4. Buffer- área de almacenamiento temoral de datos.

5. Firmware- Parte del computador que es simulada por un microprograma en un "Hardware" microprogramable.

6. Loop- ciclo o interacción dentro de un programa.

7. Hashing- Desmenuzamiento, técnica que se utiliza para calcular direcciones de un elemento. Convierte en clave un número casi aleatorio, el que sirve después para determinar donde se almacena el elemento.

8. Hardware- Parte del computador compuesta por dispositivos físicos (alambres, circuitos, transistores, memorias magnéticas, etc).

9. Heap- Tipo de memoria estática basada en el manejo de "Stack". Consiste en un bloque de almacenamiento dentro del cual las piezas son colocadas y liberadas de una manera o estructurada.

10. Parser- (analizador sintáctico)- se encarga de la segunda fase de una traducción. Aquí las estructuras del programa son identificadas como instrucciones, declaraciones, expresiones, etc. usando los elementos ya descompuestos por el analizador de léxico.

11. Property Lists- parte de un átomo accesible por un señalador almacenado en la dirección de la memoria que representa al átomo.

12. Property Name- contiene el nombre de impresión de un atributo en un átomo.

13. Property Value- contiene el valor de un atributo en especial de un átomo.

14. Pop- Función que extrae el elemento en el tope del "stack".

15. Push- Función que agrega un elemento al tope del "stack".

16. Scanner- (analizador de léxico)-realiza la fase básica en una traducción que consiste en descomponer el programa fuente en elementos o "tokens" (identificadores, operadores, números,palabras claves, comentarios, delimitadores, etc.).

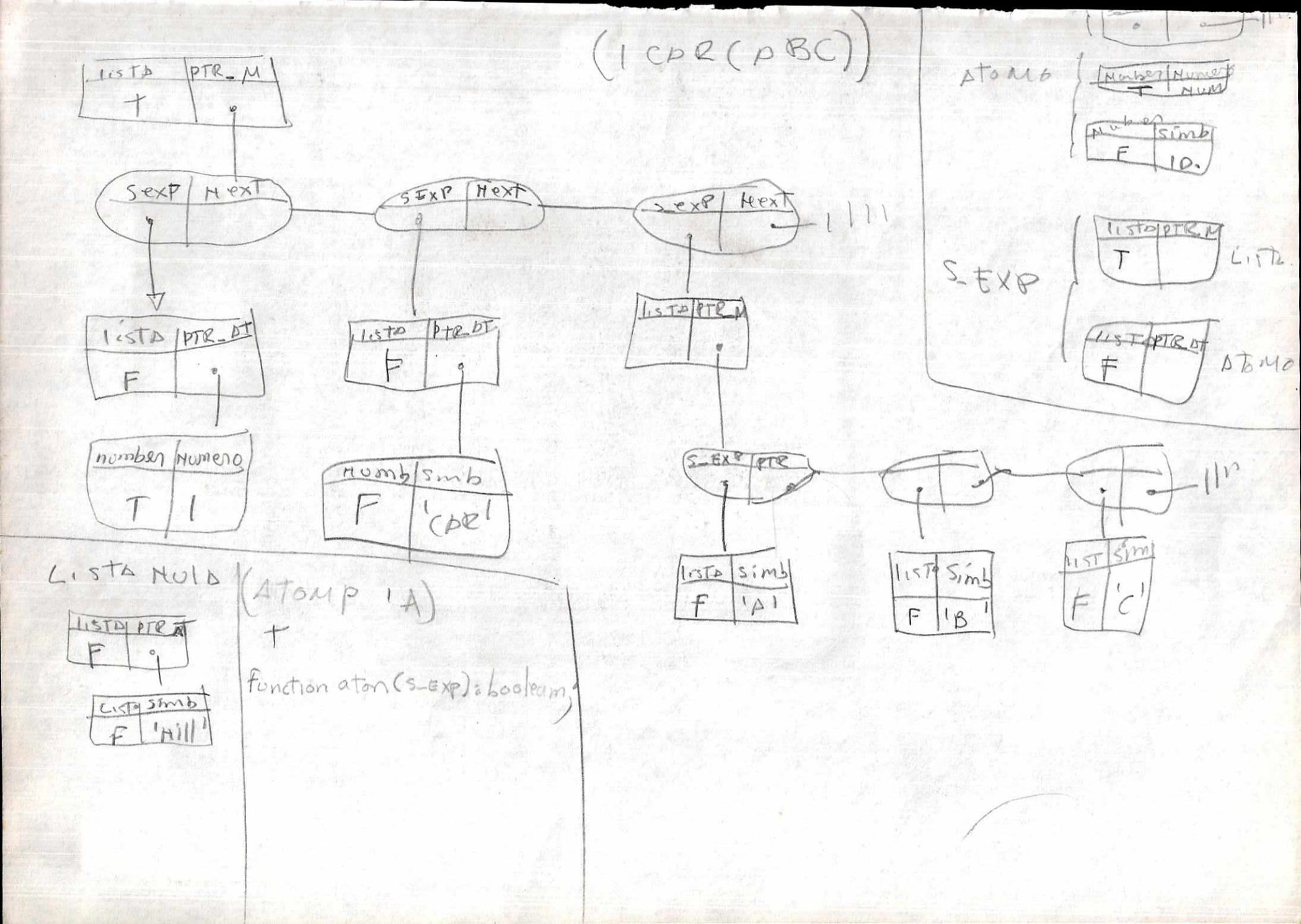
17. Software- parte del computador consistente en programas que facilitan la comunicación con los dispositivos físicos.

18. Stack- es la estructura para arreglos lineales de tamaño variable más simple. Es también conocida como LIFO (last-in, first-out, último en entrar, primero en salir.) y la inserción y eliminación de elementos está restringida por un señalador que indica el último elemento de la estructura.

19. String- (cadena) -serie de cero o más caracteres.

20. Token- elemento perteneciente a un léxico. Puede ser un operador, identificador,número, delimitador o una palabra clave.

21. Top-Down- técnica de programación consistente en dirigir la secuencia de ejecución de un programa de arriba hacia abajo.


(1 CDR (A B C))
LISTA | PTR_M
T
SEXP | NEXT
LISTA | PTR_AT
F
NUMBER | NUMERO
T | 1
LISTA | PTR_M
F
NUMB | SIMB
F | 'CDR'
S_EXP | NEXT
LISTA | PTR_M
S_EXP | PTR
LISTA | SIMB
F | 'A'
LISTA | SIMB
F | 'B'
LIST | SIMB
F | 'C'
ATOMO
NUMBER | NUMERO
T | NUM
NUMBER | SIMB
F | ID.
S_EXP
LISTA | PTR_M
T
LISTA
LIST | PTR_AT
F
ATOMO
LISTA NULA
LISTA | PTR_AT
F
LISTA | SIMB
F | 'NIL'
(ATOMP 'A)
function aton(S_EXP): boolean;

# B I B L I O G R A F I A


Aho.A.; J.D. Ullman. Principles of compiler design. Addis-
1977 on -Wesley Reading.

Jackowitz,P.;P.Segal. A LISP Interpreter en prime Fortran
1981 IV. Rensselear Polytechnic Institute, Troy, New York.

Friedman, D. The little LISPer. Bloomington, Indiana, Sci-
1974 ence Research Associates, INC.

Gries,D. Compiler construction for digital computers. Wiley
1971 International.

Haderson,P. Functional Programming, Aplication and Imple-
1980 mentation. Prentice-Hall International

McCarthy,J. LISP 1.5 Programmer's Manual. Cambridge, Mass.
1962 Massachusetts Institute of Technology, M.I.T. Press.

Pratt,T.W. Programming Languages:Design and implementation.
1975 USA Prentice-Hall International.

Siklossy,L. Lets talk LISP. Prentice Hall International.
1976

Winston P.H.; B.K.Horn. LISP. Addisson Wesley.
1981

Wexelblat L.R. History of Programming Languages. Pennsylvania,
1981 Academic Press.

```
CDR (L:ptr_SExp);
new(P1)
P1^.Listp:=t
P1^.PTR_M:=(L^.PTR_M^.NEXT)
RepLoop Echo(CDR(Listp))
```

```
CDR(L:ptr_S_Exp):ptr_S_ex
CDR^.Listp:=t
CDR^.PTR_M=L^.PTR_M^.N
new(P1)
P1=CDR(L)
```

Eval

Apply

Soporte. LTQ +1 +2