UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ALGORITMOS DE RECOLECCION DE BASURA PARA UN INTERPRETE DE LISP

T E S I S

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A:

ANA ELVIA OLMEDO PEREZ

MEXICO, D. F. 1984

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE.

0. INTRODUCCION.

No obstante que, antes de 1958, el propósito principal de una computadora digital fué el de realizar operaciones aritméticas, es a partir de 1958, que las computadoras fueron aplicadas cada vez más a problemas no numéricos, lo que originó que se empezara a idear la forma de cómo manejar la información simbólica. De ahí que surgieran preguntas tales como: Cuáles serían las operaciones básicas para efectuar un trabajo no numérico?, Cuáles serían las mejores formas de almacenar la información?, etc.

En respuesta a todas estas ideas se crearon los lenguajes de procesamiento de estructuras de datos ligadas que permiten el manejo de la información simbólica de una forma apropiada.

Dentro de las características fundamentales que constituyen a un lenguaje de procesamiento de listas, se encuentran:
La manipulación de símbolos, la no especificación por adelantado de los requerimientos de almacenamiento, la representación de las nuevas expresiones y el borrado de las antiguas.

Estas propiedades requieren un manejo dinámico de la memoria, lo que hace que surjan técnicas para administrar de forma eficiente su uso.

El objetivo del presente trabajo es introducirnos al estudio de una de estas técnicas, la que se denomina "Recolección de Basura", la cual fué inventada por John McCarthy al diseñar el lenguaje LISP.

Primero hacemos una presentación del problema y sus antecedentes, luego una referencia a las características del lenguaje LISP, mencionando las expresiones que utiliza, la representación interna de éstas, la organización de la memoria para esta representación, el tipo de manejo de memoria que emplea, los problemas que surgen al mantener el área de memoria, el problema de la basura, ejemplos de este problema, la recuperación del espacio que es basura, las diversas soluciones de recuperación que se han dado a lo largo del tiempo desde que surgió este tipo de manejo de memoria, los antecedentes, las etapas, el funcionamiento y los algoritmos de Recolección de Basura y las conclusiones.

Una parte de los resultados o conclusiones presentados en este trabajo están basados en la exposición de Knuth y en un artículo de Cohen, cuya referencia aparece en la bibliografía

del final del texto.

Los algoritmos objeto de estudio, que se refieren a la identificación del área que es basura, los programé en lenguaje C y los probé usando el intérprete de LISP que realizó el Dr. Miguel Tomasena, en la computadora PDP11/34 de la Facultad de Ciencias de la Universidad Nacional Autónoma de México.

Por último, deseo expresar mi agradecimiento al maestro Miguel Tomasena Fagoaga por haberme sugerido el tema y por su estimable ayuda en la elaboración de este trabajo.

También deseo manifestar mi reconocimiento a los maestros Christian Lemaitre León, Cristina Loyo Varela, Arturo Olvera y Elisa Viso Gurovich por haber revisado cuidadosamente el presente trabajo y por sus valiosas sugerencias y observaciones.

1. PRESENTACION DEL PROBLEMA.

El problema a tratar reside en identificar el área de memoria denominada basura, que está ocupada por los elementos que ya no están unidos a alguna estructura de datos de listas ligadas.

Una vez identificada esta área, se puede recuperar para dejarla nuevamente disponible para su uso.

Este problema surge cuando se agota el área de memoria disponible, pues ya no hay espacio para almacenar más estructuras de lista.

1.1. ANTECEDENTES.

En las estructuras de datos de listas ligadas, se da por supuesto que la memoria que está libre siempre está disponible para que la utilicen nuevos elementos que van a ser ligados a las listas.

Sin embargo, cuando se borra (se desliga) un elemento de la lista, la memoria en que estaba almacenado este elemento se pierde a menos de que se tomen las medidas para volverla a usar.

En los primeros lenguajes de procesamiento de listas se utilizaron las operaciones de manejo de listas ligadas, para controlar la memoria que se volvía disponible a través de borrar los elementos. Toda la memoria disponible se unía a través de apuntadores a una lista conocida como "lista de almacenamiento disponible". Cuando se borraban los elementos de las listas activas, la memoria no usada se insertaba al inicio de la lista de almacenamiento libre. La recuperación de la memoria de tales programas se especificaba casi siempre explícitamente por el programador.

En el lenguaje de programación LISP, se usó el método de Recolección de Basura. Todas las listas activas declaradas en este lenguaje se unen mediante apuntadores, de manera que en cualquier momento un programa puede rastrear los apuntadores y determinar todas las localidades de memoria activas. Cuando se agota el espacio disponible, se llama a la rutina de Recolección de Basura para hacer el rastreo, la cual sigue los apuntadores de la lista para marcar cada localidad de memoria con un bit especial en caso de ser un elemento de una lista activa. Todas las localidades no marcadas, que entonces se sabe que son basura, se regresan a la lista de almacenamiento libre.

2. CARACTERISTICAS DEL LENGUAJE LISP.

LISP (por LISt Processing) es uno de los diversos lenguajes de procesamiento de listas (IPL-V [New 60], FLPL [Gel 60], COMIT [Yng 63], SLIP [Wei 63]) que fueron diseñados para facilitar la representación y el procesamiento de estructuras de datos dinámicas, tales como listas ligadas y árboles binarios.

Este lenguaje fué diseñado por John McCarthy [McC 60], mientras estuvo en el Instituto de Tecnología de Massachusetts (MIT), en Nueva Inglaterra, EUA, donde él y su grupo de Inteligencia Artificial lo implantaron para la IBM 704, alrededor de 1960.

Se distinguen dos periodos durante el desarrollo del lenguaje LISP:

> Del verano de 1956 al verano de 1958, cuando la mayor parte de las ideas clave fueron desarrolladas.
>
> Del otoño de 1958 a 1962, cuando el lenguaje de programación fué implantado y aplicado a problemas de Inteligencia Artificial.

Las ideas por las que se caracteriza LISP, como lenguaje de programación, son las siguientes:

> El cálculo con expresiones simbólicas, en lugar de hacerlo con números.
>
> La representación interna de las expresiones simbólicas por medio de las estructuras de lista en la memoria de la computadora.
>
> Operaciones que seleccionan y construyen, expresadas como funciones, y la composición de éstas como una herramienta para formar más funciones.
>
> El uso de expresiones condicionales para obtener bifurcaciones en la definición de funciones.
>
> La definición de funciones recursivas mediante las expresiones condicionales.
>
> La capacidad de definir funciones (expresiones-lambda).
>
> La equivalencia entre datos y programas.
>
> La función "eval" que sirve como una definición formal del

lenguaje, y como un intérprete.

La técnica de Recolección de Basura como un medio de resolver el problema de recuperar el área que ocupan las estructuras de lista borradas (ya no accesibles).

Muchas de estas ideas fueron tomadas de otros lenguajes, pero la mayoría eran nuevas, tales como la técnica de Recolección de Basura, introducida por primera vez por McCarthy, las expresiones condicionales, la definición de funciones recursivas, etc.

Hacia el final del período inicial, se hizo claro que esta combinación de conceptos crearía un lenguaje de programación práctico, que se podría caracterizar como un lenguaje funcional, simbólico, procesador de listas, recursivo y lógico.

2.1. ORIGENES.

A continuación se describe el origen de las características de LISP:

1) La primera de ellas es el cálculo con expresiones simbólicas, en lugar de hacerlo con números.

Esta idea surgió porque existía la inquietud de hacer investigación sobre Inteligencia Artificial por varios investigadores, entre ellos John McCarthy, quién propuso el sistema denominado Tomador de Decisiones (Advice Taker) en noviembre de 1958.

Mediante dicho sistema, una máquina podría ser instruída para manejar cláusulas declarativas e imperativas y podría exhibir cierto sentido común para llevar a cabo sus instrucciones. Para ello McCarthy deseaba un lenguaje que pudiera manipular las expresiones que representaran estas cláusulas, para que el sistema Advice Taker pudiera hacer deducciones.

2) La siguiente es la representación interna de expresiones simbólicas por medio de estructuras de lista en la memoria de la computadora.

La noción de estructura de lista proviene de Newell, Simon y Shaw, quienes hicieron un trabajo extenso sobre este tipo de estructuras ligadas, y crearon una serie de lenguajes denominados IPL (Information Processing Language), en los que las únicas estructuras de datos disponibles eran las listas ligadas y las pilas (stacks) [New 60].

Newell, Simon y Shaw, describieron el lenguaje IPL 2 en la

Conferencia de Investigación del Verano de Dartmouth sobre Inteligencia Artificial en 1956. Esta conferencia fué el primer estudio organizado sobre dicho tema.

Las características del lenguaje IPL son: la representación de la información a través de listas ligadas, que significa que el tamaño de la información es variable, la posibilidad de añadir y suprimir elementos de en medio de la lista, que LISP rara vez usa y la idea de subrutinas recursivas usando una pila.

El lenguaje IPL era un lenguaje de máquina. Este lenguaje de procesamiento de listas se implantó en la computadora JOHNNIAC de la corporación RAND en 1957 [Wex 81].

Esta serie de lenguajes influenciaron fuertemente a los subsecuentes, para manejar información no numérica.

Fué en la conferencia de Dartmouth donde surgió la idea de John McCarthy de realizar un lenguaje de procesamiento de listas, para su trabajo de Inteligencia Artificial e implantarlo en la computadora IBM 704.

Existían dos motivos para desarrollar el lenguaje LISP en la máquina IBM 704.

> IBM estaba instalando un Centro de Computación en Nueva Inglaterra en el MIT.
>
> IBM estaba comprometiendose a desarrollar un programa para desarrollar teoremas sobre Geometría Plana, del cual McCarthy era asesor.

Además, en ese tiempo, IBM se veía como algo seguro para seguir vigorosamente la investigación sobre Inteligencia Artificial y se esperaban proyectos adicionales.

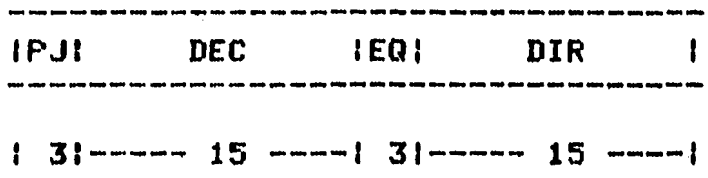
La representación de las cláusulas del sistema Advice Taker, de McCarthy, mediante las estructuras de lista parecía apropiada - todavía lo es - y un lenguaje de procesamiento de listas también parecía adecuado para programar las operaciones implicadas en la deducción - todavía lo es también -.

Esta representación interna de información simbólica promovió el cambio de la notación infija, en favor de la notación prefija, que simplifica la tarea de programar los cálculos esenciales, como son, la deducción lógica o la simplificación algebraica, la diferenciación, o la integración.

El primer problema a resolver era como representar las estructuras de lista en la memoria de la máquina IBM 704.

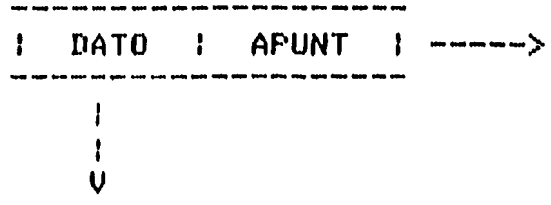
Esta computadora tenía una palabra de 36-bits y su dirección era de 15-bits, de manera que era recomendable que la estructura de lista tuviera apuntadores de esta longitud.

La palabra de 36-bits estaba dividida en cuatro campos: la dirección, el decremento, el prefijo y la etiqueta. Las dos primeras eran de 15 bits cada una, y cada una de las dos últimas eran de tres bits, las que se encontraban separadas una de otra por el decremento.



Un nodo de una estructura de lista, se representaría entonces mediante una palabra de la computadora, usando los campos de decremento y dirección, ya que cada nodo se consideraba dividido en dos partes que correspondían cada una a 15 bits de la palabra. Una parte sería la del dato y la otra la del apuntador, que apuntaría a la siguiente palabra, esto es, que diera la dirección de la siguiente palabra en la estructura de lista.

El elemento dato de la lista podía ser por si mismo un apuntador a una lista, y por lo tanto una lista como un todo, el cual se representaría mediante la localidad del primer elemento de la lista a la que apuntara.



3) Las operaciones que seleccionan y construyen, expresadas como funciones, y la composición de éstas como una herramienta para formar más funciones.

En este momento, sólo existía la duda sobre cuales operaciones básicas debería de haber, pues la operación de seleccionar una parte de la palabra mediante una máscara, se consideraba aparte de la operación de elegir el contenido de una palabra en memoria como una función de su dirección.

Lo que se quería hacer era pensar en un registro y en una operación como una función de su localidad.

El primer conjunto de funciones propuesto incluía "cwr" que significa "Contents of the Word in Register number" y cuatro funciones que extrajeran las partes de la palabra:

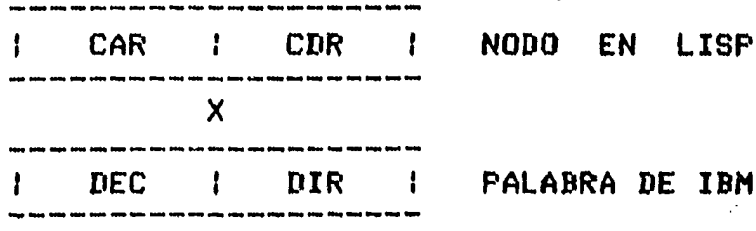
"car" (Contents of the Adress Part of the Register number),
"cdr" (Contents of the Decrement Part of the Register number),
"ctr" (Contents of the Tag Part of the Register number),
"cpr" (Contents of the Prefix Part of the Register number),

Las partes de dirección y decremento se manipulaban por instrucciones especiales para mover su contenido a y de los registros índice de 15-bits, lo que hizo más fácil la implantación de las operaciones "car" y "cdr".

En la definición de un nodo de la estructura de lista, se dió el nombre de "car" a la parte izquierda y el de "cdr" a la parte derecha.

El "car" apunta al primer elemento de la estructura de lista y el "cdr" apunta al resto de los elementos de la lista.

El orden de como aparecen los campos "car" y "cdr" en la definición de un nodo de una estructura de lista en LISP, refleja al lenguaje ensamblador SAP de la IBM y no a la palabra de la máquina.



La operación de construcción fué idea de Gelertner. Esta operación consiste en construir nuevas estructuras de lista.

El grupo que estaba a cargo del proyecto sobre Geometría Plana, del cual John McCarthy era asesor, empezó a generar nuevas ideas, una de las cuales, la mejor según McCarthy, se reflejó en LISP. Esta era la operación de construcción, que era necesaria para crear nuevas estructuras de lista.

Esta operación se definió primero como una subrutina: "cons(a,d,p,t)", (donde a es la dirección, d es el decremento, p es el prefijo y t es la etiqueta), y no como una función con un valor.

Ahora bien, el proyecto de Geometría, que era idea de Minsky [Wex 81], requería de cálculos simbólicos, representados por estructuras de lista, los que expresarían a las figuras geométricas, y también a las afirmaciones acerca de dichas figuras, tales como: "los dos lados son iguales", y a otras similares.

Nathaniel Rochester de la IBM (quién en la Conferencia de Investigación del Verano de Dartmouth sobre Inteligencia

Artificial contrató a Herbert Gelertner para desarrollar el trabajo de geometría) decidió Junto con Gelertner y con la ayuda de McCarthy, implantar un lenguaje de procesamiento de listas usando FORTRAN, porque ésta parecía la forma más fácil de empezar.

Dicho trabajo lo emprendieron Herbert Gelertner y Carl Gerberich en la IBM y desarrollaron al final el sistema FLPL, que significa FORTRAN List Processing Language, [Gel 60].

Fué entonces que Gelertner y Gerberich se dieron cuenta de que "cons" debería de ser una función, y no una subrutina, cuyo valor sería la localidad de la nueva palabra que se tomara de una lista de almacenamiento disponible ("free list").

Esto además permitiría construir nuevas expresiones de subsubexpresiones, mediante la composición de las ocurrencias de "cons".

Las expresiones se podían manejar fácilmente en FLPL, lenguaje que fué usado en forma satisfactoria para el programa de Geometría, con el único inconveniente de que no tenía expresiones condicionales, ni recursión, y el borrado de las estructuras de lista era manejado explícitamente por el programa.

Las siguientes características que forman parte del lenguaje LISP, fueron creadas por John McCarthy.

4) El uso de expresiones condicionales para obtener bifurcaciones en la definición de funciones.

Al estar haciendo un programa sobre ajedrez en FORTRAN, McCarthy deseaba usar una instrucción semejante a la del IF que existía en el lenguaje ALGOL (if p then a else b) para tomar decisiones dentro del programa, pero sólo podía utilizar el IF de FORTRAN, el cual le resultaba poco conveniente porque le era difícil seguir su funcionamiento, que era: si una expresión es menor que cero, entonces haz esto, si es igual a cero, haz lo otro y mayor que cero, haz otra cosa.

Para él esto era una fuente de equivocación, lo que lo llevó a crear una instrucción nueva, así que, inventó un IF para FORTRAN de tres argumentos -XIF(P,A,B)- cuyo valor era A o B según si la expresión P valía cero o no.

Esta función mejoró la programación de muchos problemas y los hizo más fáciles de entender, pero se tenía que usar con cuidado, porque los tres argumentos debían de evaluarse antes de que se ejecutara la función XIF, ya que ésta era llamada como una función de FORTRAN.

Esto condujo a dos cosas: a la invención de la expresión

condicional lógica, que sólo evalúa uno de los argumentos A o B, de acuerdo a si la expresión P es verdadera o falsa y al deseo de un lenguaje de programación que permitiera su uso.

Las expresiones condicionales permiten combinar los casos en una sola fórmula.

5) La definición de funciones recursivas mediante las expresiones condicionales.

Una función recursiva es una función definida por una expresión condicional, parte de la cual requiere de una evaluación de la función completa (para diferentes valores de sus argumentos). La recursividad era la única estructura de control de secuencia.

6) La capacidad de definir funciones (expresiones-lambda).

McCarthy utilizó el nombre lambda para definir funciones. Dicho nombre se refiere a la notación-lambda de Church.

Además se incluyeron funcionales o funciones cuyos argumentos son funciones.

7) La equivalencia entre datos y programas.

La representación es la misma para datos o para funciones, lo que permite manejar los programas como datos o ejecutar datos como programas.

8) La función "eval" que sirve como una definición formal del lenguaje, y como un intérprete.

La utilización de una función que evalúa (eval), la cual interpreta su argumento como una representación de lista para una función y ejecuta la función. Por lo que los programas son completamente modificables, en el sentido de que, al tiempo de ejecución, se pueden cambiar o crear las definiciones de las funciones, para luego aplicarlas a los argumentos.

9) La técnica de Recolección de Basura como un medio de resolver el problema de recuperar el área que ocupan las estructuras de lista borradas (ya no accesibles).

En las primeras versiones de LISP, la recuperación era explícita. De hecho, existía una función llamada "ERALIS" (ERAse List). Después, esta forma de recuperación fué reemplazada por una mejor idea: la técnica denominada "Recolección de Basura".

Este mecanismo de manejo de memoria empieza a funcionar cuando se acaban las palabras de la lista de almacenamiento disponible y consiste en recuperar las localidades de memoria que están ocupadas por los elementos borrados (elementos basura), para poderlas recuperar y regresarlas al área disponible para su reuso.

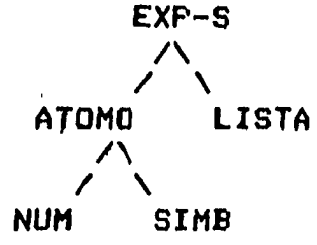
Una vez reunidas estas ideas, se procedió a implantarlas dando lugar a la creación del lenguaje LISP.

2.2. ELEMENTOS DEL LENGUAJE.

Los elementos del lenguaje son las expresiones simbólicas y las funciones primitivas.

2.2.1. EXPRESIONES SIMBOLICAS.

McCarthy denominó a los elementos del lenguaje, que son los átomos y las listas, como expresiones simbólicas (expresión-S).



Los átomos son los elementos básicos de información, que originalmente McCarthy [McC 60] llamó objetos. Los átomos pueden ser simbólicos o numéricos.

La sintáxis de los nombres de los átomos es:

```
<atomo>        ::=  <atomo simb> / <numeral> / - <numeral>
<atomo simb>   ::=  <letra> / <atomo simb> <letra>
               ::=  <atomo simb> <digito>
<numeral>      ::=  <digito> / <numeral> <digito>
<letra>        ::=  A/B/C ... /Z
<digito>       ::=  0/1/2 ... /9
```

Ejemplos: MKL, A3, PTR, ANN, 789, CAR, PLUS, QUOTE.

Las listas se definen como una secuencia ordenada de átomos o listas, separados por comas o por espacios y encerrados entre paréntesis.

La sintáxis de la lista es:

<lista> ::= (<elem lista>, ..., <elem lista>)
<elem lista> ::= <atomo> / <lista>

Una lista especial es la lista vacía, la cual no tiene elementos y se representa así: "()",

Ejemplos:

x = (A, B, C) (1)

Es una lista, donde A, B, y C son átomos.

y = (A ((B C D) E F) (G H) I) (2)

Es una lista cuyos elementos son átomos y listas. El segundo elemento es una lista de tres elementos (uno es la lista de los átomos B, C y D y los otros dos son los átomos E y F).

Un árbol binario se puede representar por medio de la notación con punto, donde (A.B) es un par ordenado (dotted-pair) de los átomos A y B. En lugar de A o B (o ambos), el usuario puede colocar cualquier par ordenado, etc., y definir así un árbol binario.

La notación de lista y la notación de punto se pueden usar indistintamente, en el entendimiento de que una lista (S1, S2, ..., Sn) es equivalente al par con puntos:

(S1. (S2. ...(Sn.NIL)...))

El átomo NIL desempeña un papel especial como terminador de las listas (se define como equivalente a la lista vacía).

2.2.2. FUNCIONES PRIMITIVAS.

Así como toda el álgebra se puede construir a partir de unas cuantas operaciones primitivas sobre números (como la adición) así también podemos formar un esquema de procesamiento de información completo para las estructuras de lista a partir de unas operaciones básicas, entre las cuales se encuentran las siguientes:

atom, car, cdr, eq, cons y cond

Cada una de estas operaciones se puede considerar como una función, (ver apéndice).

A partir de estas funciones, se pueden construir prácticamente todas las funciones computables del lenguaje.

Las funciones booleanas "atom" y "eq" se pueden usar como proposiciones en la función "cond", y esta puede proporcionar más funciones de verdad. Las funciones "car" y "cdr" se usan para dividir una lista en partes, y "cons" se usa para construir nuevas listas de las partes dadas (ver apéndice).

Se debe hacer notar que la elección de los nombres de las funciones es arbitraria y mientras que la mayoría son abreviaciones, "car" y "cdr" fueron nombres escogidos por McCarthy por su significado en relación a una computadora en particular.

Un programa en LISP se puede pensar siempre como un problema de evaluación de funciones. La evaluación de una función de una expresión-S da otra expresión-S.

Los programas, entonces, consisten de funciones. Las funciones básicas del lenguaje (ver apéndice) producen valores que se obtienen fácilmente de sus argumentos, por ejemplo, el valor puede ser una subexpresión determinada de un argumento, o una lista de los argumentos.

3. REPRESENTACION INTERNA.

La representación interna de los elementos del lenguaje LISP, en la memoria de la computadora, hace uso de las estructuras de datos denominadas listas ligadas.

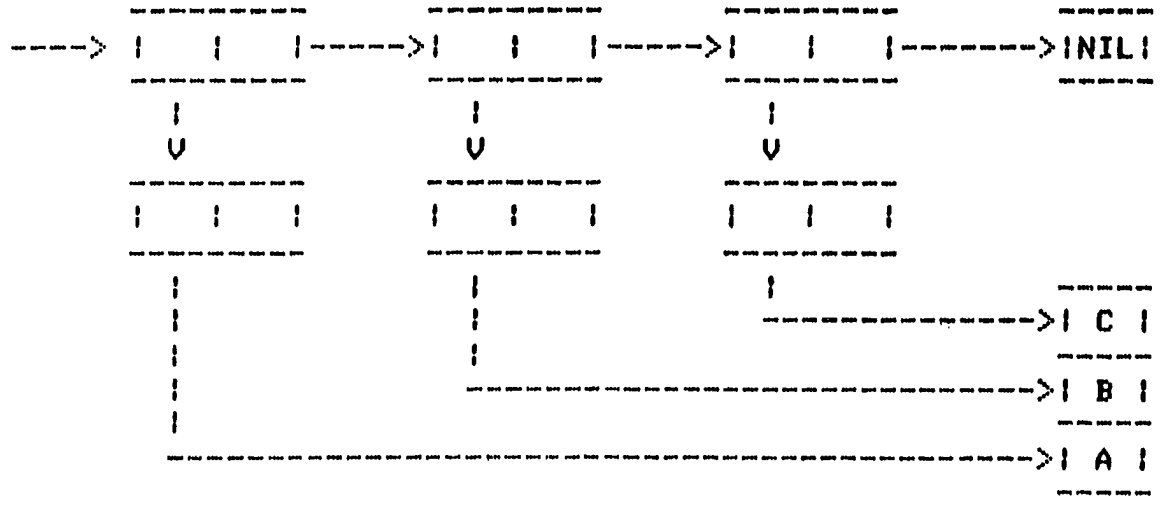
Antes de ver el tipo de memoria que utiliza la implantación del intérprete de LISP que estamos usando ([Tom 83], [Tuc 79]), estudiaremos la forma en que se realiza la representación interna de estos elementos.

3.1. REPRESENTACION DE LAS EXPRESIONES SIMBOLICAS.

Sabemos que una expresión simbólica puede ser un átomo o una lista y que si es un átomo, éste puede ser simbólico o numérico.

Veamos cual sería la representación interna de una expresión que es una lista, y luego trataremos a cada elemento por separado.

Por ejemplo, la expresión simbólica (A B C) se representa de la siguiente manera:



Esta representación gráfica tiene una equivalencia directa en la memoria de la computadora.

Notemos que hay dos tipos de "cajas" que representan a las localidades de la memoria.

Cada "caja-doble" representa a un elemento de una estructura de lista que contiene dos apuntadores.

Cada "caja-sencilla" representa una localidad que contiene el nombre de un átomo simbólico.

Como podemos ver en la figura, las "cajas-dobles" se agrupan en una área de memoria, mientras que las "cajas-sencillas" se agrupan en otra. Esto se hace para obtener una mejor organización de la memoria.

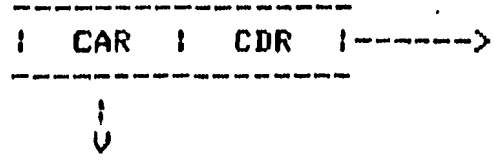
A una "caja-doble" la llamaremos localidad de apuntadores (dotted-pair) y a una "caja-sencilla" la denominaremos localidad de nombre del átomo.

En la localidad de apuntadores, la dirección de la izquierda representará al apuntador CAR y la dirección de la derecha representará al apuntador CDR.

Si la palabra en la memoria de la computadora es suficientemente grande, los campos de direcciones CAR y CDR se deben de empacar en una sola palabra (como en la implantación original de LISP en la IBM 704).

Sin embargo, en algunas máquinas, una palabra o localidad puede no ser suficiente para almacenar dos direcciones, como es el caso de la PDP/11, cuya palabra es de 16-bits, por lo que se requieren dos palabras de la memoria de la máquina para representar un elemento de una estructura de lista.

Los apuntadores referenciarán localidades de nombres de átomo o apuntarán a otras localidades de apuntadores.



Esta representación interna de las expresiones simbólicas se llama listas simplemente ligadas, que es un caso especial del esquema denominado estructuras de listas ligadas.

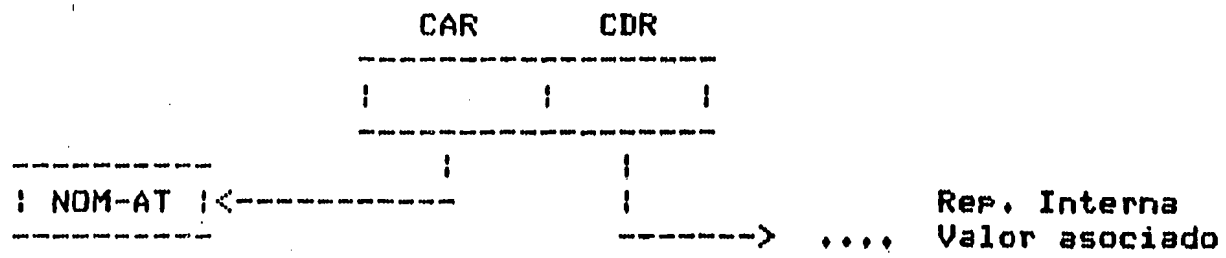
El término "simplemente ligada" se refiere al hecho de que para encontrar los siguientes elementos en la representación, debemos seguir los apuntadores, los cuales nos indican como encontrar los elementos sucesores de la estructura.

Las listas simplemente ligadas son el tipo más simple de estructuras de datos dinámicas y son las básicas para LISP.

3.1.1. REPRESENTACION DE ATOMOS SIMBOLICOS.

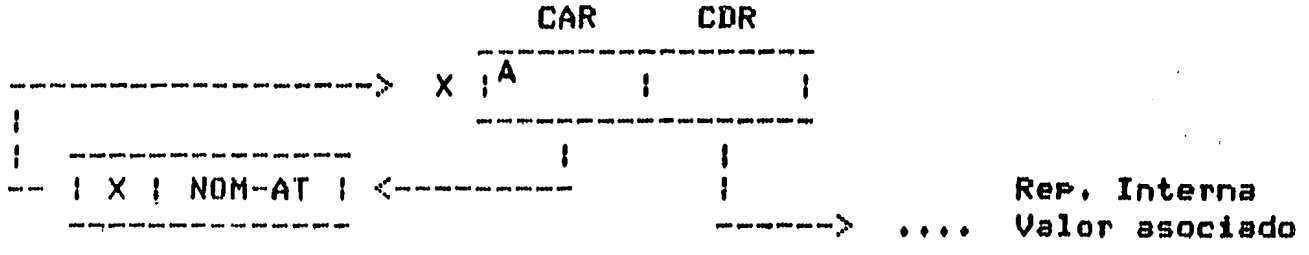
Para representar un átomo simbólico se utiliza una localidad de apuntadores y una localidad de nombre de átomo.

La localidad de apuntadores sirve para guardar en el CAR la dirección a la localidad de nombre de átomo, y en el CDR la dirección de la localidad que inicia la representación interna del valor asociado al átomo.



Cada localidad de nombre de átomo tiene asociada otra localidad, que contiene la dirección de la localidad de apuntadores que representa al átomo dado.

De esta manera, la localidad de apuntadores que representa al átomo, puede ser accesada a través de la localidad de dirección que está asociada a la localidad del nombre del átomo y viceversa.



Ahora bien, cada átomo simbólico se representa una sola vez en la memoria.

Esto es, si hay dos átomos con el mismo nombre, su representación ocupará la misma localidad de apuntadores, la cual apuntará a la misma localidad del nombre del átomo.

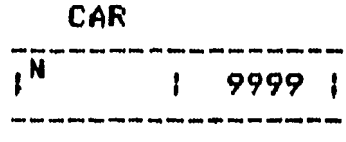
Se dice que la localidad de apuntadores que representa a un átomo simbólico es un átomo simbólico y se señala mediante el indicador A que aparece en la "caja", el cual es un bit que se prende en la palabra.

Los átomos simbólicos siempre se conservan, nunca se borran, por lo que las localidades de apuntadores que los representan siempre existen, es decir, siempre son accesibles por el intérprete.

3.1.2. REPRESENTACION DE ATOMOS NUMERICOS.

Los átomos numéricos se representan en una localidad de apuntadores.

Esta localidad contiene en el lugar del CDR, el valor del átomo numérico.



donde 9999 indica el valor de un átomo numérico dado.

Se dice que esta localidad es un átomo numérico, y se señala mediante el indicador N que aparece en la "caja". Tal indicador es un bit que se prende en la palabra.

Un átomo numérico no se representa de manera única en la memoria a diferencia de un átomo simbólico.

Los átomos numéricos pueden dejar de usarse, por lo que su espacio se debe de recuperar para su reuso.

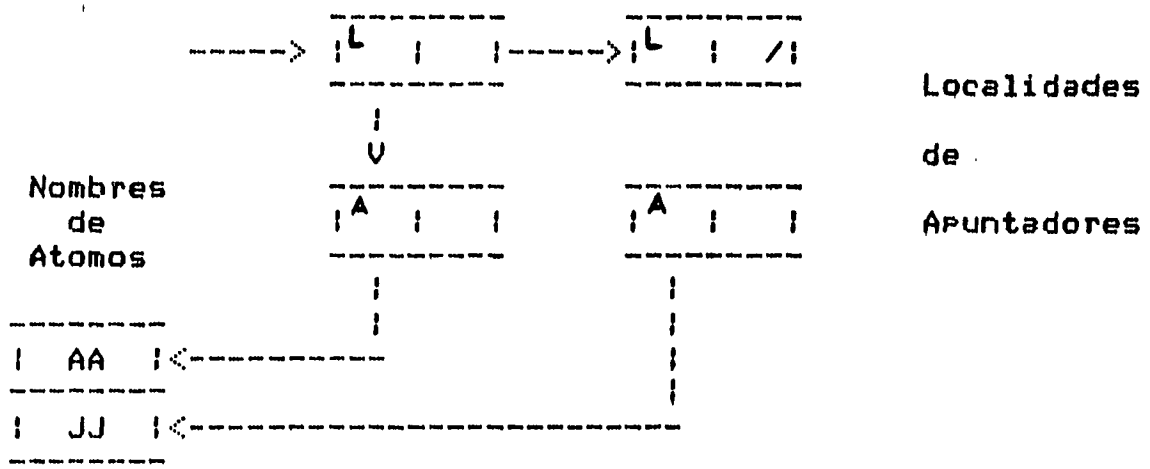
3.1.3. REPRESENTACION DE LISTAS.

Las listas se representan en localidades que contienen dos apuntadores, el CAR y el CDR. El primer apuntador, CAR, apunta al primer elemento de la lista, y el segundo, CDR, apunta al resto de la lista.

Por ejemplo:

La lista (AA JJ) se representa por cuatro localidades de apuntadores, dos de las cuales son átomos simbólicos.

El CAR de la primera localidad, es la dirección de la localidad que apunta al átomo AA, y el CDR es la dirección que apunta al resto de la lista.

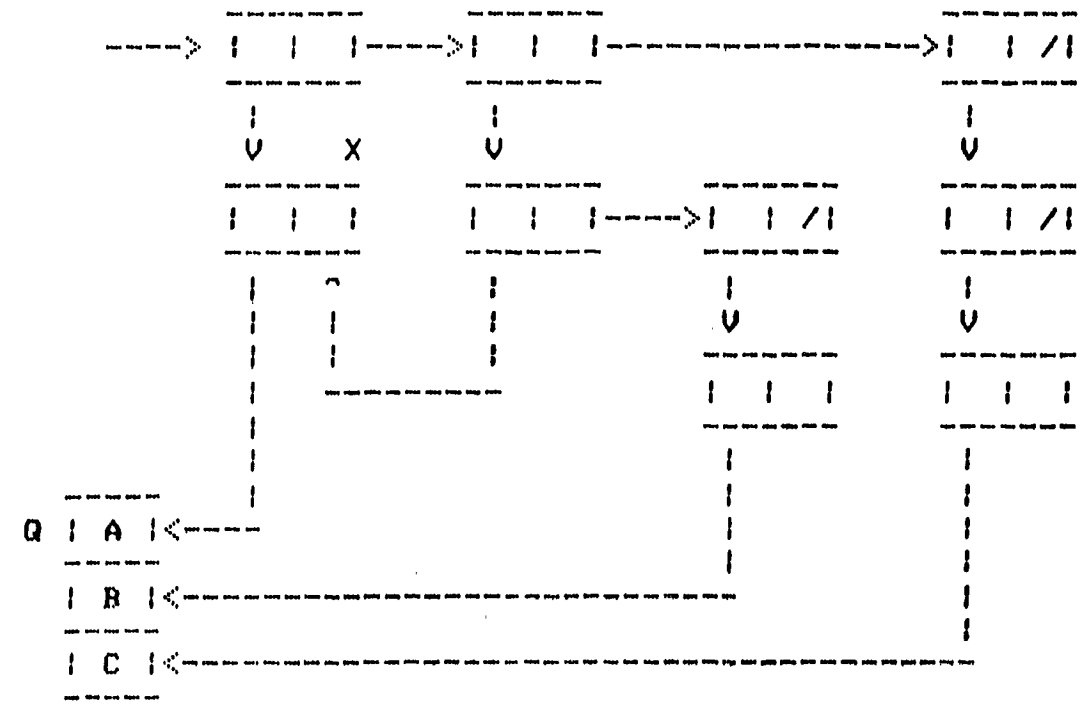


Una caja con una línea diagonal contiene la dirección de la localidad donde está contenido el nombre del átomo NIL, que indica el final de la expresión simbólica. El átomo NIL es equivalente a la lista vacía "()".

Las localidades de apuntadores que apuntan a los primeros elementos de lista y las que apuntan al resto de la lista, se denominan localidades de lista y se señalan con una L dentro de la "caja". Para tal indicación también se marca la palabra, apagando los bits de A y N.

Ahora mostraremos un ejemplo de una lista con dos átomos simbólicos con el mismo nombre, para mostrar que estos átomos se representan de manera única en la memoria.

Sea la lista (A (A B) (C)), cuya representación es:



De esta forma, podemos ver que cada referencia al átomo A es un apuntador a la localidad X, que apunta a la localidad Q, donde está el nombre del átomo A.

Como vimos las localidades de apuntadores, además de contener apuntadores, tienen indicadores para señalar hacia que tipo de elementos apuntan éstos.

Estos indicadores son bits de la palabra, los cuales indican si los apuntadores apuntan a átomos simbólicos, o a listas, o bien si la localidad contiene, en lugar del CDR, un átomo numérico.

Además de estos indicadores, las localidades de apuntadores cuentan con otros dos. Uno para utilizarlo en el proceso de marca de la Recolección de Basura y otro para señalar la alteración de apuntadores, que se hace en los algoritmos de Recolección de Basura que invierten apuntadores.

En el diseño del intérprete de la PDP-11/34 [Tom 83], se asignaron dos palabras de la memoria para representar un nodo de una estructura de lista. Una palabra corresponde al campo CAR y la otra al campo CDR.

En cada campo, los bits 0-11 sirven para guardar los valores de los apuntadores correspondientes, y los bits siguientes para representar a los indicadores.

bit 15 del campo CAR	-	Indica átomos simbólicos.
bit 14 del campo CAR	-	Indica átomos numéricos.
bit 13 del campo CAR	-	Bit de marca para la Recolección de Basura.
bit 13 del campo CDR	-	Indica alteración de apuntadores.

Los bits 15 y 14 están apagados cuando se trata de representar una localidad de apuntadores que no es átomo (que no es un nodo hoja del árbol binario).

3.2. REPRESENTACION DE LAS FUNCIONES PRIMITIVAS.

En la representación interna de las funciones primitivas hay que tomar en cuenta el nombre de la función y el tipo de la misma.

Estas funciones son de dos tipos, dependiendo de como se evalúen sus argumentos:

SUBR - Evalúan primero sus argumentos y luego aplican la primitiva.

NSUBR - Aplican directamente la primitiva, sin evaluar necesariamente sus argumentos.

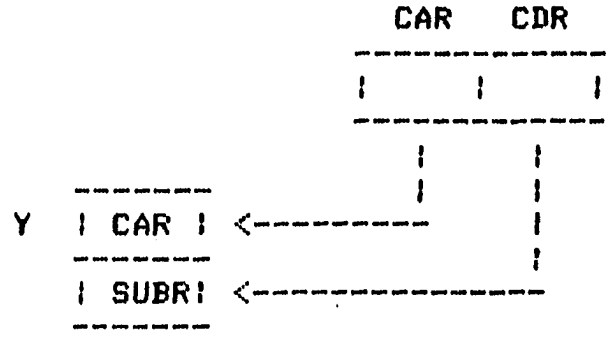
Los nombres de las funciones primitivas y el nombre del tipo que les corresponde se consideran como átomos simbólicos, por lo que su representación es similar a la de estos átomos.

Así cada uno de los nombres de los átomos NSUBR y SUBR se representan mediante una localidad de apuntadores y una localidad de nombre del átomo.

Ahora bien, a cada una de las funciones primitivas se les asocia su tipo, lo cual se representa mediante una localidad de apuntadores, cuyo CAR apunta a la localidad donde está el nombre de la primitiva, y cuyo CDR apunta a la localidad donde está el nombre del átomo SUBR o NSUBR, según sea el tipo de la función.

Por ejemplo:

La función primitiva CAR, que es del tipo SUBR se representa así:



Las localidades de apuntadores que representan a las funciones primitivas y a sus tipos, así como a otros átomos significativos para LISP, como son los paréntesis izquierdo y derecho y los valores de verdad T y NIL, son localidades fijas, y se definen en la parte de inicialización del intérprete.

Las direcciones de estas localidades de apuntadores permanecen constantes durante la ejecución del intérprete.

El código de ejecución de cada función que aparece en el intérprete escrito en C, tiene asociado el número de la dirección de la localidad que la representa (por ejemplo el código de CAR tiene asociado la dirección Y de la localidad de apuntadores donde está representada la función), así el intérprete evalúa esta función siguiendo el procedimiento descrito en el código que tiene asociado el número Y.

El ciclo de ejecución del intérprete consiste en leer cada expresión simbólica, en evaluarla y en escribir el resultado de la evaluación. El ciclo termina cuando el intérprete lee la expresión (EXIT).

```
while (expresion <> (EXIT))
   begin
      lectura;
      evaluación;
      escritura;
   end;
```

En la lectura se efectúa la representación interna de la expresión simbólica leída.

En la evaluación se representa el valor de la expresión.

En la escritura, el intérprete imprime el valor de la expresión, siguiendo los apuntadores de la representación interna de éste.

4. ORGANIZACION DE LA MEMORIA.

La memoria que usa la implantación del intérprete de LISP que estamos utilizando ([Tuc 79], [Tom 83]), es la memoria principal, la cual es estática y fija.

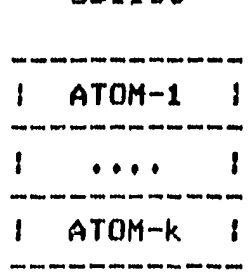
La memoria se divide en varias áreas, según el uso que el intérprete haga de las localidades de la misma para efectuar la representación de las expresiones simbólicas.

Lo que nos lleva a la siguiente organización de la memoria, que consiste en tener diversas áreas, en las que se clasifican las localidades de memoria por su diferente uso.

- Area de objetos.
- Area de apuntadores.
- Area para llamados recursivos.
- Area para elementos auxiliares y en construcción.

4.1. AREA DE OBJETOS.

El área de objetos llamada OBLIST, es una lista ligada en la que se almacenan los nombres de los átomos simbólicos, tanto los que tienen significado para LISP, como los que se definen por el usuario.



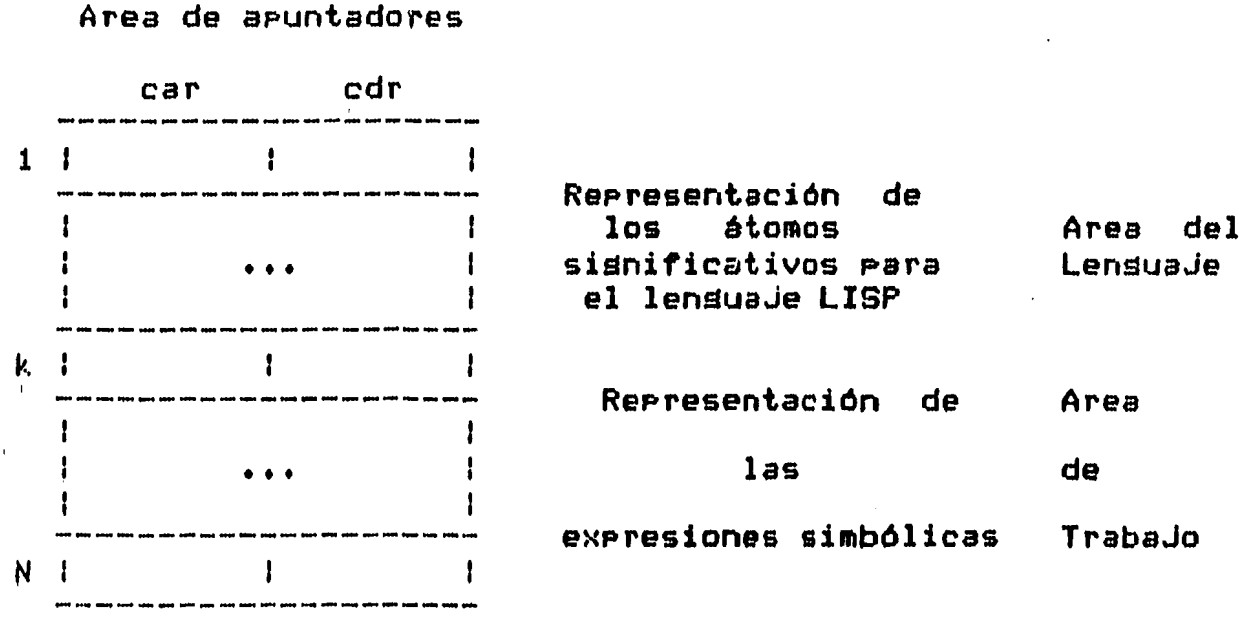
4.2. AREA DE APUNTADORES.

El área de apuntadores es la lista de N localidades de memoria que contienen dos apuntadores, las cuales se utilizan para la representación de los elementos de las estructuras de listas ligadas.

Esta área se divide en dos partes:

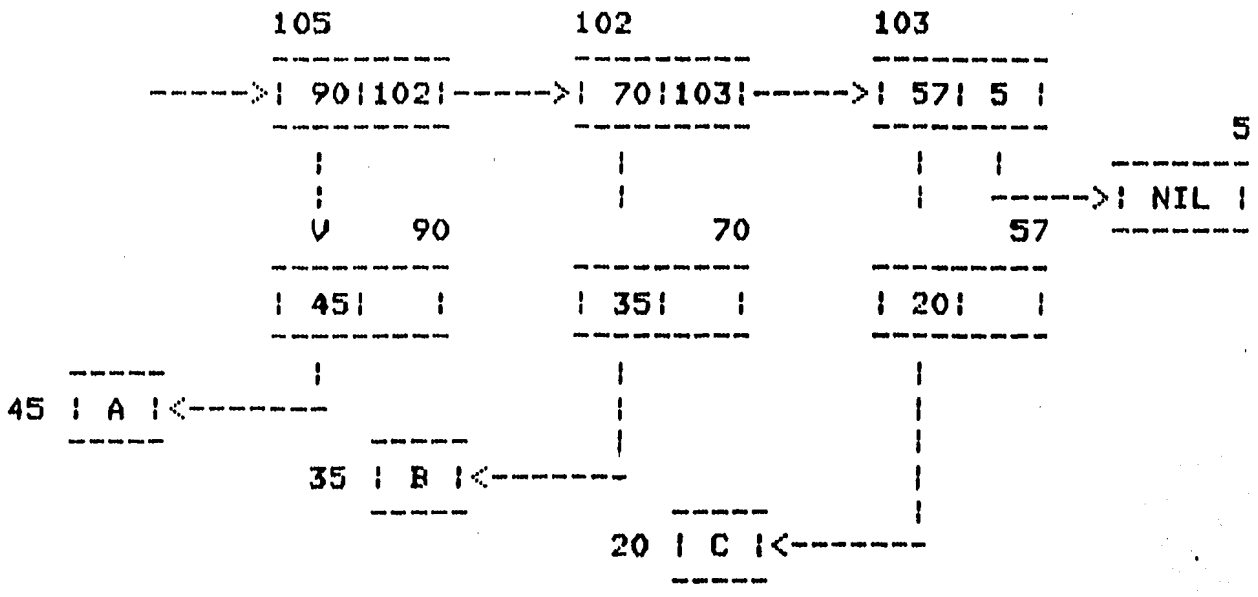
a) Area del lenguaje.- Para almacenar la representación de los átomos que corresponden a las funciones primitivas, a los tipos de las funciones, las constantes T y NIL, y a los paréntesis (y). Esta área ocupa las primeras k-1 localidades del Area de Apuntadores, donde k es un número cualquiera.

b) Area de Trabajo.- Para almacenar la representación de las expresiones simbólicas, la cual ocupa las N-k+1 localidades restantes del Area de Apuntadores, en donde se van a almacenar las estructuras de lista. Esta área se maneja a través de la lista disponible y de la técnica de Recolección de Basura.

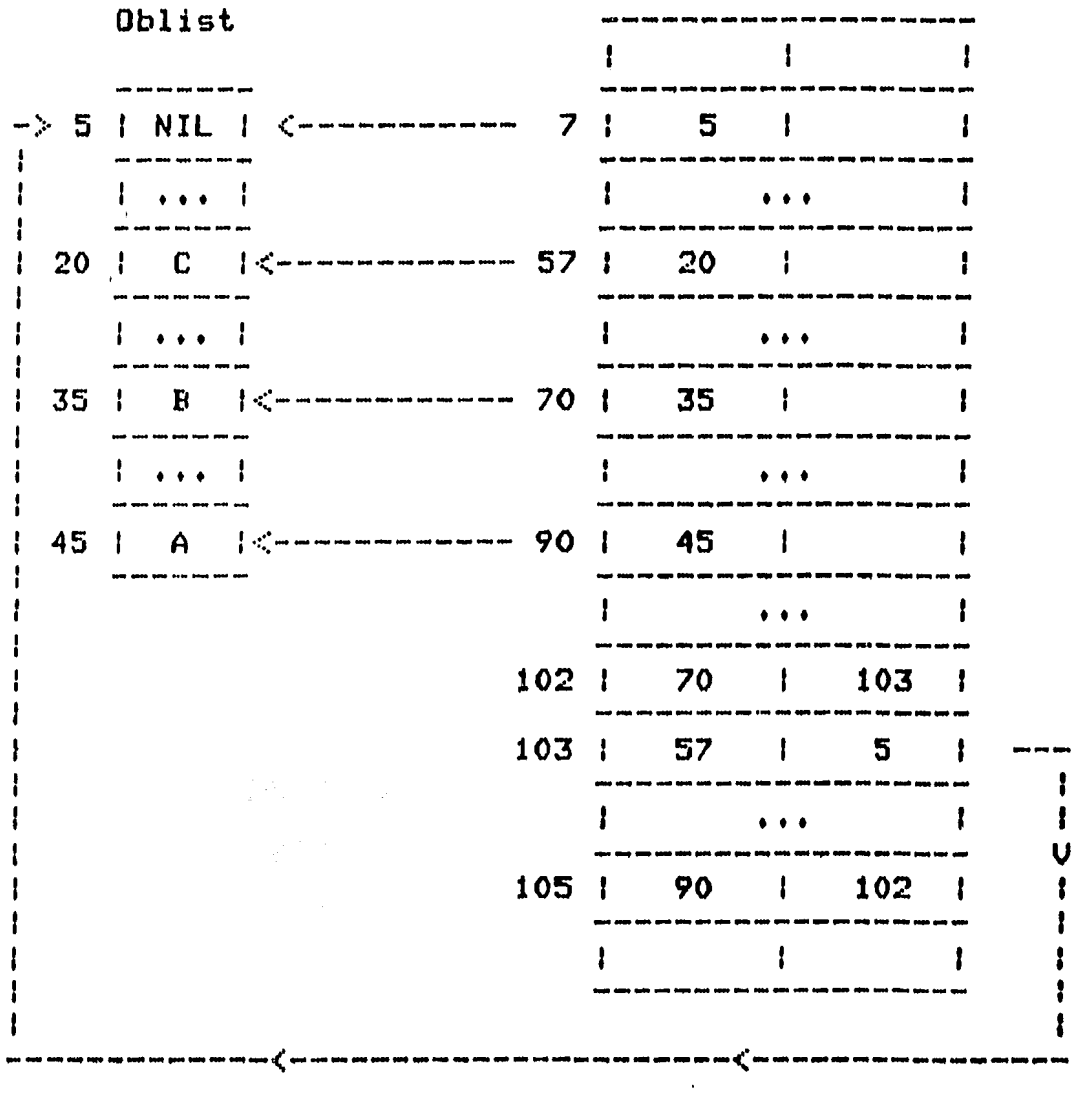


En nuestro ejemplo anterior de la representación de la expresión simbólica (A B C), tenemos que las "cajas-dobles" son las localidades que pertenecen al área de trabajo y las "cajas-sencillas", con los nombres de los átomos A, B, C y NIL, corresponden al área de la lista de objetos (OBLIST).

Si representamos cada "caja" mediante una dirección X, tenemos que la representación de la expresión (A B C) es:

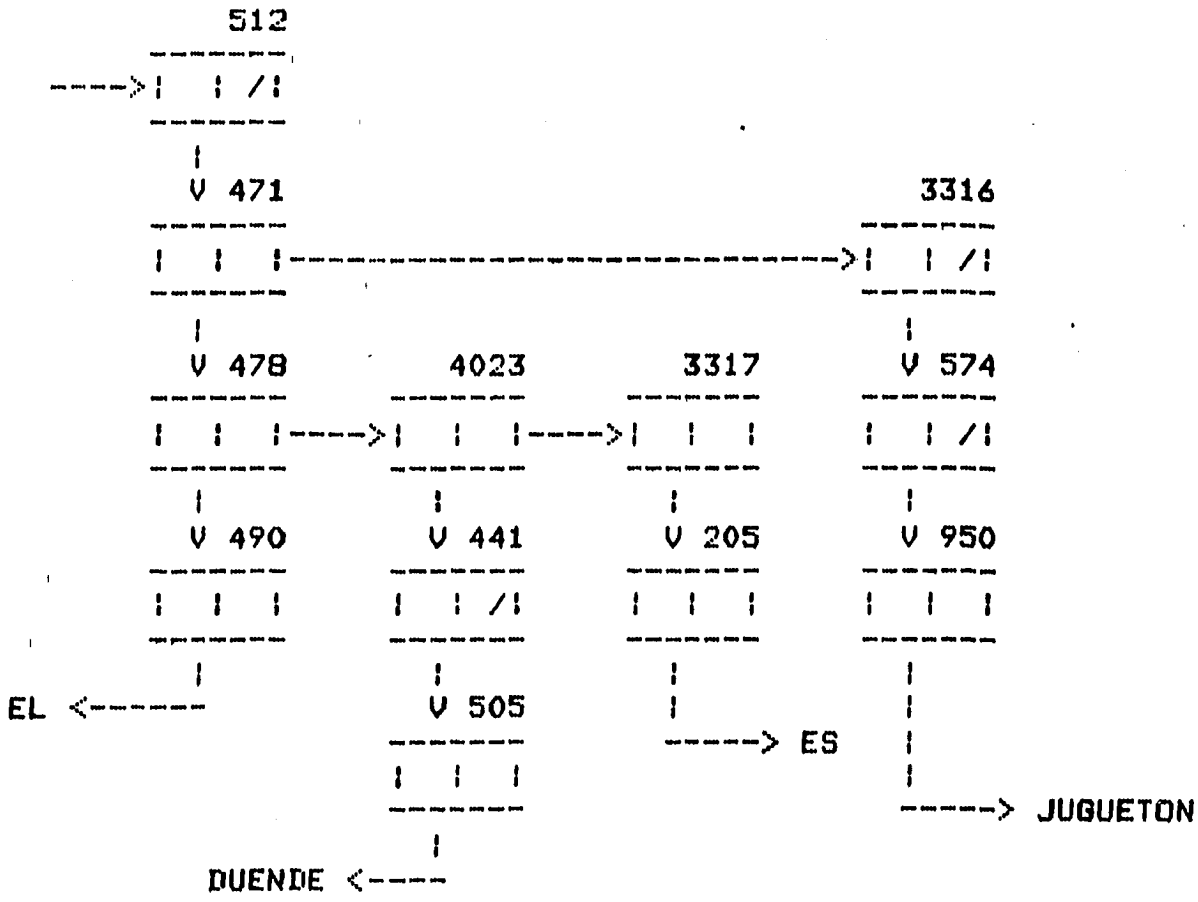


Area de Apuntadores



Para ilustrar mejor la estructura de la memoria, analicemos una estructura de lista, recorriendola a partir de diferentes localidades de su representación interna. De esta forma, encontraremos distintas expresiones simbólicas, dependiendo de la localidad de donde empecemos a recorrer la representación de la expresión dada inicialmente.

Sea la expresión: (((EL (DUENDE) ES) (JUGUETON)))



Lista encabezada por la localidad	Expresión simbólica
512	(((EL (DUENDE) ES) (JUGUETON)))
471	((EL (DUENDE) ES) (JUGUETON))
478	(EL (DUENDE) ES)
3316	((JUGUETON))
4023	((DUENDE) ES)
3317	(ES)
574	(JUGUETON)
441	(DUENDE)
490	EL
505	DUENDE
205	ES
950	JUGUETON

Como podemos ver, los números asociados a las localidades, que son las direcciones de éstas en la memoria, se han escogido arbitrariamente para mostrar que el almacenamiento de los elementos de la estructura de lista no es secuencial.

Las razones por las que no se tiene una organización secuencial son:

1) Algunas listas tienen muchas ramificaciones.

2) Las listas tienen una longitud variable e impredecible, a diferencia de los números, vectores y matrices.

3) El orden en el borrado de los elementos de las estructuras de lista y en la identificación de las localidades ocupadas por éstos para su recuperación, es totalmente aleatorio.

Podemos decir también, que el tipo de las localidades en el Area de Trabajo varía, según la etapa de ejecución del intérprete. Así tenemos:

1) Al inicio de la ejecución del intérprete.

   Todas las localidades están disponibles.

2) Durante la ejecución y antes de agotarse la lista disponible.

   Existen localidades disponibles, localidades activas y localidades basura.

3) Al detenerse la ejecución por agotarse la lista disponible.

   Sólo existen localidades activas y localidades basura.

4) Al continuar la ejecución, inmediatamente después de terminarse de efectuar la Recolección de Basura.

   Existen localidades activas y localidades disponibles.

4.3. AREA PARA VALORES DE LOS PARAMETROS.

Al evaluarse una función, las direcciones de las localidades que se usan en la representación del valor de sus argumentos formales se guardan en esta área que está organizada como una pila.

Así, por ejemplo, la parte de evaluación del intérprete hace lo siguiente al evaluar una lista:

- Evalúa el CAR de la lista recursivamente.
- Si el CAR de la lista da una expresión LAMBDA (ver apéndice) los argumentos de la llamada de la función se evalúan uno a la vez.
- Guarda los valores anteriores de los argumentos formales.
- Asigna nuevos valores a los argumentos formales.
- Evalúa el cuerpo de la función LAMBDA.
- Recupera el valor anterior de los argumentos formales.

Estas reglas garantizan el buen manejo de los valores de los argumentos.

4.4. AREA PARA ELEMENTOS AUXILIARES Y EN CONSTRUCCION.

Es un área auxiliar (pila), que se utiliza en las etapas de lectura y de evaluación del intérprete, en donde se almacenan las direcciones de las localidades que se están utilizando en la construcción de la representación interna de la expresión leída o del valor de la expresión.

Por ejemplo, supongamos que se está leyendo la expresión (CAR (QUOTE (A B C))), (ver apéndice) de la que el intérprete construye una representación interna. Las direcciones de las localidades que se utilizan en esta representación se almacenan en esta área auxiliar con el fin de conservarlas siempre como activas para el momento en que se agote el área disponible. Si no se hiciera de esta forma, al suceder una interrupción del intérprete por falta de espacio disponible y accionar el Recolector de Basura para recuperar área, las localidades que se están usando en las representaciones en proceso de creación se recuperarían, por lo que al volver a ejecutarse el intérprete no se podría completar la representación que había quedado pendiente, sino que habría que volver a leer la expresión para efectuar su representación.

En la evaluación, esta área también sirve para almacenar las direcciones de las localidades de la representación del valor de la expresión, con el mismo propósito de no perder las localidades que se están usando, cuando se agota la memoria disponible antes de terminar de realizar la representación.

En el caso de evaluar una expresión aritmética como (PLUS (TIMES 3 2) 5) se utiliza esta área auxiliar para guardar el valor temporal de (TIMES 3 2) que se saca después para evaluar PLUS.

5. MANEJO DE MEMORIA.

El diseño de un lenguaje de programación está fuertemente influido por las consideraciones del manejo de memoria.

Mientras que el diseño de cada lenguaje permite el uso de ciertas técnicas de manejo de memoria, los detalles de los mecanismos y su representación en hardware y software, son la tarea del diseñador.

El diseño del lenguaje LISP, permite el uso de las técnicas de la Lista Disponible y la Recolección de Basura para manejar la memoria del Area de Trabajo.

Sin embargo, existen diversos algoritmos que hacen posible la implantación de la técnica de Recolección de Basura, entre las cuales el diseñador podrá escoger alguno.

El manejo de memoria consta de cuatro aspectos básicos:

1) Inicialización. Al inicio de la ejecución del intérprete cada localidad del área de almacenamiento disponible se puede utilizar para algún uso o dejarla libre.

2) Asignación. Si inicialmente la localidad está libre, entonces está disponible para que el intérprete la asigne dinámicamente a una nueva estructura, durante su ejecución.

3) Recuperación.- El área que ha sido utilizada por una estructura que se convierte en inaccesible, se debe recuperar por el administrador de la memoria para su reuso. La recuperación puede ser muy simple, como en la reposición de un apuntador a una pila, o muy compleja, como en la Recolección de Basura.

4) Recolección.- El área identificada para ser recuperada, se junta en una lista para volverla a utilizar.

5.1. INICIALIZACION.

En esta etapa, se representan en las primeras k-1 localidades del Area de Apuntadores, que es el Area del Lenguaje, los átomos simbólicos significativos para LISP, y se almacenan en la lista de objetos (OBLIST) los nombres de dichos átomos.

Las N-k+1 localidades restantes del Area de Apuntadores, que se encuentran disponibles, y que forman el Area de Trabajo, se unen secuencialmente, mediante el apuntador derecho CDR de cada localidad, formando así lo que se denomina "Lista Disponible" ("lista libre").

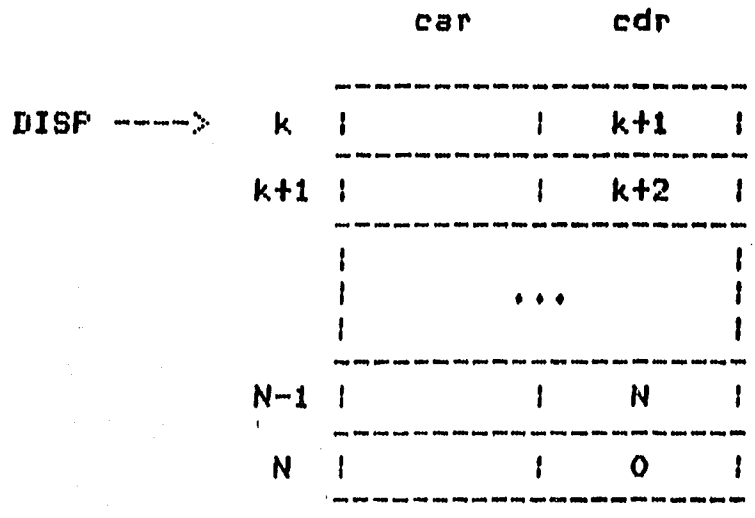
5.1.1. LISTA DISPONIBLE.

La lista disponible se conoce en la literatura como Lista Libre ([McC 60], [New 60], [Wai 64]).

Este concepto de lista libre fué introducido por Newell y Shaw [New 60], y desde entonces ha sido incorporado en un gran número de lenguajes de procesamiento de listas ([Wei 63], [McC 60], [Gel 60], [Wil 63]). De esta lista, que es el Area de Trabajo disponible, el intérprete va a ir usando las localidades para formar las nuevas estructuras de lista durante su ejecución.

Para que el programa tenga acceso a las localidades disponibles, existe un indicador contenido en una celda llamada DISP, que es un apuntador a la cabeza de la Lista Disponible, el cual apunta a la primera localidad disponible de la lista.

Lista Disponible = Area de Trabajo Disponible



Cuando el intérprete necesita una localidad para formar algún nuevo elemento de una estructura de lista ligada nueva, toma la primera localidad disponible, apuntada por DISP, y cambia el valor del apuntador al número de la siguiente localidad disponible.

5.2. ASIGNACION.

Veamos como se efectúa la asignación de las localidades de memoria, en la representación interna de las expresiones simbólicas, durante la ejecución del intérprete.

LECTURA.

Al leer la expresión simbólica, el intérprete construye la representación interna correspondiente.

Existe una rutina que acepta una expresión simbólica de lista, que construye toda la representación interna, usando las localidades de apuntadores del Area de Trabajo.

Existe otra rutina que aloja los nombres de los átomos en las localidades de la lista de objetos (OBLIST). Pero antes de hacer esto, la rutina primero examina si el nombre del átomo dado ya está presente en OBLIST, si no está, la rutina emplea una nueva localidad de esta lista para alojar el nombre del átomo, y si está ya no lo almacena.

El proceso de lectura lo que hace internamente en la memoria es:

1) Crear apuntadores, uno a la primera localidad de apuntadores de la representación interna de la expresión simbólica de lista, y otros para encadenar las localidades de apuntadores usadas en tal representación.

2) Crear apuntadores a las localidades de la lista de objetos, donde se alojan los nombres de los átomos simbólicos.

Siempre que la rutina de lectura encuentra un átomo, ésta regresa un apuntador a la localidad de apuntadores que conduce a la localidad en OBLIST, que tiene el nombre del átomo.

De esta forma durante la ejecución del intérprete de LISP, se van usando nuevas localidades de apuntadores en la lectura y

otras se dejan de usar cuando se borran elementos de las estructuras, o se utilizan nuevas durante la evaluación.

EVALUACION.

La evaluación, se realiza mediante la rutina "eval", que es la parte principal del intérprete.

Esta rutina se encarga de evaluar la expresión simbólica, cuyo resultado se representa internamente.

El argumento de "eval" es la representación interna de la expresión simbólica, y la evalúa de acuerdo a la convención del lenguaje LISP.

Si la expresión es una lista, su valor se determina al aplicar la primitiva o la función, especificada por el CAR de la lista, a la lista de argumentos, determinada por el CDR de la lista.

Si la expresión es un átomo simbólico, su valor será la representación interna asociada con el átomo.

Si la expresión es un átomo numérico, o bien los átomos simbólicos T y NIL, su valor será la misma expresión.

Al realizarse la evaluación de una expresión simbólica, lo que sucede internamente es lo siguiente:

En el caso de unas expresiones, algunos de los apuntadores creados en el proceso de la lectura se eliminan, mientras que otros se crean para que apunten a las localidades donde empieza la representación interna del valor de la expresión.

ESCRITURA.

Esta rutina toma como argumento un apuntador a una representación interna e imprime el valor como una expresión simbólica.

Ahora bien, tenemos que el proceso de eliminación de apuntadores, convierte en logicamente inaccesibles a algunas de las estructuras de la representación interna, pues ya no hay trayectorias de acceso (apuntadores) a esas localidades, que hagan posible que el intérprete las pueda accesar para su reutilización.

Por lo tanto en el momento en que la localidad queda inaccesible,

el área ocupada por ésta se podría recuperar para volverse a usar, pero al intentar recuperar este espacio de memoria, pueden presentarse algunos problemas, dependiendo del momento en que se decida hacer tal recuperación.

5.3 PROBLEMAS RELACIONADOS.

Existen dos problemas clave, en el diseño del método que recupera el área ocupada por las estructuras de la representación interna, que están en proceso de volverse inaccesibles.

1) Referencias pendientes.

2) Basura.

El problema de la basura se presenta en LISP, más no el de las referencias pendientes.

5.3.1 REFERENCIAS PENDIENTES.

Las referencias pendientes surgen cuando se recupera el área ocupada por una estructura de datos, antes de que todas las trayectorias de acceso (apuntadores) hayan sido eliminadas.

En PL/I:

ALLOCATE ELEM SET(P)	Asigna un elemento del espacio libre y la variable P contiene el apuntador a este elemento.
Q = P	Copia el apuntador P en Q.
FREE P -> ELEM	Borra el elemento apuntado por P y libera el área para su reuso. El apuntador en Q no se elimina, por lo que queda una referencia pendiente.

En el contexto del manejo de memoria del área de apuntadores, una referencia pendiente sería un apuntador a una localidad que ha sido regresada a la lista libre (o un apuntador a una localidad que hubiera sido regresada y más tarde reutilizada para otro propósito).

Las referencias pendientes pueden llevar a un caos completo, porque modifican el área en uso, de forma aleatoria.

Por lo que, las referencias pendientes se deben de evitar mediante el uso de un proceso de recuperación que anule cada trayectoria de acceso, sin permitir que el almacenamiento ocupado por la estructura de datos se recupere, hasta que se elimine la última trayectoria.

5.3.2. BASURA.

Una estructura de datos se convierte en basura cuando se elimina su última trayectoria de acceso, sin que el área de memoria ocupada por ésta se haya recuperado.

En el contexto del manejo de memoria del área de apuntadores, una localidad basura es una que podría estar disponible para volverse a usar, pero que no está en la lista de área disponible y por lo tanto está inaccesible para el intérprete.

La dificultad con las estructuras basura es la recuperación del almacenamiento que ocupan.

Si la basura se acumula se reduce gradualmente el almacenamiento disponible, hasta que el programa es incapaz de continuar por falta de espacio libre reconocido.

5.3.2.1. EJEMPLOS.

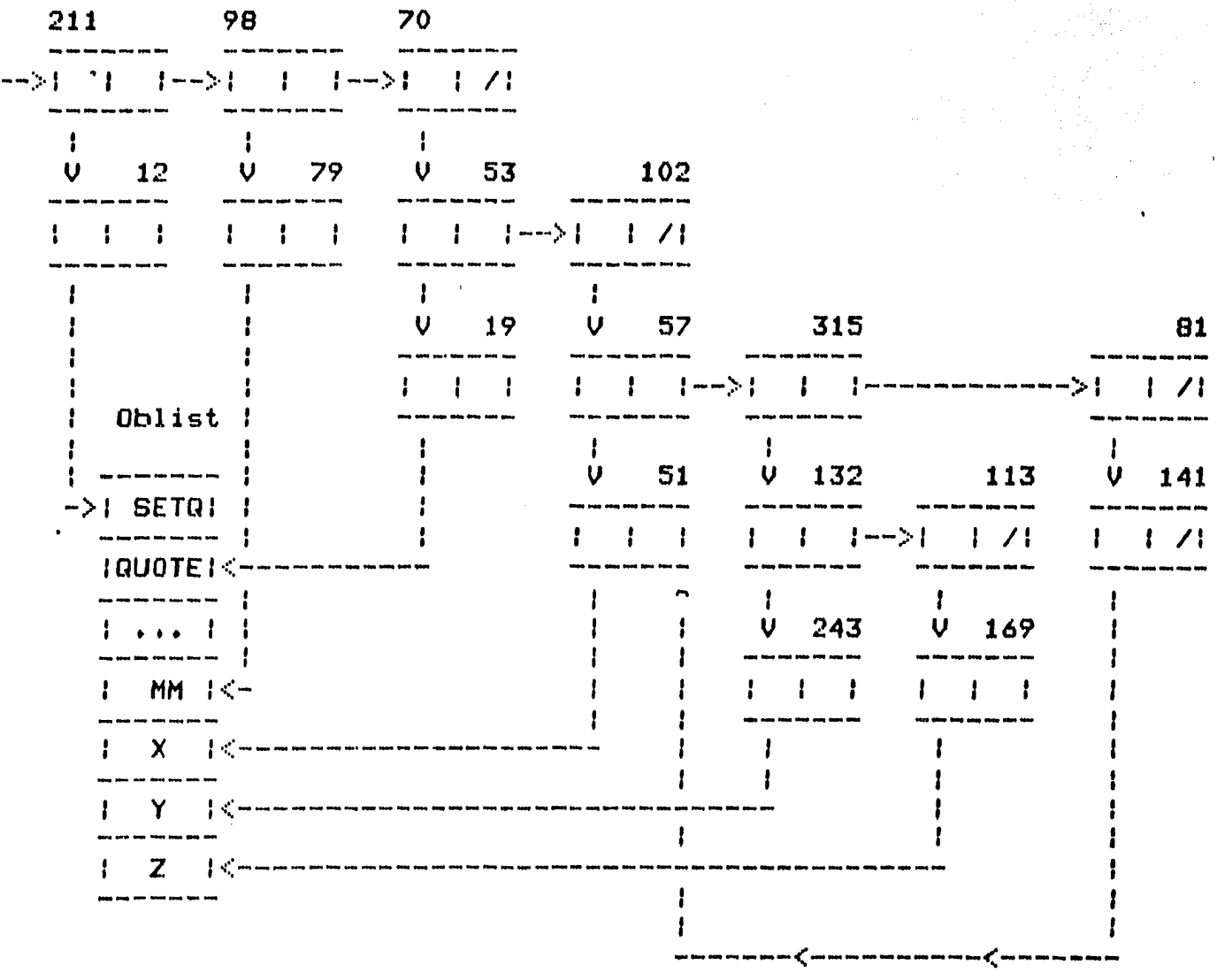
En PL/I:

ALLOCATE ELEM SET(P)	Asigna un elemento del espacio libre, la variable P contiene el apuntador a éste.
P = Q	Elimina el único apuntador al elemento, dejándolo como basura.

En LISP

Veamos la estructura de lista, que se crea cuando el intérprete lee la siguiente expresión simbólica (ver apéndice) y como al evaluarla algunas localidades ocupadas por la estructura se convierten en basura.

(SETQ MM (QUOTE (X (Y Z) (X))))



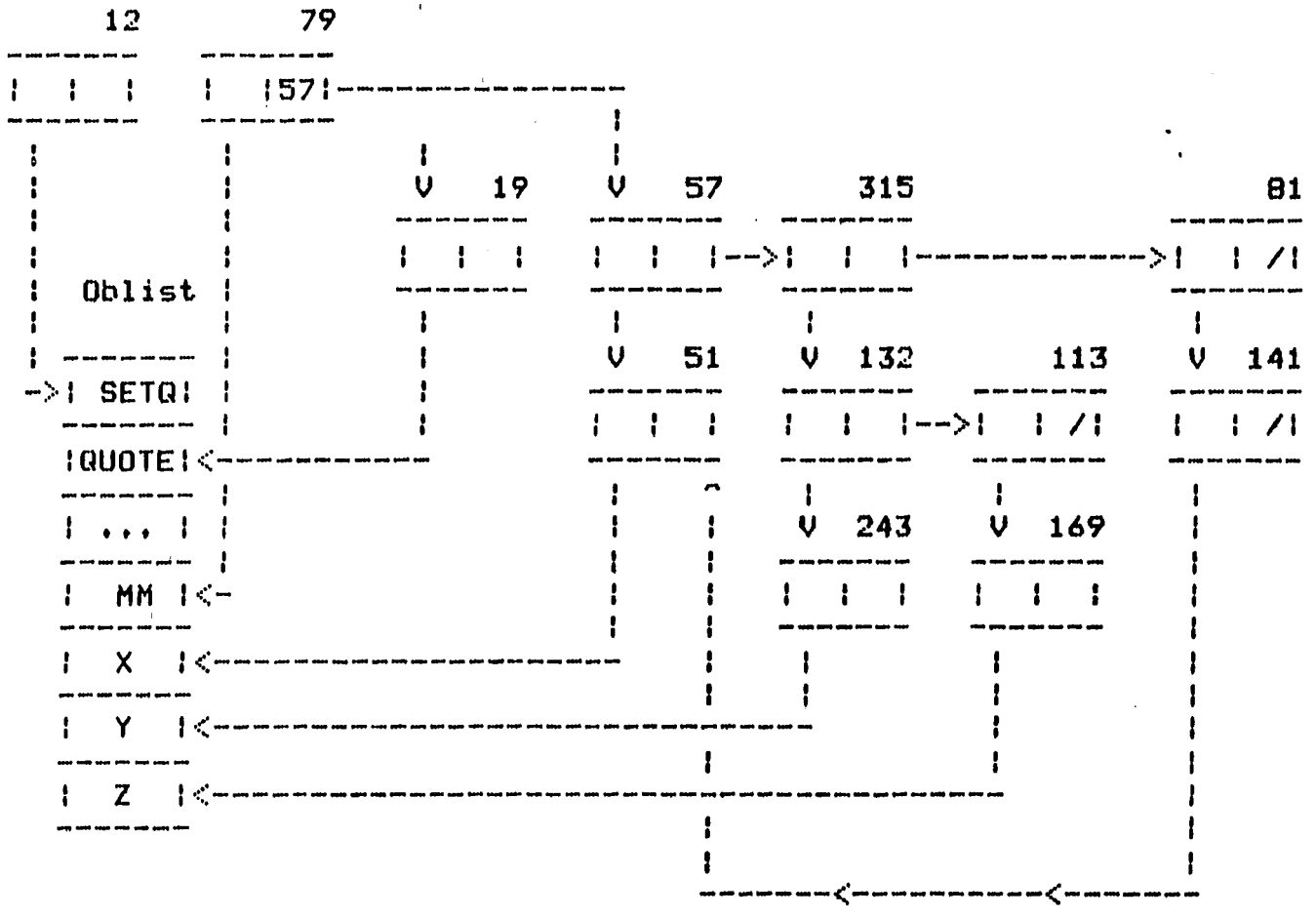
Al evaluar el intérprete esta expresión simbólica, se crea un apuntador a la localidad donde se inicia la representación interna del valor de tal expresión.

Además del resultado de la evaluación, se pueden producir otros efectos, como por ejemplo, la asociación de alguna expresión.

La localidad, cuya dirección es 79, que contiene el apuntador al átomo MM, alojará al nuevo apuntador, que apunta a la localidad de apuntadores número 57 donde empieza la representación interna del valor de MM (tal representación es el valor asociado a MM).

Para este ejemplo, las localidades: 211, 98, 70, 53 y 102 ya no son accesibles desde el valor de MM. Por lo que se convierten en "basura" y el intérprete ya no tiene acceso a ellas. Las localidades 12 y 19 no son basura porque representan funciones de LISP.

Así tenemos que la representación interna del valor de MM es la siguiente:



En la impresión, la rutina de escritura toma como argumento la representación interna del valor de MM. La accesa a través del apuntador CDR 57, que está guardado en la parte derecha de la localidad 79.

De esta manera, el intérprete imprime el valor de MM, en forma de expresión simbólica.

(X (Y Z) (X))

5.4. RECUPERACION DE BASURA.

El término recuperación significa, regresar las localidades ocupadas por las estructuras basura a la lista disponible.

Este regreso puede ser simple si dicha área está identificada. Pero la identificación y por la tanto la recuperación pueden ser difíciles.

El problema reside en determinar qué elementos de las estructuras de datos son basura y poder regresar el área que ocupan al área disponible.

Este problema ha recibido considerable atención en la literatura ([New 60], [McC 60], [Gel 60], [Col 60], [Wei 63], [Wil 64]), y a lo largo del tiempo se han propuesto las tres soluciones que se presentan a continuación, siendo la técnica de la Recolección de Basura la que trataremos más ampliamente por ser el objeto de estudio de este trabajo.

5.4.1. SOLUCIONES.

Existen tres métodos para mantener la lista de espacio disponible:

1) Regreso explícito por el programador o el sistema.

   Esta forma fué usada por Newell y Shaw [New 60]. El lenguaje IPL V incluye instrucciones que ocasionan que las listas y las estructuras de lista sean borradas, de esta manera regresan sus localidades a la lista libre.

   Cuando un elemento que ha estado en uso, se convierte en basura, se identifica explícitamente como "libre" y se regresa a la lista de espacio libre.

   En otros lenguajes, como PL/I y PASCAL, existen las instrucciones FREE y DISPOSE, respectivamente, para liberar explícitamente las estructuras de datos.

   El regreso explícito parecería la técnica natural de recuperación para el manejo de memoria del Area de Trabajo, pero esto no es posible, ya que, como el programador tiene que proteger el estado de las listas, de las sublistas, etc., puede suceder que libere parte de una lista que sea compartida por otras listas, y que por lo tanto todavía no se deba recuperar.

2) Contador de Referencias.

Debida originalmente a Gelertner et al. [Gel 60], extendida por Collins [Col 60] y usada por Weizenbaum ([Wei 62] y [Wei 63]), para sistemas que usan sublistas compartidas.

Este método requiere de mantener un contador de referencias hechas a una lista y regresa las localidades ocupadas por ésta a la lista disponible, cuando la cuenta llega a cero.

En una estructura de lista simplemente ligada, es imposible localizar la cabeza de lista cuando se hace una referencia a alguna localidad a lo largo de la lista. Por lo que, la localidad inicial de la lista referenciada se debe tratar como una nueva lista y se debe crear un nuevo contador de referencia.

La proliferación de contadores de referencias, y la gran cantidad de mantenimiento involucrado, hace que este método sea extremadamente pesado.

En una lista doblemente ligada [Wei 63] siempre es posible localizar la cabeza, y por lo tanto no es necesario colocar un nuevo contador de referencia. Sin embargo, se debe encontrar la cabeza de la lista e incrementar el contador de referencia en 1. Esto puede prevenir el regresar parte de una lista a la lista disponible.

Este método no funciona en el caso de una lista circular (i.e., una en que la lista es una sublista de sí misma). En este caso el contador de referencia no se puede hacer cero, aún cuando la lista completa llegue a ser inaccesible [McB 63].

3) La Recolección de Basura.

Esta técnica fué propuesta por John McCarthy [McC 60], y por ser la técnica que vamos a estudiar con detalle, le dedicamos un capítulo aparte.

6. TECNICA DE RECOLECCION DE BASURA.

Recolección de Basura es un término que denota el proceso de reclamar, como disponibles, las localidades de memoria que son basura.

La filosofía básica de la técnica de Recolección de Basura es simplemente permitir la generación de basura a fin de evitar la creación de referencias pendientes.

6.1. ANTECEDENTES.

Al diseñar McCarthy el lenguaje LISP, consideró el problema de recuperar área para su reuso, y se le hizo poco funcional usar la recuperación explícita que se empleaba en el lenguaje IPL.

Existían dos alternativas en ese entonces:

1) Usar el contador de referencias.

   Este método requería de cierto espacio adicional en la palabra, para guardar el contador de referencias.

   La palabra de la IBM sólo contaba con 6 bits adicionales a los usados por los apuntadores, que se encontraban separados en dos partes de 3 bits, lo que hacía que los contadores de referencias no fueran posibles de usarse sin tener que hacer un cambio drástico en la forma en que estaban representadas las estructuras de lista. (Un esquema de manejo de listas usando contadores de referencias, fué usado después por Collins [Col 60] en una computadora CDC con palabras de 48 bits).

2) Usar la técnica de Recolección de Basura que el mismo McCarthy inventó:

   En éste método, no se conservan contadores de referencias, y sólo requiere de un campo nuevo de un bit en cada localidad, denominado "bit de marca".

   Las localidades de apuntadores no se regresan a la lista libre sino hasta que esta lista se ha consumido, esto es, cuando

esta lista ya no tiene más localidades disponibles para usarlas en nuevas estructuras de lista.

Entonces se inicia el procedimiento conocido como Recolección de Basura", que examina todas las localidades del área de memoria (Area de Trabajo). Tales localidades pueden estar accesibles (activas) o ser localidades basura (inactivas).

La localidad accesible (activa) es aquélla que el intérprete puede accesar a través de un conjunto inicial de apuntadores (direcciones de las localidades inmediatamente accesibles). En cambio, una localidad basura (inactiva) no se puede accesar a través de ese conjunto inicial de apuntadores.

El proceso de Recolección de Basura marca las localidades activas, empezando por aquéllas que son inmediatamente accesibles, y deja sin marcar las localidades basura. Luego recorre el área de memoria secuencialmente para regresar a la lista disponible las localidades basura y borra la marca de las localidades activas, para el caso de que la Recolección de Basura tenga que usarse otra vez.

Al usar e implantar un procedimiento de Recolección de Basura se tienen diversos problemas:

a) El problema básico es el de revisar todas las estructuras de lista. Es problema, porque, en general, las estructuras tienen muchas ramificaciones y todas éstas se deben de rastrear.

   Se han sugerido varios algoritmos que las revisan, pero algunos requieren de una cantidad de almacenamiento adicional para almacenar las localidades donde existe una bifurcación y otros el de volver a examinar muchas veces las grandes porciones de la estructura.

b) Un segundo problema se presentaba cuando los datos consistían de números con signo que se almacenaban en una palabra completa.

   Este problema se originó en la forma en que inicialmente McCarthy decidió implantar la técnica de Recolección de Basura.

   La implantación consistía en marcar cada localidad activa con un signo menos, mientras que las localidades basura permanecían con signo positivo.

   El signo de cada localidad activa se hacía positivo después de haberse juntado las localidades basura (las que estaban sin marcar), en la lista de almacenamiento disponible.

El problema era que si este procedimiento no era modificado, cambiaría el signo de cualesquier número negativo.

La modificación efectuada fué la de utilizar un bit de la palabra, llamado bit de marca, que inicialmente está apagado en todas las localidades de apuntadores y que sólo se prende para marcar cada localidad activa, quedando apagado en las localidades basura.

6.2, ETAPAS.

El procedimiento de la Recolección de Basura consiste de dos etapas:

1) Marca - Marca todas las localidades que están activas, esto es, identifica las localidades basura, dejándolas sin marcar.

2) Recolecta - Incorpora las localidades basura (las no marcadas) en la lista disponible, para que el intérprete las pueda accesar.

Borra la marca de cada localidad activa, para un subsecuente uso de la Recolección de Basura.

La técnica de Recolección de Basura marca cada localidad activa, empezando por las localidades inmediatamente accesibles.

6.3. LOCALIDADES INMEDIATAMENTE ACCESIBLES,

En el momento en que se inicia la Recolección de Basura, existe un ambiente activo (accesible). Este incluye a las pilas que se están usando y las localidades que representan a los átomos simbólicos.

El ambiente especifica el conjunto de localidades inmediatamente accesibles, las cuales son las cabezas de las listas activas, a partir de las cuales se empieza el rastreo y por lo tanto definen todas las estructuras activas.

De esta forma, las localidades de apuntadores inmediatamente accesibles, aparte de las que representan a los átomos simbólicos, son aquéllas cuyas direcciones están almacenadas en las siguientes áreas de memoria.

1) Area de Trabajo.- En los CDR's de las localidades de apuntadores que representan a los átomos simbólicos están las direcciones de las primeras localidades de la representación de los valores asociados a estos átomos.

2) Area para valores de los parámetros.- Están las direcciones de las primeras localidades de apuntadores que se usan para representar los valores de los argumentos.

3) Area para elementos auxiliares y en construcción.- Están las direcciones de las localidades que se están usando en las representaciones en construcción (en proceso de creación en la lectura), o las de las localidades utilizadas en la representación del valor de una expresión, como resultado de la evaluación.

Las localidades que no se pueden accesar mediante los apuntadores de las localidades activas, son las localidades basura.

6.4. FUNCIONAMIENTO.

La técnica de Recolección de Basura empieza a ejecutarse en el momento en que se agota la lista disponible, instante en que se detiene la ejecución del intérprete, lo cual puede ocurrir en cualquier momento, por ejemplo, cuando el programa se encuentre ejecutando una función recursiva.

Al iniciarse la Recolección de Basura, solo existen localidades activas y localidades basura en el Area de Trabajo.

El procedimiento de Recolección de Basura marca cada localidad activa, empezando por marcar las localidades inmediatamente accesibles.

Al finalizar de marcar todas las localidades activas, el procedimiento de Recolección de Basura recorre el área de memoria secuencialmente para unir las localidades basura que formarán la nueva lista disponible, y apaga el bit de marca de las localidades activas, para la siguiente vez que se ejecute este método.

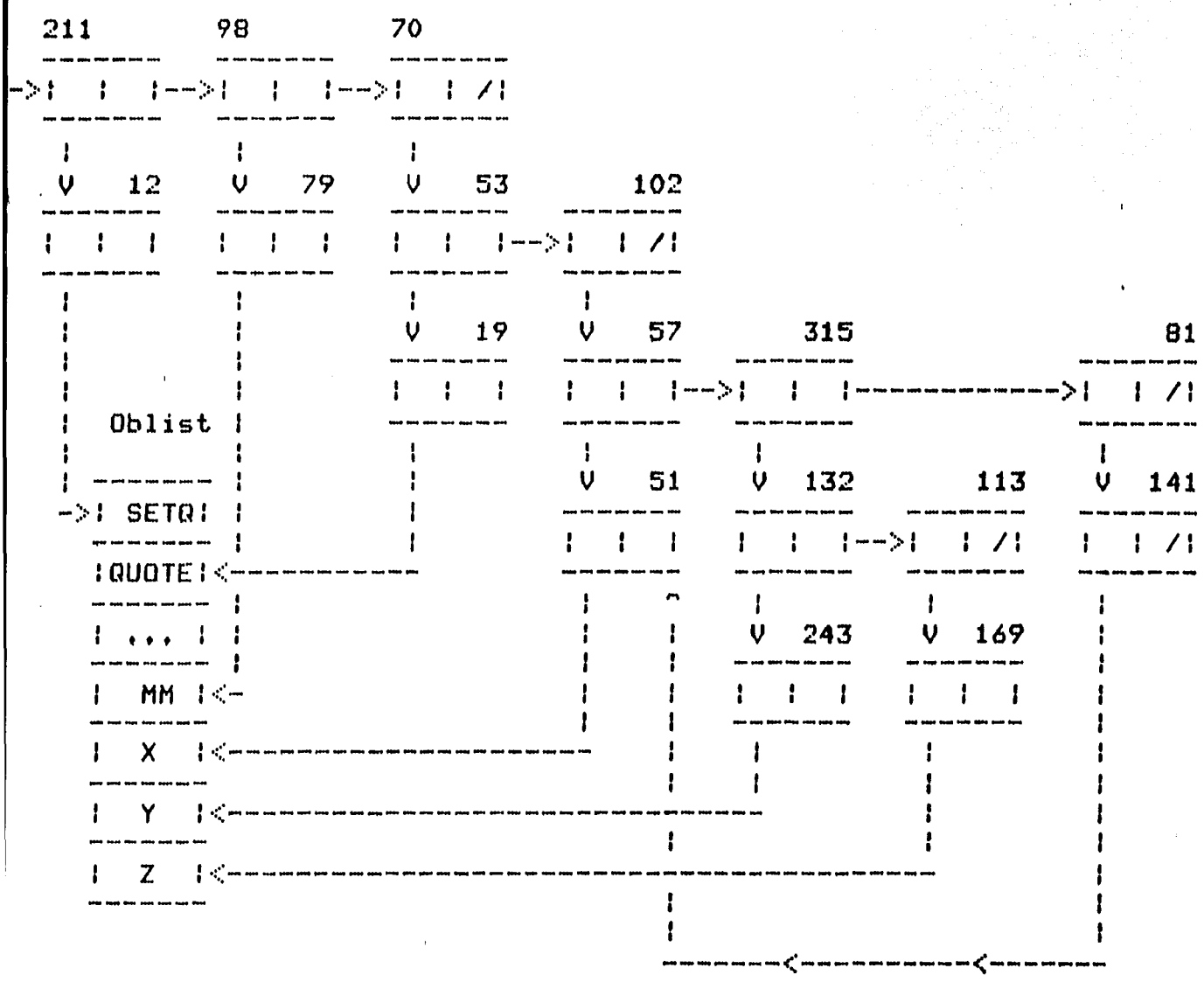
Con esto termina de ejecutarse la Recolección de Basura, momento en el que se reanuda la ejecución del intérprete.

Así tenemos que durante la ejecución del intérprete, es posible crear localidades de apuntadores que son basura y el trabajo de la Recolección de Basura es encontrar todas estas localidades para hacerlas nuevamente disponibles.

Por ejemplo, si LISP evalúa la expresión simbólica que examinamos anteriormente en la sección referente a basura:

(SETQ MM (QUOTE (X (Y Z) (X)))) (1)

Tenemos que su representación interna es:



El resultado de la evaluación será la siguiente expresión, que corresponde al valor de MM:

(X (Y Z) (X)) (2)

Suponsamos ahora lo siguiente:

- El intérprete va a leer la expresión simbólica (CONS 'A '(B)), (ver apéndice) para la cual necesita tomar localidades disponibles para construir su representación interna.

- La lista disponible en este instante, ya se ha agotado.

Dado esto surge la necesidad de recuperar área. Entonces empieza a funcionar la técnica de Recolección de Basura para recuperar el área de las localidades basura.

La etapa que marca recorre la memoria para marcar las localidades activas: 12, 79, 19, 57, 315, 81, 51, 132, 113, 141, 243 y 169 del Area de Trabajo.

Después de que se han marcado todas las localidades de apuntadores activas, la etapa que recolecta, recorre toda el Area de Trabajo secuencialmente, para unir todas las localidades basura (que no se marcaron), mediante el apuntador CDR, para formar la nueva lista disponible y borra la marca de las localidades activas que se marcaron (para la siguiente vez que funcione la Recolección de Basura).

En este ejemplo, vimos que la Recolección de Basura empezó a funcionar en la lectura de la segunda expresión (CONS 'A '(B)), al querer el intérprete tomar localidades de la lista disponible, pero bien pudo haber funcionado al momento de construirla en la lectura, o al momento de evaluarla, si la lista disponible se hubiera agotado después.

En general, el problema de recuperar área mediante la Recolección de Basura puede surgir en cualquier momento, durante la ejecución del intérprete.

6.5. ALGORITMOS.

El problema de la recuperación de área que es basura, es la identificación de la misma, lo cual se lleva a cabo mediante la etapa de marca de la Recolección de Basura.

De tal etapa estudiaremos los diversos algoritmos que describe Knuth [Knu 68].

El propósito de estos algoritmos consiste en marcar todas las localidades activas del Area de Trabajo, empezando por aquellas que son inmediatamente accesibles.

Las consideraciones comunes a los algoritmos son:

1) Sea M = N - k + 1, el número de localidades de apuntadores del Area de Trabajo.

2) Sean LOC(k), LOC(k+1), ..., LOC(N) las M localidades del Area de Trabajo.

3) Cada localidad k está constituída por dos apuntadores y por cuatro indicadores:

CAR(k) - Apuntador izquierdo.

CDR(k) - Apuntador derecho.

ATOMS(k)
- 0 - Indica que apunta a otro elemento de la lista.
- 1 - Señala que apunta a un átomo simbólico.

ATOMN(k)
- 0 - Contiene apuntador CDR.
- 1 - Contiene átomo numérico en lugar del CDR.

MARCA(k)
- 0 - No está marcada la LOC(k), para la Recolección de Basura.
- 1 - Si está marcada la LOC(k).

El siguiente bit sirve para el proceso de inversión de apuntadores, se prende cuando se invierte el apuntador CAR y se deja apagado cuando se invierte el apuntador CDR. De esta manera en la parte de restaurar los valores de los apuntadores para mantener la lista original se tiene que:

RESTB(k)

- 0 - Indica que se recorre la lista siguiendo el apuntador derecho, para reestablecer el valor de este apuntador.
- 1 - Señala que se recorre la lista a través del apuntador izquierdo, para reestablecer el valor de este apuntador.

Los algoritmos marcan todas las localidades de apuntadores que se pueden accesar mediante las trayectorias que parten del CAR y del CDR de las localidades marcadas (activas) y que no son átomos.

Los algoritmos 1, 2 y 4 marcan primero todas las localidades inmediatamente accesibles.

En los algoritmos 3, 5 y 6, tenemos que para cada localidad inmediatamente accesible (cabeza de lista) se recorre el resto de la lista para marcar las demás localidades activas.

Las localidades se marcan una sola vez (se llega a localidades marcadas porque las localidades de las listas se comparten).

Daremos una breve descripción del funcionamiento de los algoritmos al inicio de cada uno de ellos.

ALGORITMO 1.

Este algoritmo hace lo siguiente:

1) Marca todas las localidades inmediatamente accesibles.

2) Examina todas las localidades del área de trabajo empezando por la primera localidad, cuya dirección se guarda en la variable J.

   En la variable J1 se guarda el valor de la dirección de la siguiente localidad en secuencia a J (J+1).

   Se examina la localidad hijo izquierdo (derecho) de la

localidad J que no sea átomo y esté marcada, y se marca.

En la variable J1 se guarda la menor de las direcciones de las siguientes localidades:

- La siguiente en secuencia a J, (J+1).

- La que es hijo izquierdo (derecho) de la localidad J.

Una vez que se tiene en la variable J1 el valor de la dirección mínima, se asigna este valor a la variable J, y se continua el rastreo a partir de esta dirección.

1. [Inicializa].

```
Marca todas las localidades inmediatamente accesibles.

Sea  J = k;        /* k es la primera localidad */
                   /*   del  área  de  trabajo  */
```

2. [Examina las M localidades del área de trabajo].

```
while   ( J menor o igual que M )
    begin
        J1 = J + 1;          /* siguiente localidad */

        if   (LOC(J) no es atomo y LOC(J) está marcada)
            begin

                /* Examina hijo izquierdo */
                i = CAR(J);
                [Examina LOC(i)];

                /* Examina  hijo  derecho */
                i = CDR(J);
                [Examina LOC(i);

            end;

        J = J1;
    end;
```

3. [Examina LOC(i)].

```
if   ( (i apunta al área de trabajo)   y
       (LOC(i)  no está marcada)  )
```

```
begin

    Marca LOC(i) como localidad activa;
    if   (LOC(i) no es átomo)
         J1 = mínimo apuntador entre J1 y i;

end;
```

ALGORITMO 2.

Este algoritmo es similar al 1 con una modificación.

1) Se marcan todos los nodos inmediatamente accesibles.

2) Hace un recorrido secuencial del Area de Trabajo, empezando por la primera localidad, cuya dirección se guarda en la variable J.

   Utiliza el mecanismo del algoritmo 1, guardando en la variable J1 la menor dirección de las siguientes localidades:

   - La siguiente en secuencia a J (J+1).

   - La que es hijo izquierdo (derecho) de la localidad J.

   Se examinan las localidades del Area de Trabajo y se marcan las que son hijo izquierdo y derecho de las localidades que no son átomos y están marcadas. La modificación consiste en que durante este proceso se guarda en J1 la menor de las direcciones de las localidades examinadas. Al terminar de recorrer toda el área de trabajo, se vuelve a recorrer ésta a partir de la localidad que corresponde a la dirección mínima almacenada en J1, siempre que su valor sea menor a la dirección de la última localidad de apuntadores.

1. [Inicializa].

   Marca todas las localidades inmediatamente accesibles.

```
Sea  J = k;        /* k es la primera localidad */
                   /*   del  área  de  trabajo  */
  y  J1 = J;
```

2. [Examina las M localidades del área de trabajo].

```
while   ( J1 menor o igual que M )
    begin
        J1 = M + 1;

            while (J menor o igual que M)
                begin

                    if  ( (LOC(J) es átomo)  o
                          (LOC(J) no está marcada)
                       begin
                           J = J + 1;
                           continua con el segundo while;
                       end;

                       /* Examina hijo izquierdo */
                    i = CAR(J);
                    [Examina LOC(i)];

                       /* Examina  hijo  derecho */
                    i = CDR(J);
                    [Examina LOC(i)];

                    J = J + 1;
                end;
         J = J1;
     end;
```

3. [Examina LOC(i)].

```
if   ( (i apunta al área de trabajo)   y
       (LOC(i)  no está marcada)  )
    begin

        Marca LOC(i) como localidad activa;
        if   (LOC(i) no es átomo)
            J1 = mínimo apuntador entre J1 y i;

    end;
```

ALGORITMO 3.

Para cada localidad inmediatamente accesible (cabeza de lista), que se marca, se recorre el resto de la lista para marcar las demás localidades activas.

Este algoritmo utiliza una pila, en el que se almacenan las direcciones de las localidades marcadas que no son átomos, recorriendo la lista de la siguiente forma:

Primero visita la localidad cabeza de lista, cuya dirección se guarda en la pila, después se saca de la pila, y se visita la localidad hijo izquierdo, la cual se marca si no estaba marcada, y se guarda en la pila, si ya estaba marcada ya no se guarda en la pila, lo mismo sucede con el hijo derecho.

Después se saca la dirección del hijo derecho, que se convierte en cabeza de lista, y se repite este proceso hasta que la pila queda vacía, en cuyo caso se termina el algoritmo.

1. [Inicializa].

Sea la siguiente pila con T localidades:

PILA[0], ..., PILA[T-1]

Sea t el tope de la pila y
PILA[0] = p; donde, p es un apuntador a una localidad accesible.

2. [Se guardan las direcciones en la pila].

```
while   (pila no está vacía)
    begin

            /* Se saca un apuntador de la pila */

        J = PILA[t];
        t = t - 1;

        if  (LOC(J)  es  átomo)  continua con el while;

            /* Recorre rama izquierda */
        i = CAR(J);
        [Recorre rama];

            /* Recorre  rama  derecha */
        i = CDR(J);
        [Recorre rama];

    end;
```

3. [Recorre rama].

```
if   ( (i apunta al área de trabajo)   y
       (LOC(i)  no está marcada)  )
     begin

         Marca LOC(i) como localidad activa;

         /* guarda apuntador i en la pila */

         t = t + 1;
         PILA[t] = i;

     end;
```

ALGORITMO 4.

Este algoritmo combina los algoritmos 1 y 3, solo que en lugar de usarse una pila se utiliza una cola circular. En la variable J1 se almacena el valor de la última dirección más uno.

En este algoritmo se utiliza primero el mecanismo de almacenar las direcciones de las localidades inmediatamente accesibles, ya marcadas, en la cola circular.

El valor del lugar de la cola donde se almacena cada dirección, se calcula sacando el módulo tamaño de la cola del tope de la cola.

Cuando la cola se llena, se saca una dirección de esta cola (cuyo lugar se calcula sacando el módulo tamaño de la cola del fondo de la cola), y en su lugar se guarda la dirección de la localidad siguiente. La dirección que se sacó de la cola se compara con la que existe en la variable J1 y la que resulte menor se guarda otra vez en J1.

Después se examinan las localidades hijo izquierdo (derecho) de las localidades que no son átomos y están marcadas y se marcan.

El orden en como se guardan y se sacan las direcciones de las localidades en la cola es: primero se almacena la localidad hijo izquierdo y luego la localidad hijo derecho. A continuación se saca la localidad hijo derecho y se almacena la dirección del hijo izquierdo de esta localidad, y así sucesivamente.

Al terminarse este proceso, se presunta si existe un valor menor a la última dirección de la memoria (guardado en J1). Si este es el caso, se recorre el Area de Trabajo secuencialmente, a partir de la localidad con dirección mínima, y al encontrar una

localidad que no sea átomo y esté marcada se repite el proceso de la cola circular empezando con esta localidad. En caso contrario se termina el algoritmo.

1. [Inicializa].

Marca todas las localidades inmediatamente accesibles y las guarda en la cola circular.

Sea COLA[0], ..., COLA[T-1] la cola circular.

Sea t el tope de la cola, donde t = T - 1 y
b el fondo de la cola, donde b = T - 1

Guardar en la cola significa lo siguiente:

t = (t+1) mod T,
COLA[t] = P, donde, P es la dirección de una localidad accesible.

Si t es igual a b, entonces

b = (b+1) mod T, y J1 = min(J1, COLA[b])

donde J1 inicialmente tiene el valor de la dirección de la última localidad más uno.

2. [Saca una dirección de la cola].

```
/* mientras la cola circular no esté vacia */

while   ( t  diferente de b )

    begin
        J = COLA[t];
        t = (t+1) mod T;
        [Guarda las direcciones en la cola];
    end;

[Recorre el Area de TrabaJo];
```

3. [Guarda las direcciones en la cola].

```
if (LOC(J)  es  átomo)  regresa a 2.

      /* Recorre rama izquierda */
  i = CAR(J);
```

```
[Recorre rama];

    /* Recorre  rama  derecha */
i = CDR(J);
[Recorre rama];
```

Regresa a 2.

4. [Recorre rama].

```
if    ( (i apunta al área de trabajo)   y
        (LOC(i)  no está marcada)  )
    begin
        Marca LOC(i) como localidad activa;

        /* guarda apuntador i en la cola */

        t = (t+1) mod T;
        COLA[t] = i;
        if (t  igual  a  b)
          begin
             b = (b+1) mod T;
             j1 = mínima dirección entre J1 y i;
          end;
```

5. [Recorre Area de Trabajo].

```
while   (J1  menor o igual que M)

    begin
        if  (LOC(J1)  es átomo o no está marcada)
            begin
                J1 = J1 + 1;
                continua con el while;
            end;


         if  (LOC(J1)  está marcada)
            begin
                j = J1;
                J1 = J1 + 1;
                regresa a 2.
            end;
    end;
```

ALGORITMO 5.

Este algoritmo lo presentaron Herbert Schorr y W.M. Waite en 1965. El funcionamiento de este algoritmo consiste en invertir los apuntadores, para utilizar la misma estructura de lista como una pila.

Invertir un apuntador significa alterar el valor de éste para que apunte a su padre y no a su hijo.

Al alterar el apuntador izquierdo se marca la localidad prendiendo un bit. Este indicador sirve después en el proceso de restauración de los valores de los apuntadores.

Al encontrar una localidad que esté marcada de esta forma, se restaurará el valor de su apuntador izquierdo, de lo contrario se reestablecerá el valor de su apuntador derecho.

La forma de recorrer las lista para marcar las localidades activas e invertir los apuntadores es de la siguiente forma:

Se visita la cabeza de lista de la cual si no tiene alterado su apuntador izquierdo, se recorre la rama izquierda.

La localidad hijo izquierdo si no está marcada, se marca, y se invierte el apuntador izquierdo de su padre y marca esta localidad padre para indicar que se alteró su apuntador izquierdo. Ahora la localidad hijo izquierdo si no es átomo se convierte en cabeza de lista y se repite el proceso de recorrer la rama izquierda. Si esta localidad es átomo se procede a restaurar los apuntadores.

Si la localidad hijo izquierdo ya estaba marcada, se recorre la rama derecha.

La localidad hijo derecho si no está marcada, se marca, y se invierte el apuntador derecho de su padre (el bit que indica la alteración de apuntador queda apagado). En forma similar al proceso efectuado con la localidad hijo izquierdo, el hijo derecho si no es átomo y no está marcado de haber alterado su apuntador izquierdo, se convierte en cabeza de lista y se procede a recorrer la rama izquierda.

Si la localidad hijo derecho ya estaba marcada, se procede a restaurar los apuntadores. Al restaurar los apuntadores se considera la localidad cuyo apuntador se alteró, si esta localidad está marcada de que se alteró su apuntador izquierdo, se apaga el bit que lo indica, y se reestablece el valor de este apuntador para que apunte a su localidad hijo izquierdo.

Después se recorre la rama derecha de la localidad a la que se le restauró su apuntador izquierdo y se realiza lo que

describimos arriba con la localidad hijo derecho.

Si la localidad cuyo apuntador se alteró no está marcada de que se alteró su apuntador izquierdo, se reestablece el valor del apuntador derecho y sigue efectuando el proceso de reestablecer los valores de los apuntadores hasta que se llegue a la cabeza de lista inicial.

[Inicializacion].

Para cada localidad inmediatamente accesible que se marca, se ejecutan los siguientes pasos:

Sea t = 0, donde, t jugará el papel de tope de la pila artificial, que es la misma lista.

J = P /* P es la dirección de una localidad activa */

[Examina las localidades de apuntadores del área de trabajo].

```
while          ( (LOC(J) no es atomo)   y
        ( LOC(J) no está marcada para restaurar CAR(J) ) )
    begin

                /* recorre  rama  izquierda */

        i = CAR(J);

        if      ( (i apunta al área de trabajo)   y
                    (LOC(i) no está marcada) )
          begin
             Marca LOC(i) como localidad activa;
             Marca LOC(J) para indicar alteración de CAR(J);
             [Invierte apuntador izquierdo CAR(J)].
             continúa efectuando el while de este paso 2.
          end;

                /* recorre  rama  derecha */

        i = CDR(J);

        if      ( (i apunta al área de trabajo)   y
                    (LOC(i) no está marcada)
          begin
             Marca LOC(i) como localidad activa;
             [Invierte apuntador derecho CDR(J)].
             continúa efectuando el while de este paso 2.
          end;
```

```
            Se sale del while para restaurar apuntadores;

          end   while;
```

3. [Se restauran apuntadores].

```
      while       (T apunte a alguna localidad)
        begin

          i = t;

          if    (LOC(i) está marcada para restaurar CAR(i))
           . begin

              Se apaga el bit que indica que se restablece el CAR(i);
              [Reestablece el valor del apuntador CAR(i)];

                     /* recorre  rama  derecha  */
              i = CDR(J);
              if (i apunta al área de trabajo    y
                                  LOC(i)  no está marcada)

                begin
                   [Invierte el apuntador derecho CDR(i)].
                   [Examina las localidades de apuntadores].
                                (regresa  a  2)
                end;
              [Se reestablecen apuntadores]   (regresa a 3).
            end;

            [Se reestablece el valor del apuntador CDR(i)].

            Repite el paso 3.

          end;
```

4. [Invierte apuntador izquierdo CAR(J)].

```
      CAR(J) = t;
      t = J;
      J = i;
```

5. [Invierte apuntador derecho CDR(J)].

```
      CDR(J) = t;
      t = J;
      J = i;
```

6. [Reestablece el valor del apuntador CAR(i)].

```
t = CAR(i);
CAR(i) = J;
J = i;
```

7. [Reestablece el valor del apuntador CDR(i)].

```
t = CDR(i);
CDR(i) = J;
J = i;
```

ALGORITMO 6.

Este algoritmo se debe también a Schorr y Waite que lo crearon en 1967. Combinaron el algoritmo que utiliza una pila y el algoritmo que utiliza el proceso de inversión de apuntadores.

Cuando se llena la pila empieza a funcionar el algoritmo que invierte los apuntadores.

1. [Inicializacion].

```
Sea  r = p0;      donde  p0 es un apuntador a una
                         localidad accesible

     t = 0;       t es el tope de una pila implicita.
```

2. [Se guardan los apuntadores en la pila].

```
[Guarda apuntador CDR en la pila];
[Localidad siguiente apuntada por CAR];

if  (pila  esta  vacia) termina el algoritmo;

if  (pila no está vacia)
    begin

        r = pila[s];
```

```
                    s = s - 1;
                    continua ejecutando este paso;

                end;
```

3. [Guarda apuntador CDR en la pila].

```
    i = CDR(r);
    if  ( (i apunta al área de trabajo)   y
          (LOC(i)  no está marcada)  )
        begin

            Marca LOC(i) como localidad activa;
            if    ( (LOC(i)  no es átomo)
                begin

                    if (pila está llena) [Invierte apuntadores];

                    if  (pila no está llena)
                        s = s + 1;
                        pila[s] = j;
                end;
        end;
```

4. [Localidad siguiente apuntada por CAR].

```
    j = CAR(r);
    if  ( (j apunta al área de trabajo)   y
          (LOC(j)  no está marcada)  )
        begin

            Marca LOC(j) como localidad activa;
            if     ( (LOC(j)  no es átomo)
                begin

                    r = j;
                    continua ejecutando el paso 2;

                end;
        end;
```

5. [Invierte los apuntadores del área de trabajo].

```
    while   (verdadero)
        begin
```

```
        [Recorre rama izquierda];
        [Recorre rama derecha];

    end;
```

6. [Recorre rama izquierda];

```
    i = CAR(J);
    if  ( (i apunta al área de trabajo)    y
          (LOC(i)  no está marcada)  )
        begin

            Marca LOC(i) como localidad activa;
            if      ( (LOC(i)  no es átomo)
                begin

                    Marca LOC(J) para indicar restauración del CAR(J);
                    [Invierte apuntador izquierdo CAR(J)];
                    Continua ejecutando el paso 6;

                end;
        end;
```

7. [Recorre rama derecha].

```
    i = CDR(J);
    if  ( (i apunta al área de trabajo)   y
          (LOC(i)  no está marcada)  )
        begin

            Marca LOC(i) como localidad activa;
            if      ( (LOC(i)  no es átomo)
                begin

                    [Invierte apuntador derecho CDR(J)].
                    Continua ejecutando el paso 6;

                end;

            [Se restauran apuntadores];

        end;
```

8. [Se restauran apuntadores].

```
while   (t apunta a alguna localidad)
    begin

        i = t;

        if    (LOC(i) está marcada para restaurar CAR(i)
            begin
                Se apaga bit que indica restauración de CAR(i);
                [Restaura el valor del apuntador CAR(i)].
                [Recorre rama derecha]; /* para invertir CDR */

            end;

        [Se restaura el valor del apuntador CDR(i)];

    end;

[Localidad siguiente apuntada por CAR].

if  (Pila está vacía) termina el algoritmo;

if  (Pila esta llena)

     begin

         r = Pila[s];
         s = s - 1;
         continua ejecutando el paso 2;

     end;
```

Los siguientes procedimientos:

- [Invierte apuntador izquierdo CAR(J)]
- [Invierte apuntador derecho CDR(J)]
- [Restaura el valor del apuntador CAR(i)]
- [Restaura el valor del apuntador CDR(i)]

son los mismos que aparecen descritos en el algoritmo 5.

6.6. COMPARACION.

Los algoritmos que resuelven la etapa de identificación del área que es basura, que hemos descrito en este capítulo, nos muestran como se rastrea el Area de Trabajo para marcar las localidades activas y de esta manera dejar identificadas las localidades que son basura para su recuperación y uso posterior.

Los algoritmos los programé en lenguaje C y los probé en el intérprete que hizo el maestro Miguel Tomaseña y obtuve una comparación de ellos para ver cuál es más eficiente en cuanto a tiempo de ejecución y cuál en cuanto a espacio de almacenamiento.

Para ello, saqué de cada algoritmo el número de asignaciones, el número de comparaciones, el número de preguntas que se hacen por el estado de una localidad para saber si ésta ya está marcada y el número de palabras (variables) que emplea el algoritmo.

Para obtener estos datos ejecuté el intérprete y funcionó el Recolector de Basura de forma explícita (el usuario lo llama) y en forma implícita (el intérprete lo llama cuando se agota la lista disponible). (Ver ejemplo en el apéndice).

Ahora bien, con objeto de ejecutar cualquier programa en LISP, el primer paso en la ejecución del intérprete es leer una serie de funciones de LISP que están contenidas en el archivo UTILERIA.LIS, tecleando la expresión simbólica (EXECUTE "UTILERIA.LIS"). Después se leen los programas que se desean ejecutar, en nuestro caso se lee y ejecuta un programa que resuelve el problema de las 8 reinas que consiste en colocarlas en un tablero de ajedrez de manera que no queden en posición de ataque.

La llamada explícita al Recolector se hace antes y después de leer un archivo y de ejecutar el programa contenido en tal archivo, mientras que la llamada implícita la hace el intérprete cuando necesita localidades y ya está agotada la lista disponible.

Al hacer funcionar el Recolector se obtienen una serie de contadores que nos dan la información sobre cuantas localidades disponibles existen antes de hacer la recolección y cuantas se recuperaron después de hacerla (ver apéndice), así como los demás valores en lo que a operaciones se refiere.

Sabemos que el contenido de las localidades del Area de Trabajo varía según el momento de ejecución del intérprete, así tenemos que al inicio de la ejecución todas las localidades están disponibles, que durante la ejecución y antes de agotarse la lista disponible existen localidades disponibles, activas y basura, que al suspenderse la ejecución por agotarse la lista

disponible sólo existen localidades activas y localidades basura y que al continuarse la ejecución, inmediatamente después de terminarse de efectuar la Recolección, existen localidades activas y localidades disponibles.

El número total de localidades de apuntadores que se manejan en la memoria del intérprete es de 5000, de las cuales 45 corresponden a la representación de ciertas funciones básicas de LISP que sirven para construir las restantes, que están contenidas en el archivo UTILERIA.LIS, por lo que quedan 4955 localidades disponibles que constituyen el Area de Trabajo, de donde el intérprete toma las localidades para efectuar la representación interna de las expresiones simbólicas.

Así al inicio de la ejecución del intérprete y antes de hacer la llamada explícita al Recolector se tiene que todas las 4955 localidades del Area de Trabajo están disponibles y después de hacer la llamada mediante la expresión simbólica (RECOLECTOR T), se utilizan dos localidades de apuntadores para representar la lista (RECOLECTOR T) ya que los átomos T y RECOLECTOR ya están representados por ser átomos básicos de LISP. De esta forma después de ejecutarse el Recolector se tienen de las 4955 localidades, 4953 disponibles y 2 activas.

A continuación se leen las funciones de LISP que están en el archivo UTILERIA.LIS, y se crea su representación interna. Antes de ejecutarse el Recolector se tienen 4150 localidades disponibles, de las cuales, una vez que se ejecuta el Recolector se recuperan 117 localidades lo que sumadas a las disponibles dan un total de 4267 localidades disponibles.

En este momento de la ejecución del intérprete la tabla comparativa que se obtiene es la que aparece a continuación, mostrando los siguientes elementos; el número de las asignaciones que efectúa cada algoritmo, de las comparaciones booleanas, de las preguntas que se hacen para saber si las localidades están o no marcadas (preg-marc-loc), de las variables que se utilizan y el tiempo de ejecución en segundos.

Elem \ Alg	1	2	3	4	5	6
asignaciones	15184	46349	2595	8695	7613	4229
comp. booleanas	6612	39424	665	5573	1429	1022
preg-marc-loc	8340	46945	1929	9318	1905	2076
variables	6	4	34	38	7	5
tiempo en seg.	9	11	8	9	9	9

Si observamos otro momento de la ejecución del intérprete, cuando funciona el Recolector de forma automática en la ejecución del programa Reinas que coloca 5 de éstas, vemos que la tabla comparativa es la siguiente:

Elem \ Alg	1	2	3	4	5	6
asignaciones	20536	48818	3658	10780	17004	20511
comp. booleanas	8325	43866	916	5798	1952	1422
pres-marc-loc	11911	49731	2624	10772	2586	2856
variables	6	4	34	38	7	5
tiempo en seg.	10	11.3	8.3	9	9	9

Observando los valores de ambas tablas vemos que el algoritmo 2 es el que realiza un mayor número de asignaciones (todas las operaciones de asignación que efectúa el algoritmo), así como de comparaciones y de preguntas para saber si la localidad de apuntadores ya está marcada. Esto es porque el algoritmo revisa varias veces las localidades del Area de Trabajo, lo que hace que tarde más en ejecutarse, en comparación con los demás algoritmos que tienen otra forma de realizar la identificación del área activa para localizar las localidades basura.

En cambio el algoritmo 3 es el que realiza menor número de operaciones por lo que se lleva menos tiempo en ejecutarse.

Podemos deducir de la observación de ambas tablas que el tiempo que emplean en ejecutarse los algoritmos es proporcional al número de localidades que se liberan (localidades basura).

Dar un análisis preciso de estos algoritmos para su comparación es muy difícil al existir en su estructura una diferente secuencia de comparaciones booleanas que hacen que la ejecución de las siguientes operaciones dependan directamente del resultado de éstas comparaciones.

Mencionaremos cuales son las características principales de cada

uno de ellos, para establecer las diferencias principales.

En el algoritmo 1 y 2 lo que se hace es recorrer el Area de Trabajo para marcar todas las localidades activas, y se puede decir que en el peor caso, que sería cuando esta área contiene gran cantidad de localidades activas, el algoritmo tomaría un tiempo proporcional a pM, donde p es el número de localidades activas (factor de ocupación) y M el número de localidades del Area de Trabajo.

En el algoritmo 3, que utiliza una pila, tiene un tiempo de ejecución proporcional al número de localidades que marca (localidades activas).

Este algoritmo utiliza más área de memoria que el algoritmo 1 y 2, por el uso de la pila. Esto depende mucho de la implantación del Recolector , pues se puede definir esta área auxiliar como parte del Area de Trabajo, lo que hace que se requiera menos memoria. Esta manera de hacerlo se debe de haber hecho en las primeras implantaciones de LISP, ya que el área de memoria era muy reducida en las máquinas que se usaron para ello.

El algoritmo 4 es el algoritmo 1 cuando el valor del tamaño de la cola es de 1, y es el algoritmo 3 cuando este valor es mayor, del orden de 30.

Este algoritmo es más eficiente si el tamaño de la cola es grande, ya que se almacenarían más localidades inmediatamente accesibles, y no habría que recorrer de nuevo algunas localidades a partir de una que tenga una dirección mínima. En el caso de que se pudieran almacenar todas, solo se usaría el mecanismo del algoritmo 3 que usa una pila.

En el algoritmo 5 que utiliza el invertir los apuntadores de las localidades que no son átomos, de forma que apunten a sus padres y no a sus hijos, lo que hace que se utilice la misma estructura de lista como una pila. Al invertir los apuntadores, lo que se está haciendo es cambiar la estructura de ligado de la lista para conservar la forma de la pila.

Este algoritmo termina en la localidad cabeza de lista, de donde parte para examinar las localidades activas de la lista respectiva.

En este algoritmo también se requiere de un tiempo proporcional al número de nodos que se marcan.

El algoritmo 6 utiliza dos pilas, una que es la misma estructura de lista y la otra que es una pila normal.

El área adicional que se requiere en estos algoritmos es el bit de marca que se emplea para identificar las localidades activas, y en los que invierten apuntadores otro bit que se utiliza para indicar cual de los dos apuntadores CAR o CDR contiene una dirección artificial.

La mejores rutinas que marcan de Recolección de Basura, tienen un tiempo de ejecución [Knu 68] de la forma:

c1N + c2M donde, c1 y c2 son constantes,
N es el número de nodos marcados,
M es el número total de localidades del Area de Trabajo.

Por lo que, M - N es el número de localidades basura que se recuperan y la cantidad de tiempo requerido para unir estas localidades (al recorrer secuencialmente el Area de Trabajo), es por localidad:

(c1N + c2M) / (M - N)

Si sustituímos N por pM en la fórmula tenemos que:

(c1pM + c2M) / (M - pM) = (c1p + c2) / (1 - p)

El parámetro p es un factor de ocupación de la memoria. Si p = 3/4, o sea que el Area de Trabajo está llena las tres cuartas partes, tenemos que 3c1 + 4c2 son las unidades de tiempo que toma por localidad al ser unida a la lista disponible.

La Recolección de Basura es ineficiente cuando el Area de Trabajo (memoria del intérprete) se llena, y es muy eficiente cuando la demanda de área de memoria es pequeña.

El grado de dificultad de estos algoritmos en cuanto a su programación se refiere, en esta implantación, estriba en definir bien las máscaras que se utilizan para prender o apagar bits de la palabra para cuestión de marcar y desmarcar respectivamente las localidades de apuntadores.

El buen funcionamiento del intérprete garantiza que el Recolector está trabajando correctamente mientras que una falla en éste, hace que se destruyan algunas estructuras de lista que son necesarias para los procesos siguientes durante la ejecución del intérprete, lo que hace que se detecte facilmente cualquier error de programación del algoritmo del Recolector. Esto permite que sea fácil depurar los algoritmos.

La ejecución del intérprete debe ser detenida cuando el Recolector libere cero localidades de memoria o bien un número muy reducido, esto a fin de evitar un ciclo dentro de la ejecución.

7. CONCLUSIONES.

De este breve estudio que hemos realizado sobre el problema que representa el identificar el área de memoria que es basura, hemos visto que el método para regresar esta área a la lista disponible es una parte esencial de cualquier sistema de procesamiento de listas. En este trabajo vimos cuales han sido las técnicas que han surgido para resolver el problema, siendo la de Recolección de Basura la que más nos interesó por ser la que satisface de forma eficiente la identificación y por lo mismo la recuperación del área que es basura.

Ya que esta técnica se creó como una innovación dentro del diseño del intérprete del lenguaje LISP, es que la estudiamos dentro del contexto de este lenguaje.

Al analizar con detalle esta técnica hemos visto que contempla dos etapas, una que identifica el área que es basura para que pueda ser recuperada con el fin de que el intérprete la pueda volver a usar para continuar con su funcionamiento, y la otra que consiste en recolectar toda el área basura en la lista disponible para que el intérprete la pueda accesar.

Ahora bien de estas dos etapas, escogimos estudiar la primera ya que resulta ser la más interesante por su dificultad en identificar el área que es basura.

De esta etapa hemos visto que existen diversos algoritmos que la resuelven, aunque cada uno de diferente forma, lo que produce su determinado grado de eficiencia.

Revisé la bibliografía sobre esta técnica y me encontré con que desde que fué inventada por John McCarthy se ha seguido su estudio por otros investigadores. Ahora bien, la más antigua se refiere a la etapa que identifica el área que es basura y la más reciente corresponde al estudio de la segunda etapa con la modificación de que se compacta el área en el caso de que se esté usando un número variable de localidades de memoria para formar un nodo de tamaño variable de la estructura de lista.

En la bibliografía incluyo algunas referencias recientes [Wes 72], [Ste 75], [Mor 78], sobre la parte de compactar el área disponible para luego recolectarla, en el caso de que se esté trabajando con elementos de la estructura de lista de tamaño variable.

En resumen la Recolección de Basura es una técnica que a primera vista resulta ser sencilla pero que al introducirse

más a su estudio estudio, es un método que resuelve un problema difícil en cuanto a la identificación del área basura se refiere ya que ésta debe hacerse de una manera relativamente rápida para evitar que el intérprete suspenda su funcionamiento por mucho tiempo, de lo contrario representaría una pérdida de tiempo considerable en las aplicaciones que se hicieran al usar este lenguaje.

8. APENDICE.

8.1. FUNCIONES BASICAS DE LISP.

La siguiente es una lista de algunas funciones básicas de LISP.

Función de inhibición.

QUOTE	(QUOTE <exp-s>) Inhibe la evaluación. Es equivalente a '<exp-s>.

Funciones de manejo de listas.

CAR	(CAR '<lista>) Regresa el primer elemento de la lista.
CDR	(CDR '<lista>) Regresa la lista sin el primer elemento.
CONS	(CONS '<exp-s> '<lista>) Añade la exp-s a la lista como su primer elemento.
LIST	(LIST '<exp-s 1> ... '<exp-s n>) Regresa una lista de n elementos construída por sus n argumentos.
APPEND	(APPEND '<lista 1> '<lista 2>) Regresa una sola lista, cuyos elementos son los elementos de las dos listas.

Funciones aritméticas.

PLUS	(PLUS <numero 1> ... <numero n>) Regresa la suma de todos los números.
DIFFERENCE	(DIFFERENCE <numero 1> ... <numero n>) Regresa el resultado de restar el segundo número en adelante del primero.

TIMES	(TIMES <numero 1> ... <numero n>) Regresa el producto de todos los números.
QUOTIENT	(QUOTIENT <numero 1> ... <numero n>) Regresa el resultado de dividir el primer número por todos los demás.

Funciones lógicas y de relación.

AND	(AND '<exp-1> ... '<exp-n>) Regresa NIL si cualquiera de las exp-s es NIL, de otra manera regresa <exp-n>.
OR	(OR '<exp-1> ... '<exp-n>) Regresa la primera exp-s diferente de NIL. Si todas las exp-s son NIL regresa NIL.
NOT	(NOT <exp-s>) Regresa T si la exp-s es NIL. Regresa NIL si la exp-s es T. Es equivalente a NULL.
NULL	(NULL '<exp-s>) Regresa T si la exp-s es la lista vacía. Regresa NIL si la exp-s no es la lista vacía. Es equivalente a NOT.
ATOM	(ATOM '<exp-s>) Regresa T si la exp-s es un átomo. Regresa NIL si la exp-s no es un átomo.
NUMBER	(NUMBER <exp-s>) Regresa T si la exp-s es un número. Regresa NIL en otro caso.
GREATER	(GREATER <numero 1> ... <numero n>) Regresa T si el primero es mayor que el segundo y NIL en caso contrario.
EQ	(EQ <exp-s 1> <exp-s 2>) Regresa T si las dos exp-s son átomos idénticos.

Función de asignación.

SETQ	(SETQ <atomo> '<exp-s>) Regresa la exp-s. Además la exp-s es el valor del átomo.

Función de evaluación.

EVAL — (EVAL <exp-s>)
Regresa el valor de la exp-s evaluada.

Otras funciones

DEFINE — (DEFINE (<nom-func> LLL(<lista args>)) <cuerpo>)
Define una función tipo LLL (donde LLL es LAMBDA o NLAMBDA).
Regresa el valor asociado al nombre de la función (nom-func).

COND — (COND (<prueba-1> ... <resultado-1>)
. . .
(<prueba-n> ... <resultado-n>))

Se evalúa la parte <prueba> de cada lista, hasta encontrar una que tenga un valor diferente de NIL.
Entonces, se evalúan todas las expresiones de esa lista y el valor de COND será la última expresión evaluada.

MAPCAR — (MAPCAR <especificación-función> <args>)
Aplica una función a una lista o a una lista de argumentos.

APPLY — (APPLY <función> <lista de argumentos>)
Opera sobre los argumentos con la función dada, como si la función apareciera como el primer elemento de la lista.

8.2. EJEMPLO DE EJECUCION DEL RECOLECTOR.

En este ejemplo mostraremos como funciona el Recolector de Basura durante la ejecución del intérprete.

Hay dos formas de que se ejecute el Recolector: una es tecleando la expresión simbólica (RECOLECTOR T) para hacerlo funcionar cuando todavía no se agota el espacio de la lista disponible, y la otra es cuando funciona de forma automática al agotarse las localidades de la lista disponible.

En la ejecución del intérprete, cada vez que funcione el Recolector aparecerán tres cifras que indican el estado de la memoria antes de ejecutarse el Recolector, y dos cifras que enseñan como está la memoria después de ejecutarse el

Recolector. La descripción de estas cinco cifras se da a continuación:

La primera muestra el número de localidades disponibles que existen antes de que el intérprete asigne localidades para efectuar la representación interna de la expresiones simbólicas leídas o evaluadas.

La segunda indica el número de localidades de apuntadores que asignó el intérprete para representar las expresiones simbólicas. Estas localidades son las activas, de las cuales algunas pueden ser basura.

La tercera muestra el número de localidades disponibles que existen después de que el intérprete ya asignó localidades de apuntadores.

La cuarta indica el número de localidades de apuntadores que liberó el Recolector de Basura. Estas son las localidades basura.

La quinta muestra el número total de localidades que están disponibles después de ejecutarse el Recolector.

```
 RUN LISP
TT6>
(RECOLECTOR T)

      Localidades disponibles inicialmente                  4955

      Localidades activas (algunas o todas son basura)      2

      Localidades disponibles despues de la asignacion      4953

          ** SE EJECUTA EL RECOLECTOR DE BASURA **

      Localidades basura que fueron liberadas               0

      Localidades disponibles en total                      4953

=>T

(EXECUTE "UTILERIA.LIS")

UTILERIA.LIS
EQUAL MEMBER REVERSE POWER ADD1
 T SUB1 COUNTATOMS DEPTH MAX
 T BREAK LAST LENGTH ROTATE-L ROTATE-R
 T UNION INTERSECTION FACTORIAL
 T CDL MOD DIV INTERSECTP SAMESETP
 T LDIFFERENCE T =>T
```

```
(RECOLECTOR T)
        Localidades disponibles inicialmente                    4955
        Localidades activas (algunas o todas son basura)        803
        Localidades disponibles despues de la asignacion        4152
            ** SE EJECUTA EL RECOLECTOR DE BASURA **
        Localidades basura que fueron liberadas                 115
        Localidades disponibles en total                        4267
=>T

(EXECUTE "REINA.LIS")
REINA.LIS
INTENTA CONFLICTO REINAS T =>T

(RECOLECTOR T)
        Localidades disponibles inicialmente                    4267
        Localidades activas (algunas o todas son basura)        234
        Localidades disponibles despues de la asignacion        4033
            ** SE EJECUTA EL RECOLECTOR DE BASURA **
        Localidades basura que fueron liberadas                 17
        Localidades disponibles en total                        4050
=>T

(REINAS 4)
((1 2 ) (2 4 ) (3 1 ) (4 3 ) )
 ((1 3 ) (2 1 ) (3 4 ) (4 2 ) )
 =>TERMINE

(RECOLECTOR T)
        Localidades disponibles inicialmente                    4050
        Localidades activas (algunas o todas son basura)        2427
        Localidades disponibles despues de la asignacion        1623
            ** SE EJECUTA EL RECOLECTOR DE BASURA **
        Localidades basura que fueron liberadas                 2427
        Localidades disponibles en total                        4050
=>T
```

```
(REINAS 5)
((1 1 ) (2 3 ) (3 5 ) (4 2 ) (5 4 ) )
 ((1 1 ) (2 4 ) (3 2 ) (4 5 ) (5 3 ) )
 ((1 2 ) (2 4 ) (3 1 ) (4 3 ) (5 5 ) )

      Localidades disponibles inicialmente                4050

      Localidades activas (algunas o todas son basura)    4050

      Localidades disponibles despues de la asignacion    0

          ** SE EJECUTA EL RECOLECTOR DE BASURA **

      Localidades basura que fueron liberadas             4025

      Localidades disponibles en total                    4025

((1 2 ) (2 5 ) (3 3 ) (4 1 ) (5 4 ) )
 ((1 3 ) (2 1 ) (3 4 ) (4 2 ) (5 5 ) )
 ((1 3 ) (2 5 ) (3 2 ) (4 4 ) (5 1 ) )
 ((1 4 ) (2 1 ) (3 3 ) (4 5 ) (5 2 ) )

      Localidades disponibles inicialmente                4025

      Localidades activas (algunas o todas son basura)    4025

      Localidades disponibles despues de la asignacion    0

          ** SE EJECUTA EL RECOLECTOR DE BASURA **

      Localidades basura que fueron liberadas             4022

      Localidades disponibles en total                    4022

((1 4 ) (2 2 ) (3 5 ) (4 3 ) (5 1 ) )
 ((1 5 ) (2 2 ) (3 4 ) (4 1 ) (5 3 ) )
 ((1 5 ) (2 3 ) (3 1 ) (4 4 ) (5 2 ) )

      Localidades disponibles inicialmente                4022

      Localidades activas (algunas o todas son basura)    4022

      Localidades disponibles despues de la asignacion    0

          ** SE EJECUTA EL RECOLECTOR DE BASURA **

      Localidades basura que fueron liberadas             4024

      Localidades disponibles en total                    4024

=>TERMINE
```

9. BIBLIOGRAFIA.

[All 78] Allen, John, ANATOMY OF LISP, Mc Graw Hill, New York, 1978.

[Coh 82] Cohen, J., "Computer-Assisted Microanalysis of Programs", CACM 25, 10, 724-733, 1982.

[Col 60] Collins, G. E., "A Method for Overlapping and Erasure of Lists", CACM 3, 12, 655-657, Dec. 1960.

[Gel 60] Gelernter, H., Hansen, J., Gerberich, C., "A FORTRAN-compiled List Processing Language", JACM 7, 87-101, 1960.

[Knu 68] Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, VOL. I: FUNDAMENTAL ALGORITHMS, Addison Wesley, Reading, Mass., 1968.

[Mau 72] Maurer, W. D., A PROGRAMMER'S INTRODUCTION TO LISP, American Elsevier Publishing Company, Inc., New York, 1972.

[McB 63] McBeth, H., "On the Reference Counter Method", (letter) CACM 6, 9, p. 575, Sep. 1963.

[McC 60] McCarthy, John, "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I", CACM 3, 4, 184-195, April 1960.

[Mor 78] Morris, F.L., "A Time-and Space-Efficient Garbage Compaction Algorithm", CACM, 21, 8, 662-665, 1978.

[New 57] Newell, A., and Shaw, J., and Simon, H., "Programming the Logic Theory Machine", Proc. Western Joint Computer Conference", Feb. 1957.

[New 60] Newell, A., and Tonsue, F., "An Introduction to Information Processing Language V", CACM 3, 205-211, 1960.

[Org 78] Organick, E., Forsythe, A., Plummer, R., PROGRAMMING LANGUAGE STRUCTURES, Academic Press, 1978.

[Pra 75] Pratt, Terrence W., PROGRAMMING LANGUAGES: DESIGN AND IMPLEMENTATION, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Sam 72] Sammet, J., "Programming Languages: History and Future", CACM 15, 7, 601-610, 1972.

[Sch 67] Schorr, H., and Waite, W., "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures", CACM 10, 8, 501-506, Aug. 1967.

[Sik 76] SikLossy L., LETS's TALK LISP, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[Ste 75] Steele, G.Jr., "Multiprocessing Compactifying Garbage Collection", CACM, 18, 9, 495-508, 1967.

[Tom 83] Tomasena, F. M., MANUAL DE LISP para la PDP-11/34, Facultad de Ciencias, UNAM, 1983.

[Tuc 79] Tucker, T. S., "The Design of an M6800 LISP Interpreter", BYTE 4, 8, Aug. 1979.

[Wai 64] Waite, W., and Schorr, H., "A note on the Formation of a Free List", CACM 8, 478, Aug. 1964.

[Wes 72] Wesbreit, B., "A generalized Compactifying Garbage Collector", COMPUTER JOURNAL 15, 3, 204-208, Aug. 1972.

[Wei 62] Weizenbaum, J., "Knotted List Structures", CACM 5, 3, 161-165, Mar. 1962.

[Wei 63] Weizenbaum, J., "Symmetric List Processor", CACM 6, 9, 524-544, Sep. 1963.

[Wex 81] Wexelblat, R. L., HISTORY OF PROGRAMMING LANGUAGES, ACM MONOGRAPH SERIES, Academic Press, Inc., New York, 1981.

[Woo 61] Woodward P. M., and Jenkins D. P., "Atoms and Lists", COMPUTER JOURNAL 4, 47-53, 1961.