Universidad Nacional Autónoma de México

Facultad de Ingeniería

IMPLEMENTACION DEL LENGUAJE FUNCIONAL LISPKIT

T E S I S

Que para obtener el Título de
INGENIERO EN COMPUTACION

P r e s e n t a n

RAYMUNDO JOSE LUIS OREA LOZANO
SALVADOR CARREON IBARRA

Dir. Ing. LUIS CORDERO B.

México, D. F. 1987

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

APENDICES

INTRODUCCION

La simulación de una computadora, ha surgido como una herramienta necesaria en el diseño de nuevas arquitecturas de máquinas, así como la implementación de nuevos lenguajes de programación. Generalmente la idea es, diseñar una computadora apropiada a la ejecución del lenguaje que se desea implementar. Si no se hiciera bajo la técnica de simulación, implicaría altos costos en la construcción de nuevas máquinas. Las ventajas que presenta el implementar una máquina virtual son justificables, como es la portabilidad del compilador a implementar, y la optimización de el conjunto de instrucciones que que se pueden tener dentro de ésta.

Este trabajo tiene como finalidad principal, la de implementar un lenguaje funcional llamado LISPKIT LISP.

La implementación consiste, en desarrollar un simulador para una máquina abstracta llamada SECD, la cual permita ejecutar programas escritos en Lispkit. Para poderlos ejecutar es necesario, cargar dentro de ésta máquina abstracta el compilador de Lispkit, que proporciona Peter Henderson en su libro de "Functional Programming". El compilador se presenta en forma fuente y objeto, teniendose la característica de que el fuente está escrito en Lispkit, por lo que permite experimentar con este.

El lenguaje funcional Lispkit es una variante o subconjunto de el Lisp original creado por McCarthy a fines de la decada de los 50, por lo tanto cumple con las mismas características que se tienen en éste, como es: funcionalidad, manejo de datos simbólicos, recursividad, etc.

Además del objetivo principal, se pretende otras finalidades particulares como es: la exploración de un lenguaje relativamente reciente con un estilo de programación funcional, su estructura, y el tipo de aplicaciones que se pueden tener con este clase de lenguajes basadas en técnicas de programación funcional.

La implementación de Lispkit Lisp se establece en la minicomputadora PDP11/40, usando el sistema operativo RSX11-M V3.2, y el lenguaje Fortran IV. Para poder llevar a cabo la implementación, es necesario realizar una serie de actividades, como son:

- La simulación de la máquina, la cual comprende, implementar el conjunto de instrucciones básicas que la componen. Un diseño en el tamaño de los registros principales, así como en el tamaño de la palabra en memoria, y la dimensión de esta; un manejo apropiado en la entrada y salida de datos, y

finalmente un control general de la máquina.

- Elaboración de ejemplos de programas de aplicación y pruebas al sistema. El ejecutar el compilador fuente dentro de la máquina abstracta, es una prueba importante para saber que la implementación está bien, para reforzar estas pruebas se establecerán una diversidad de ejemplos con aplicaciones, muy diferentes a las que se tienen en lenguajes convencionales, como por ejemplo, el programa de diferenciación simbólica, que lo que hace es obtener la derivada para fórmulas, las cuales tengan variables,constantes y las operaciones de "+" y "x".

- Un manual del sistema que explica los componentes de Lispkit, así como su estructura, el mecanismo para armar la tarea (overlay), y los archivos que lo componen, con la finalidad de quien tenga la inquietud de modificar lo pueda hacer.

- Un manual de usuario que ayudará en el manejo del sistema. Este documento guiará al usuario, a como compilar y ejecutar programas en Lispkit.

Dado el campo que abarca la elaboración de esta tésis, se puede tener alguna utilidad como instrumento didáctico en el área de computación, como son:

- Un laboratorio de prácticas respecto a compiladores e interpretes, donde es posible aplicar cambios semánticos como sintácticos, o establecer inovaciones al compilador de Lispkit.

- Un curso de programación funcional. Actualmente se ha incrementado en el diseño de sistemas, el concepto de funcionalidad, por lo que es importante conocer el lenguaje de programación Lisp, o alguna de sus variantes, donde se presentan técnicas, para la elaboración de programas funcionales.

- Como parte de un curso sobre aplicaciones de inteligencia artificial, en el cual las aplicaciones desarrolladas, tienen a Lisp o sus variantes, como el lenguaje apropiado para este tipo de aplicaciones.

- Finalmente como apoyo en cursos relacionados con estructura de datos y programación estructurada.

Este libro se divide en seis capítulos, en el primer capítulo se establece, una comparación entre los lenguajes funcionales y los lenguajes convencionales, así como la estructura de cada uno de éstos. En el capítulo dos, se presenta todo lo referente al lenguaje a implementar, se da un panorama completo respecto a su tipo de datos de Lispkit, así como las funciones o expresiones básicas que lo componen y algunos ejemplos de éstas; en el mismo capítulo, también se abordan algunas técnicas necesarias para un estilo de programación funcional. El capítulo tres refiere a todo lo que es la máquina virtual es decir se establece la arquitectura y el conjunto de

instrucciones que maneja. En el capítulo cuatro se presenta la semántica del lenguaje, se define el compilador y como se compila cada una de las expresiones que componen el lenguaje. El capítulo cinco trata todo lo referente a la implementación del sistema, la programación de todos los módulos, así como las rutinas de servicio, como el colector de basura, el soporte recursivo que es necesario establecer en Fortran IV para efectos recursivos. En el capítulo seis se muestran una serie de ejemplos aplicados ya con el emulador. Por último , en los apéndices se presentan los manuales de Usuario y del manejo del sistema, y un manual de referencia rápida.

Hacemos patente nuestro agradecimiento a los Ings. Daniel Rios Zertuche y Luis Cordero B. por su amplia colaboración en el desarrollo de este trabajo, sin lo cual éste no hubiera sido posible.

CAPITULO 1

LENGUAJES FUNCIONALES Y LENGUAJES CONVENCIONALES.

El objetivo de este capítulo es explicar los fundamentos y diferencias existentes entre los lenguajes funcionales y los convencionales, así como realizar una comparación entre ellos.

1.1 MODELOS DE SISTEMAS DE COMPUTO

Un modelo es la representación cuantitativa o cualitativa de un sistema. Los modelos se dividen en materiales y formales, y dentro de estos últimos hay 3 tipos : descriptivos, simulativos y formalizativos. Los modelos simulativos consisten de una serie de asociaciones sobre el sistema original, expresadas en un lenguaje especial, que contiene fórmulas y expresiones aritméticas. Los lenguajes de cómputo son de este tipo y dan los elementos para elaborar modelos (programas) de los sistemas de cómputo, en los que se corren dichos programas.

Algunos modelos son abstracciones, otros son representados por hardware y otros por compiladores o intérpretes. Para examinarlos se debe ver antes el universo de alternativas. Los criterios a usar son:

1.1.1 Criterios para modelos

Fundamentos

> ¿Hay una descripción matemática del modelo que sea elegante y concisa.?
>
> ¿Es útil para proveer ayudas sobre el comportamiento del modelo?
>
> ¿Es el modelo tan complejo que su descripción es voluminosa y poco matemática.?

Tipo de semántica.

> ¿El programa transforma estados (que no son programas) sucesivamente hasta que llega a un estado terminal (semántica de transición de estado).?
>
> ¿Los estados son sencillos o complejos?, o puede un programa ser sucesivamente reducido a programas sencillos para conducir a una "forma normal de programa", la cual es el resultado (semántica de reducción)

Claridad y utilidad conceptual de programas.

LENGUAJES FUNCIONALES Y LENGUAJES CONVENCIONALES

¿Son los programas del modelo expresiones claras de un proceso de cómputo? ¿Envuelven conceptos que nos ayudan a formular y razonar acerca de los procesos?.

Clasificación de modelos.

Con los criterios anteriores podemos caracterizar 3 modelos para sistemas de cómputo :

Modelos operacionales simples
Modelos aplicativos
Modelos Von Neumman

las características de cada uno de ellos son:

Modelos operacionales simples:

Ejemplos :Máquina de Turing, varios autómatas
Fundamentos :Concisos y útiles
Semántica :Transición de estados con estados muy sencillos.
Claridad :Programas poco claros y conceptualmente poco útiles.

Modelos aplicativos:

Ejemplos :Notación LAMBDA de Church, LISP puro
Fundamentos :Concisos y útiles
Semántica :Semántica de reducción
Claridad :Son claros y conceptualmente útiles.

Modelos Von Neumann:

Ejemplos :Computadoras VN, lenguajes convencionales
Fundamentos :Complejos, voluminosos, no útiles
Semántica :Transición de estados con estados complejos.
Claridad :Moderamente claros y no muy útiles conceptualmente.

1.2 ESTRUCTURA DE LOS LENGUAJES DE COMPUTO

En un lenguaje de cómputo se pueden distinguir dos partes; una, su infraestructura (que da las reglas generales en el sistema) y otra, sus partes cambiantes, cuya existencia es anticipada por la infraestructura, pero cuyo comportamiento específico no es determinado por la infraestructura. Por ejemplo la declaración FOR y casi todas las demás declaraciones son parte de la infraestructura de ALGOL, pero las funciones de biblioteca y procedimientos definidos por el usuario son partes cambiantes. Así la infraestructura describe las características fijas y provee un medio ambiente general para sus partes cambiantes.

Los lenguajes de infraestructura pequeña pueden acomodar una gran

variedad de características como partes cambiantes, lo que permite a estos lenguajes soportar diversas características y estilos sin cambiar el lenguaje en si, de hecho, el elemento más importante en un lenguaje es poder combinar formas que puedan ser utilizadas en general para construir nuevos procedimientos a partir de otros ya definidos

1.3 LENGUAJES CONVENCIONALES.

Antes de explicar los llamados lenguajes convencionales (o derivados de la computadora VON NEUMMAN), es necesario dar una explicación de la misma. Básicamente la computadora VON NEUMANN(VN) consta de 3 partes: CPU, memoria y un canal de trasmisión, en el cual se puede trasmitir una sola palabra entre el CPU y la memoria (y enviar una dirección a la memoria). Dado que el programa lo que hace es modificar la memoria enviando y recibiendo a través del canal información, este canal se vuelve un cuello de botella, al hacerlo con una palabra a la vez.

Lamentablemente, mucho de lo que envía no son datos útiles, sino que solo son nombres de datos, así como operaciones y datos usados solo para computar tales nombres. Antes que una palabra pueda ser enviada a través del canal, su dirección debe estar en el CPU, entonces debe ser enviada a través del canal desde la memoria o generada por alguna operación de memoria. Si la dirección es enviada desde la memoria, entonces su propia dirección debe ser enviada de la memoria o generada en el CPU, y así continúa. Si la dirección es generada en el CPU, debe de ser generada siguiendo una regla fija (como incrementar el contador) o por una instrucción que fue enviada a través del canal, en cuyo caso la dirección de dicha instrucción debió de ser enviada, etc, etc...

Debe haber una forma menos primitiva de hacer grandes cambios que metiendo y sacando palabras a través del canal. Esto nos ha inducido a pensar en una palabra a la vez, en vez de hacerlo con grandes unidades conceptuales de datos.

Los lenguajes convencionales son basicamente versiones complejas de la computadora VN, usan variables para imitar celdas de almacenamiento, instrucciones de control elaboran sus saltos y pruebas y declaraciones de asignación imitan la recuperación, almacenamiento y los cálculos aritméticos. La declaración de asignaciones es el cuello de botella de los lenguajes VN y también nos hace pensar en una palabra a la vez. Ya que así opera dicha declaración de asignamiento.

Un programa típico tiene un conjunto de asignaciones que usan variables subscritas, y cada declaración de asignación produce un resultado de una palabra a la vez. El programa ejecuta dichas declaraciones muchas veces, alterando los valores de los índices de las variables, para lograr el cambio deseado en el almacenamiento y además debe controlar el flujo de palabras a través del cuello de botella que es el asignamiento, así como el anidamiento de declaraciones de control para lograr las

repeticiones.

El asignamiento divide la programación en dos mundos, uno, el del lado derecho el cual es un mundo ordenado de expresiones (aunque muchas veces se vuelve desordenado por efectos colaterales). y otro el lado izquierdo es el mundo de las declaraciones. En donde la declaración primaria es la de asignamiento. Todas las otra declaraciones se utilizan para realizar el computo, que se basa en el asignamiento.

Los lenguajes convencionales de alto nivel proveen de procedimientos o subroutinas, las cuales son basadas en la idea de funciones, inclusive se pueden definir nuevas funciones (esto es, se hace uso del concepto de composición de funciones). Sin embargo solo una cantidad limitada de funciones pueden ser definidas en esos lenguajes, y para programas prácticos se deben usar procedimientos más generales. Además en estos lenguajes es muy común alejarse del concepto de función cuando se permiten los siguientes casos:

-Procedimientos que en vez de regresar un valor, le asignan valores a sus parámetros.

-Cuando el valor de uno de los argumentos es reemplazado por el resultado

De lo anterior se observa que se computa "por efecto" (o sea, no solo se calculan nuevos valores a partir de otros, sino que además tiene el efecto lateral de asignar valores a alguno de los parámetros en vez de solo calcular valores. Y en ocasiones se altera una variable que ni siquiera es parámetro sino de otro tipo (las que llegan por common) lo que dificulta el entender los programas.

Respecto a la estructura de los lenguajes VN, parecen tener una infraestructura inmensa y partes cambiantes pequeñas ¿porque?. La respuesta esta dada por 2 problemas de los VN. El primero es el estilo de programación de una palabra a la vez, lo que obliga a tener una semántica muy cercana a el estado. Esto provoca que cada detalle de cada característica deba ser construido en el estado y en su regla de transición. Así, un lenguaje VN se vuelve rígido y con una enorme infraestructura.

El segundo problema es que sus partes cambiantes tienen poco poder expresivo los lenguajes VN proveen solo formas combinatorias primitivas y además la estructura VN presenta problemas para su uso total. Un obstáculo para el uso de las formas combinatorias es la división entre el mundo de la expresión y el mundo de la declaración. Formas funcionales naturalmente pertenecen a el mundo de las expresiones, pero no importa que tan poderosas sean solo pueden producir resultados de una palabra. Y es en el mundo de las declaraciones que esos resultados de una palabra deben ser combinados en el resultado final.

LENGUAJES FUNCIONALES Y LENGUAJES CONVENCIONALES

1.1 LENGUAJES FUNCIONALES

Una vez expuestos los defectos de los lenguajes VN, se hace evidente la necesidad de lenguajes con una mejor estructura para evitar los problemas de los lenguajes convencionales.

Una alternativa es un estilo funcional de programación sin variables convencionales (o sea aquellas que pueden llegar a cambiar su valor durante la ejecución del proceso) los lenguajes que utilizan este estilo son llamados lenguajes funcionales (LF), y reciben tal nombre por usar la función como bloque básico de construcción. Estos lenguajes se basan en el uso de formas combinatorias para construir programas. Estas formas, más definiciones de funciones simples son los unicos elementos usados para agregar nuevas funciones a las ya existentes, y además no usan asignacción entre variables.

Recordemos la definición de una función: es una regla de correspondencia entre cada miembro de un conjunto de elementos llamado DOMINIO a los cuales corresponde uno solo de un conjunto al que se denomina RANGO.

Usualmente se denota una función en la siguiente forma:

$$F(x) = y$$

Donde "x" es un elemento del dominio, "F" es la regla que se aplica y "y" es un elemento del rango.

En los lenguajes de programación convencionales es común utilizar funciones, por ejemplo:

$$Min(x,y) = \text{if } x >= y \text{ then } y \text{ else } x$$

Las caracteristicas que debe tener un lenguaje para que unicamente utilice funciones son : el lenguaje debe tener la capacidad de crear nuevas funciones en base a algunas ya definidas (este método se llama "composición de funciones" y consiste en crear nuevas funciones anidando las ya existentes).

Ejemplo:

Menor(x, y, z)=Min(Min(x, y), z)

Además se debe disponer de un grupo de funciones básicas ya definidas para poder usar la composición de funciones. Al conjunto de funciones resultante se le puede considerar un programa funcional.

En 1960 LANDIN definió lo que llamó "estructura aplicativa de expresiones" para lenguajes matemáticos (que se aplica también a lenguajes de computación). En esta estructura cada expresión puedes ser dividido en sus partes, los cuales serán operadores u operandos. Los operadores denotan funciones y los operandos denotan valores.

Ejemplo usando funciones aritméticas.

sea:

a+b*c

En donde se encuentran 2 operadores que son funciones (+ y x) y tres operandos(a, b, c). Si se definen explícitamente como:

suma(x,y)=x+y
mul(x,y)=x*y

Y se usan los nombres de las funciones en vez de los operadores la operación algebraica se vuelve

add(a, mul(b, c))

Con lo que se llega a una estructura aplicativa

Es de notarse que el valor de una expresión que usa la estructura aplicativa esta determinado exclusivamente por el valor de sus partes, de esta forma si la misma expresión ocurre dos veces en el mismo contexto su valor será el mismo. Los lenguajes que tienen esta propiedad para todas sus expresiones se llaman LENGUAJES APLICATIVOS o LENGUAJES FUNCIONALES, para denotar que utilizan funciones en forma dominante

El primer lenguaje de programación que fue funcional lo diseño MCCARTHY en 1960 y su nombre fue LISP. Mas adelante se verá que el lenguaje que se implementó en esta tesis es una derivación de este lenguaje.

Los LF no computan por efecto, el programador solo puede definir funciones que calculan valores en base a los argumentos (sin alterarlos). Además la asignación no existe. Por lo tanto el concepto tradicional de variable se pierde. En los LF las variables solo sirven para nombrar los valores de los argumentos de las funciones y estan asociadas con esos valores constantes solo durante la duración de la evaluación de la expresión algebráica que define el resultado de la función.

Los LF tienen dos características de gran poder:

Estructuración de datos.- el hecho de que estructuras completas de datos se puedan manejar como un solo valor y puedan ser pasados a funciones como argumentos y regresados por funciones como resultado es muy poderoso, y lo que es más significativo una vez que una estructura de datos ha sido creada, no puede ser alterada en su valor. Por supuesto que puede ser incorporado a otras estructuras pero cada vez que el programa requiera su valor estará disponible. Esta capacidad es de importancia fundamental para el uso de los LF y requiere un gran cuidado al implementarlo, si se desea una eficiencia adecuada.

Funciones de alto nivel.- esta característica permite crear funciones que toman una función como argumento o producen una función como resultado. El uso de tales funciones puede conducir a programas que son muy cortos en base a la computación que realizan.

1.5 COMPARACION ENTRE UN LENGUAJE CONVENCIONAL Y UN LENGUAJE FUNCIONAL

Ahora se verá una comparación entre los dos tipos de lenguajes que hemos expuesto para ver cual es el mejor. Tomando como ejemplo el cálculo del producto punto de dos vectores. Si se usan dos vectores a y b con "n" elementos, el producto punto sería la sumatoria de los productos de a(i), b(i) para i de 1 a n

$$a(1)*b(1) + a(2)*b(2) + a(3)*b(3) + ...a(n)*b(n)$$

Programa VN para el producto punto.

```
For i=1 step 1 until n do
        C=c+a(i)x b(i)
```

Lo relevante es:

-Sus declaraciones operan en un "estado" invisible siguiendo reglas complejas.

-No es jerárquico, excepto por el lado derecho del asignamiento, no construye entidades complejas de pequeñas (los programas grandes, sin embargo, si lo hacen.

-Es dinámico y repetitivo, lo debe uno ejecutar mentalmente para entenderlo.

-Computa una palabra a la vez por repetición (del asignamiento) y por modificación (de la variable i)

-Parte de los datos (n) está en el programa lo que obliga a trabajar con vectores de tamaño n, al menos que se introduzca n como parámetro

-Nombra sus argumentos, solo puede ser usado para vectores a y b. Para ser general requiere de una declaración de procedimiento lo que envuelve cosas complejas (llamada por nombre .vs. llamada por valor, commons, etc).

-Sus operaciones "cotidianas" son representadas por símbolos en lugares esparcidos (en la declaración for y los índices de asignamiento) esto hace imposible consolidar las operaciones cotidianas, que son las más comúnes en operadores sencillos, poderosos y muy útiles. Así programando con esas operaciones uno debe siempre empezar encuadrandolas escribiendo

```
For i=....
For j=....
```

Seguido por las declaraciones de asignación salpicadas con i's y j's

Programa funcional para producto punto

Se definen tres funciones, sin caer en detalles particulares de como se conformarían.

trans; que toma el elemento i de a y b y los coloca juntos. por ejem:
dados los vectores a(1 2 3)y b(4 5 6)
trans (a b) dejará:
((1 4) (2 5) (3 6))

multi; que multiplica dos elementos que se hallen como los deja trans.

dada una lista como la que deja trans
mul ((1 4) (2 5) (3 6)) = (4 10 18)

add ; está suma un conjunto de valores
aplicado a una lista de valores como la obtenida de mul
add (4 10 18) =(32)

así la función quedará
add(mul(trans(a b)))

Las propiedades son:

-Opera solo con sus argumentos, no hay estados ocultos o reglas de transición complejas. Hay solo dos tipos de reglas, una aplica una función a su argumento, la otra obtiene la función denotada por la composición de funciones f y g que son los parámetros de la forma.

-Es jerárquico, se construye de tres funciones simples (add, mul, trans) y de una forma funcional (la composición de funciones).

-Es estática y no repetitiva, en el sentido de que su estructura ayuda a entenderla sin que se ejecute mentalmente.

-Opera con unidades conceptuales completas, no palabras, tiene 3 pasos y ninguno se repite.

-No incorpora datos en el programa, es completamente general, trabaja para cualquier par de vectores.

-No nombra sus argumentos, se puede aplicar a cualquier par de vectores sin ninguna declaración de procedimiento o reglas complejas de substitución.

-Utiliza formas cotidianas y funciones generalmente útiles en otros programas, de hecho solo * y + no estan

involucradas con operaciones cotidianas. Esas formas y funciones se pueden combinar con otras para crear operadores cotidianos de alto nivel.

En base a lo anterior se puede deducir que los lenguajes funcionales presentan muchas ventajas sobre los convencionales.

CAPITULO 2

EL LENGUAJE LISPKIT

Este capitulo presenta en forma general la estructura del lenguaje manejado a lo largo del proyecto. Peter Henderson implanto Lispkit en base al Lisp original creado por McCarthy, por lo cual cumple con las mismas caracteristicas que se tienen en este. Actualmente se han implantado otros subconjuntos pero modificados a las condiciones de una máquina convencional y en los que se ha perdido la funcionalidad pura, este concepto es muy importante dentro de la implementación ya que para definición de funciones se mantiene el esquema téorico de expresiones Lambda establecido por Alonzo Church, por lo pronto se mencionará algunas de las principales caracteristicas que se tienen en Lispkit, y posteriormente se verá como es que se mantiene esa funcionalidad pura:

i) Lispkit al igual que Lisp y sus derivados es un procesador de listas dado que la estructura de sus datos mantiene precisamente una organización de listas.

ii) Es un lenguaje funcional (también se les conoce como lenguajes aplicativos), por lo tanto presenta un estilo de programación que utiliza la aplicación de funciones como estructura de control.

iii) Como consecuencia de lo anterior las asignaciones no existen ya que se remplaza a través de la asociación de argumentos de la función con los valores que tenga al aplicarla y el valor que toma el nombre de la función .

iv) La sintáxis se limita a unas cuantas reglas por lo que el lenguaje es más uniforme y general, aunque con esto se sacrifique legibilidad.

v) Es un lenguaje enfocado mas al manejo de datos simbólicos que a datos numericos, aunque no por eso deja de manejarlos.

vi) Mantiene la notacion Lambda. Esta es una propiedad importante ya que es la base para definir funciones en cualquier parte de un programa.

vii) En vez de un flujo de control secuencial o de ciclo, usa invocaciones de funciones anidadas.

Inicialmente se dará un panorama en cuanto a la estructura de lenguaje, se describirá la estructura de los datos y las funciones básicas del lenguaje y el uso de ellas a través de

una serie de ejemplos. En el capitulo 4 se verán algunos topicos inherentes al lenguaje como son recursión, funciones de alto nivel, métodos de construcción para funciones poderosas, etc. Todo esto le da potencialidad al lenguaje.

A través de todos los cápitulo se representarán los programas escritos en Lispkit en una notación abstracta (semejante a la sintáxis de Algol). La razón que se presente de esta manera es la poca legibilidad que se tiene en el lenguaje, esta es una de las pocas desventajas que se tiene ,pero esto permite reducir el número de reglas sintácticas y semánticas. También en el capitulo 4 se darán las reglas para convertir de la notación abstracta a expresiones simbólicas (e.s.) tambien conocida como notación concreta. La forma de distinguir una notacion de otra a través de este documento, en notación abstracta la programación se describirá en minúsculas a lo que la programación en Lispkit se hará en mayúsculas (notación concreta).

2.1 ESTRUCTURA DE DATOS

Así como Fortran y Algol fueron diseñados con la intención de manipular valores numéricos, Lispkit es usado para procesos no numéricos. Maneja un solo tipo de datos simbólicos conocidos como expresiones-S (expresiones simbólicas). Los programas escritos en Lispkit son solo expresiones-S las cuales son evaluadas como funciones previamente definidas. Ejemplos de expresiones-S son las siguientes:

```
( (MONTERREY (ESTA AL (NORTE) ) (EN LA REPUBLICA) )
( (CHIAPAS) (ESTA AL (SUR) ) (EN LA REPUBLICA) )
( (MONTERREY (ESTA MAS AL (NORTE) QUE ) (CHIAPAS) )
```

Como se puede apreciar las expresiones simbólicas no son más que una serie de palabras separados por paréntesis y espacios en blanco. El uso de los paréntesis es una de las condiciones necesarias para establecer estructuras de datos apropiadas, en el primer ejemplo tenemos una expresion-S que a su vez se compone de tres expresiones-S :

```
(MONTERREY)
(ESTA AL (NORTE) )
(EN LA (REPUBLICA) )
```

y las dos ultimas a su vez se componen de otras tantas. Las expresiones-S más simples que podemos tener son conocidas como átomos, en el ejemplo anterior serian los enumerados a continuación:

1) MONTERREY
2) ESTA
3) AL
4) NORTE
5) EN
6) LA
7) REPUBLICA

Los átomos pueden ser de dos tipos simbólicos y numéricos:

los simbólicos son una secuencia de caracteres donde el primero debe ser letra, los simbólicos tienen la característica de ser indivisibles y el mínimo tamaño es al menos de un caracter, los ejemplos anteriormente enumerados son casos de átomos simbólicos. Los átomos numéricos son cualquier número entero.

Los espacios funcionan como separadores entre un átomo y otro. Es recomendable utilizarlos antes y despues de cada una de las expresiones-S para mejor legibilidad.

Las expresiones-S más elaboradas se les conoce como listas, es decir pueden estar formadas por una serie de átomos donde la forma en que esten construidos determinen su estructura. A continuación se presenta un ejemplo de expresiones-S donde se tiene una lista que a su vez se compone de tres sublistas con dos átomos:

((LUIS 17) (CARLOS 21) (ENRIQUE 18))

si a esta misma expresión-S le quitaramos los paréntesis

(LUIS 17 CARLOS 21 ENRIQUE 18)

sería una sola lista conteniendo seis átomos y tendría otro significado. Como se puede apreciar los paréntesis son de vital importancia ya que con estos se determina la estructura de los datos.

En base a lo visto hasta ahora se puede establecer ciertas reglas de construcción, las cuales se resumen en lo siguiente:

1) Un átomo es una expresión-S.
2) Una secuencia de expresiones-S encerradas en paréntesis es también una expresión-S (lista).

Falta por definir otro tipo de expresiones simbólicas que son representadas por la notación punto que no es otra cosa más que la unión de dos expresiones-S a través de un registro compuesto. Un ejemplo es el siguiente:

(A . B)

donde la expresión-S A es unida con la expresión-S B

La descripción sintactica para expresiones-S (léase como E.S) es:

```
< E.S. >      ::= <ATOMO>  |  ( <LISTA > )
<LISTA >      ::= < E.S.>  |  <E.S.>  .  <E.S.>  |
                              <E.S.><LISTA >
```

aqui se definen dos categorias de frases <E.S.> y <LISTA>; en la figura 2.1 se muestrán los diagramas sintácticos para estos dos tipos.

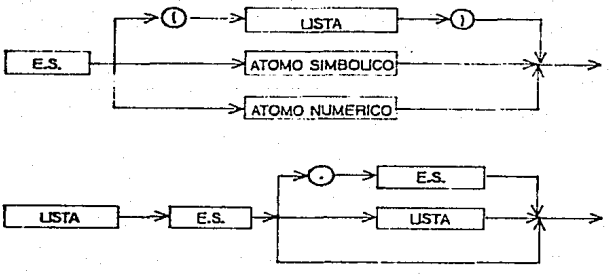


Fig. 2.1 Diagramas de Sintáxis para expresiones-S y Lista de E.S.

Para el caso de expresiones-S simples (átomos) se tiene:

```
<ATOMO SIMBOLICO> ::= <ATOMO LITERAL> | <ATOMO DIGITO>
<ATOMO LITERAL>   ::= <LETRA> | <LETRA> <PARTE ATOMO>
<ATOMO DIGITO>    ::= <DIGITO> | <DIGITO> <PARTE ATOMO>
<PARTE ATOMO>     ::= <VACIO> | <LETRA> <PARTE ATOMO> |
                                <DIGITO> <PARTE ATOMO>
<LETRA>           ::= A | B | C | ... | Z
<DIGITO>          ::= 0 | 1 | 2 | ... | 9
```

Como se ve las expresiones-S son el unico tipo de datos que maneja Lispkit. La pregunta sería, ¿si con un solo tipo de datos es suficiente para representar todas las posibles estructuras de datos simbólicos? la respuesta sería que si. En Lispkit, en base a su estructura podemos representar dos importantes estructuras de datos como son listas y árboles binarios.

Por ejemplo para representar aplicaciones algebráicas sería a través de una lista de átomos como:

```
( ADD X Y )
( MUL X Y )
```

donde se usa una notación prefija en el que se coloca el operador al principio seguido de sus operandos. Respetando esta notación se puede tener expresiones más complicadas, por ejemplo:

4 x (A + B + (7-C))

es representado de la siguiente manera:

(MUL 4 (ADD A B (SUB 7 C)))

Para representar listas en Lispkit, se necesita sólo establecer una convención para mapearlas dentro de expresiones-S:

1) Las expresiones-S para una lista de un solo elemento, es ese elemento apuntando al átomo NIL. El átomo NIL es indicador de fin de la lista (ver fig.2.2.a).

2) Las expresiones-S para una lista de dos o más elementos tal como (x1,x2,...xn) es x1 apuntando a (x2...xn) y asi sucesivamente hasta llegar al átomo de NIL, generalmente el átomo NIL no se incluye en la lista esta va implicitamente (fig. 2.2.b).

3) Un elemento de una lista puede ser una lista. Por ejemplo la lista de 2 elementos ((A) (C D)) el último es una lista (C D) (fig. 2.2.c) .

4) La lista vacia corresponde a sólo tener el átomo NIL.

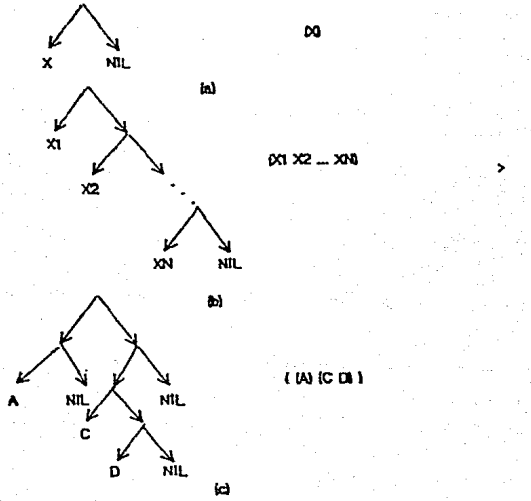


Fig. 2.2

Los árboles binarios es otra estructura que se puede representar mediante la notación punto de expresiones simbólicas. Pudiera pensarse que la estructura de listas es idéntica a la estructura de árbol binario, pero no es así ya que una estructura de árbol binario tiene la forma que presenta la figura 2.3:

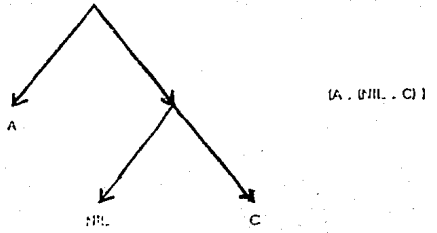


Fig. 2.3

lo cual en una lista no se puede presentar. Dado este caso se puede establecer una correspondencia entre árbol binario y lista en el cual:

Una expresión-S corresponde a una lista si y sólo si el árbol binario para la expresión-S tiene un NIL para cada rama terminal derecha.

2.2 FUNCIONES BASICAS

En esta sección se establece la definición de las funciones básicas, para eso se utilizará el termino de variable que de ninguna manera es el termino tradicional que se conoce en los lenguajes convencionales, donde un identificador es asociado a una localidad de memoria la cual puede tener diversos valores durante la ejecución de un programa, en Lispkit la variable es un apuntador a una expresión simbólica donde el apuntador estará asociado a un átomo simbólico además el valor que toma este apuntador no cambiará durante la aplicación de la función.

En cada lenguaje de programación se tiene un conjunto de operaciones básicas o primitivas las cuales son aplicadas a datos. En un lenguaje convencional tal como Fortran o Algol sus operaciones primitivas son los operadores algebraicos + , - , * , / , etc.; que son aplicados a datos como son los valores numéricos para producir otros datos que tambien son valores

numéricos. Otras operaciones básicas se consideran los operadores lógicos tal como .EQ. , .LT. , .LE. etc que son aplicados para comparar datos entre sí. En Lispkit tenemos cinco funciones básicas con las cuales se pueden manejar expresiones-S, estas son:

CAR(X):

Esta función es aplicada a una lista dando como resultado el primer elemento de esa lista. El resultado puede ser un átomo o una sublista. Es indefinido cuando la función es aplicada a un solo elemento o un átomo, entregando como resultado a NIL. Algunos ejemplos de esta función se muestrán en la siguiente tabla:

X	CAR(X)
(A B C D E F)	A
(2 3 4 5 6 7)	2
((A B) (X Y) (W Z))	(A B)
((M N))	(M N)
(A)	NIL

CDR(X):

Esta función es aplicada a una lista dando como resultado esa misma lista solo que sin el primer elemento. Al igual que la función de CAR, es indefinida cuando la lista es un solo elemento o un átomo (entregando como resultado a NIL). Ejemplos de esta función se muestran en la sig. tabla:

X	CDR(X)
(A B C D E F)	(B C D E F)
(2 3 4 5 6 7)	(3 4 5 6 7)
((A B) (X Y) (W Z))	((X Y) (W Z))
((M N))	NIL
(A)	NIL

CONS(X Y):

Esta función toma dos listas como argumentos y da como resultado la unión de esas dos listas a través de un registro punto. Generalmente el segundo parámetro de CONS es una lista y el primer parámetro es un elemento,por lo tanto si tenemos la lista Y de longitud "N" entonces CONS(X Y) es una lista de N+1. Aquí si podemos tener una lista vacia la cual sería NIL y cuya longitud es cero, la siguiente tabla nos da una idea de como funciona esta primitiva:

X	Y	CONS(X Y)
A	(B C D E F)	(A B C D E F)
1	(2 3 4 5 6)	(1 2 3 4 5 6)
(A B)	((X Y) (Z W))	((A B) (X Y) (Z W))

(A B)	NIL	((A B))
A	NIL	(A)

ATOM(X):

Por medio de esta función podemos determinar la estructura de una expresión-S, es decir, si la expresión-S es un átomo o una lista. El resultado de aplicar esta función es el valor de cierto o falso que esta dado por los átomos de T o F. Como se puede ver la función es un predicado y solo será T cuando la expresión es un átomo simbólico o numérico. Algunos ejemplos estan dados en la sig.tabla:

X	ATOM(X)
(A B C D E F)	F
(2 3 4 5 6 7)	F
((A B) (X Y) (Z W))	F
((A B))	F
A	T
10	T
NIL	T

Notese que la función de ATOM no distingue entre un átomo simbólico y un numérico.

EQ(X Y):

Esta función determina si dos átomos son iguales. Lo mismo que la anterior es una función predicado y regresa el atomo T si es cierto y F si es falso. Si ambos parámetros son una lista el resultado es indefinido, de otra manera será F cuando los átomos son diferentes y T cuando son iguales. La tabla nos muestra algunos ejemplos:

X	Y	EQ(X Y)
A	A	T
A	Z	F
10	10	T
10	A	F
((A B))	A	F
A	((A))	F

A partir de estas funciones básicas se podría construir un tipo de funciones selectoras. Este tipo de funciones permiten accesar cualquier elemento de una lista, así con las dos primeras funciones se puede accesar cualquier elemento de una lista por ejemplo si queremos accesar el segundo y tercer elemento de la lista X:

```
car (cdr (x))
car (cdr (cdr (x)))
```

A estas expresiones se pueden definir como funciones:

```
segelem(x) = car (cdr (x))
terelem(X) = car (cdr (cdr (x)))
```

Con la función de CONS se puede reconstruir una lista, por ejemplo supongamos que tenemos la lista X:

```
X = (A B C D E F)
```

y las siguientes funciones:

```
y = car(x) = A
z = cdr(X) = (B C D E F)
```

entonces

```
W = CONS(Y Z) = (A B C D E F) = X
```

Es importante notar que mediante estas tres funciones básicas se puede construir estructuras de datos y descomponerlas. A estas funciones tambien se les conoce como estructurales. Hasta el momento se han visto ejemplos simples posteriormente se verá otras aplicaciones con estas funciones básicas.

2.3 OTRAS FUNCIONES.

2.3.1 FUNCIONES ARITMETICAS.

Las funciones aritméticas también se consideran funciones básicas al igual que en un lenguaje convencional y también solo son aplicables a valores numéricos (átomos numéricos). Los operadores aritméticos que se considerán dentro de Lispkit son los siguientes:

suma	(ADD)
resta	(SUB)
multiplicación	(MUL)
división	(DIV)
residuo	(REM)

No es necesario indicar cual es el resultado de aplicar estas funciones, ya que es por todos conocido, si acaso la función de residuo que está definida como:

```
x rem y = x - (x / y) * y
```

donde "x" y "y" son átomos numéricos.

Un ejemplo práctico con estos operadores es la función de sumprod:

```
sumprod (x,y) = twolist (x + y , x * y)
```

donde twolist es definido como:

twolist (x,y) = cons (x, cons (y,NIL))

En el apartado de recursividad se verá este tipo de funciones en el que se establecen operaciones aritméticas en listas y donde un factor importante es es el concepto de recursividad.

2.3.2 FUNCION CONDICIONAL

Las expresiones condicionales, dentro del lenguaje son el mecanismo las cuales permitén tomar una dirección dentro del programa a partir de un predicado. La función condicional tiene la siguiente forma:

if e1 then e2 else e3

donde e1 viene siendo la función predicado que da el valor de cierto (T) o falso (NIL) , dependiendo del valor que tome considera la expresión e2 o e3. Un ejemplo de esta función es el siguiente:

distancia (x,y) = if x < = y then y-x else x-y

cuyo resultado es la magnitud de la diferencia entre "x" y "y". Notese que el predicado utilizado es un predicado numérico "< =" (LEQ) que junto con los dos anteriormente mencionados (EQ y ATOM) forman el conjunto de funciones predicado dentro de Lispkit.

2.3.3 FUNCION QUOT

En algunos casos los argumentos de la función pueden ser valores constantes o bien variables, para determinar el valor de la función primero se evalúan los argumentos de la función y después se aplica para esos valores. Por lo tanto es necesario diferenciar entre una constante y una variable (recordar que es un átomo simbólico el cual apunta a una expresión-S).

Supongase que se quiere generar la expresión

(A.B)

para lo cual se requiere utilizar la función CONS, si en principio se considera CONS(A,B), no producirá la expresión (A.B) dado que los simbolos A y B se interpretarian como variables. Para poder diferenciar entre una y otra, a la constante se le antepone la palabra de QUOTE por lo tanto para generar la expresión (A.B) se haría de la siguiente manera

CONS((QUOTE A) (QUOTE B))

2.4 TECNICAS A UTILIZAR EN LA PROGRAMACION FUNCIONAL.

Como en cualquier actividad en la que se requiere la solución

de un problema , siempre se tendrá asociado una serie de técnicas o herramientas las cuales permitirán llegar a una solución óptima. En Lispkit también se tienen una serie de técnicas de programación para cumplir con una funcionalidad característica fundamental dentro del lenguaje.

2.4.1 RECURSIVIDAD DENTRO DEL LENGUAJE.

En este apartado se explorará una herramienta importante dentro de Lispkit, la recursión. Dado la estructura de datos que se tiene dentro de las funciones se puede establecer un mecanismo recursivo el cual viene a solucionar muchos problemas.

2.4.2 DEFINICION DE FUNCIONES RECURSIVAS.

Mediante expresiones predicados (tal como las funciones de ATOM,EQ,LEQ) y expresiones condicionales (como la función IF) se pueden definir funciones a través de procedimientos en la que la función definida pueda ocurrir varias veces. Por ejemplo, si se describe una de las funciones más representativas en recursividad.

$$n! = (n=0 \text{ ---> } 1 \ , \ T \text{ ---> } n*(n-1)! \ )$$

Cuando usamos este procedimiento para evaluar 0! se obtiene la respuesta de 1 , porque es la forma en la cual el valor de una expresión condicional fué definida. La expresión menos significativa sería 0*(0-1)! lo cual no ocurriría . La evaluación de 2! de acuerdo a esta definición procede como a continuación se describe:

$$\begin{aligned} 2! &= (2=0 \text{ ---> } 1 \ , \ T \text{ ---> } 2*(2-1)! \ ) \\ &= 2*1! \\ &= 2* (1=0 \text{ --->} 1 \ , \ T \text{ ---> } 1*(1-1)! \ ) \\ &= 2*1*0! \\ &= 2*1* (0=0 \text{ --->} 1 \ , \ T \text{ ---> } 0*(0-1)! \ ) \\ &= 2*1*1 \\ &= 2 \end{aligned}$$

Otro ejemplo sería el del algoritmo de Newton para obtener una aproximación de una raíz cuadrada del número "a", iniciando con un valor "x" y requiriendo una aproximación aceptable "y" que satisfaga :

$$| \ y2 - a \ | < E$$

asi se tendría:

$$sqrt(a,x,E) = (| \ x2 - a \ | < E \text{ ---> } x \ , \ T \text{ ---> } sqrt(a \ , 1/2(x + a/x), \ E)$$

La definición recursiva simultánea de varias funciones es posible también y se usarán dichas definiciones si ellas son requeridas. No se garantiza que la computación determinada para una definición recursiva siempre termine, por ejemplo, un

intento de calcular n! sólo tendrá exito cuando el valor de "n" sea un entero positivo. Si el calculo no terminará la función sería indefinida para los argumentos dados. Hasta el momento sólo se ha visto ejemplos de funciones recursivas para aplicaciones numéricas, pero para Lispkit que es un lenguaje totalmente enfocado a procesos no numéricos tiene mayor interes las funciones recursivas para expresiones simbólicas.Generalmente las definiciones que se tienen en funciones recursivas tienen como parámetros listas de átomos, donde el último elemento es el átomo NIL, por lo tanto cuando se tenga una lista "x" se considerán dos casos separadamente, x= NIL y x≠ NIL. Para tener una mejor idea de este concepto a continuación se presentará dos ejemplos de funciones que son importantes en cualquier subconjunto de Lisp:

APPEND(X,Y)

Esta es una función que toma dos argumentos (ambas listas) y produce como resultado un sola lista la cual tiene todos los elementos de X seguidos por los elementos de Y (ver la tabla siguiente). Esta función podría considerarse como una función básica por el uso que se tiene en el procesamiento de listas, de hecho en algunos subconjuntos de Lisp original está predefinida como parte del lenguaje.

x	y	append(x,y)
(AB CD EF)	(GH IJ)	(AB CD EF GH IJ)
(A B C)	(D E F)	(A B C D E F)
NIL	(PERRO GATO AVE)	(PERRO GATO AVE)
(X Y)	NIL	(X Y)
NIL	NIL	NIL

La función queda definida como:

```
append(x,y) = if eq(x,NIL) then
                 if eq(y,NIL) then NIL else y else
              if eq(y,NIL) then x else
                 cons( car(x) , append(cdr(x) , y) )
```

esta función se puede presentar en una forma mas corta ya que no es necesario que se pruebe si "y" es NIL o no. Entonces la función quedaría de la siguiente manera:

```
append(x,y) = if eq(x,NIL) then y else
                 cons( car(x) , append(cdr(x) , y) )
```

SUM(X)

Esta función toma como argumento una lista de números enteros y regresa como resultado su suma. El último elemento de la lista es NIL y en este caso se considerá como cero, en la siguiente tabla se presentán algunos ejemplos con esta función:

x	sum(x)
(1 2 3 4)	10
(8 -2 4)	10
NIL	0

La función queda definida de la siguiente forma:

```
sum(x) = if eq(x,NIL) then 0 else
           car(x) + sum(cdr(x))
```

es trivial cuando se tiene x=NIL, pero cuando es diferente se determina la suma con el primer elemento de la lista car(x) más la suma del resto, es decir la función se vuelve aplicar a la lista menos un elemento.

2.1.3 FUNCIONES DE ALTO NIVEL

Debido a la similitud que a veces existe entre muchas funciones en cuanto a los parámetros que maneja y las operaciones que con ellos realiza, se pueden definir funciones de alto nivel en las cuales alguno de sus parámetros es una función que tiene la característica de parecerse en cuanto a su estructura a la función de alto nivel que la contempla. A este tipo de funciones que tienen como parámetro otra función se les conoce como funciones de alto nivel o de propósito general . Considere las siguientes funciones:

```
sum2alis(x) = if eq (x,NIL) then NIL else
                 cons( car(x)+2 , sum2alist( cdr(x) ))

res1alist(x) = if eq(x,NIL) then NIL else
                  cons( car(x)-1 , res1alist( cdr(x) ))

mul2alist(x) = if eq(x,NIL) then NIL else
                  cons( car(x)*2 ,mul2alist( cdr(x) ))
```

como se podrá apreciar ,esas funciones realizán una operación aritmética sobre la lista "x" ; en el primer ejemplo se incrementa la lista por 2, en el segundo se le resta un uno, y finalmente se multiplica la lista por dos en el tercero. Ahora, notese la siguiente función:

```
map (x,f) = if eq(x,NIL) then NIL else
               cons( f( car(x) ) , map( cdr(x) , f ) )
```

como se ve esta función tiene la misma forma que las funciones anteriores, solo que la operación aplicada a cada elemento de la lista es definido por el parametro "f".

2.1.4 EXPRESIONES LAMBDA.

Cuando se estan definiendo funciones de alto nivel, surge el problema de que se tiene que definir y dar nombre a la función que esta dada como parámetro. En uno de los ejemplos

anteriores, se define:

A2L = map (x, sum2alist)

donde x es la lista y "sum2alis" es una función. Se tiene el problema de que "sum2alis" no está definida globalmente. En Lispkit la solución se tiene mediante las expresiones Lambda. Es usual en matemáticas usar la palabra función en forma imprecisa y aplicarselo a las formas tales como y2 + x. A. Church estableció una distinción entre lo que es una forma y una función*.

Considere que se tiene la expresión f=y2+x que se establece como una función para dos variables enteras. Obviamente tendría sentido escribir f(3,4) y que el valor de esta expresión quedará determinado. Si no se tiene una notación adecuada el requerimiento de f(3,4) tendría un valor incierto ya que su valor podría ser 13 o 19 debido a que no existe una correspondencia definida. A. Church llamó a una expresión como y2+x una forma y esa forma puede ser convertida a una función solo si se puede determinar la correspondencia entre las variables que ocurren en la forma y la lista ordenada de argumentos de la función deseada.

En Lisp las expresiones Lambda quedán definidas de la siguiente manera:

λ(x1,...,xn) e

donde x1...xn son los parámetros asociados a la forma "e". Así cuando se calcula el valor de una función los argumentos dados son asociados con el orden de los parámetros x1...xn.

Así se tiene que:

λ(x , y) y2 + x

es la función con dos parámetros (x,y) y si calculamos para los argumentos (3,4) tenemos:

λ ((x , y) y2 + x) (3, 4) = 19

Los parámetros que ocurren en la lista x1...xn se les conoce como parámetros locales, esto quiere decir que solo pueden ser utilizados dentro de la expresión "e". Algunas veces se usarán expresiones en las cuales alguna de las variables dentro del cuerpo de la expresión no sean locales a estas se les conoce como variables globales.

* The Calculi of Lambda Conversion.

Una expresión Lambda es una expresión cuyo valor es una función. También se puede decir que una función es asociada al nombre de la función; así se tiene que en el ejemplo inicial:

```
λ (z) z + 2
```

es asociado a el nombre de sum2list utilizando como parámetro en la función A2L la cual se asocia a la función de map:

```
A2L (x) = map (x , λ(z) z + 2 )
```

Para evitar confusiones y poca legibilidad en la construcción de funciones de alto nivel cuando se tienen funciones como parámetros , Lispkit tiene integrado los bloques locales de LET o WHERE lo cual permitirá hacer definiciones locales de funciones; así se tiene en el ejemplo anterior:

```
A2L(x) = {map(x , sum2list)
           where sum2list = λ(z) z + 2 }
```

Las ventajas que se tienen con las expresiones Lambda son las siguientes:

i) Principalmente, nos permite definir funciones dentro de cualquier parte del programa sin contemplar excesivas reglas semánticas, esto nos permitirá una programación simple y poderosa.

ii) Se tiene la propiedad de cambiar el nombre de los parámetros sin que el valor de la función se altere por ejemplo:

```
λ( x , y ) y2 + x
λ( u , v ) v2 + u
λ( y , x ) x2 + y
```

denotán la misma función, por lo tanto se pueden establecer variables locales aunque se tengan variables globales con el mismo parámetro.

iii) Se mantiene la funcionalidad pura característica importante que se contemplo inicialmente en Lisp y que algunos dialectos de éste han ido perdiendo al implantarse en una máquina convencional.

CAPITULO 3

LA MAQUINA SECD

El propósito de este capítulo es explicar la arquitectura y las instrucciones de la máquina que se simuló para ejecutar en ella un programa realizado en LISPKIT

3.1 LA ARQUITECTURA DE LA MAQUINA SECD

Dado que la estructura del lenguaje funcional LISPKIT es muy diferente a la de los lenguajes convencionales, ya que ni se usan variables (entendiendose como variable aquella que toma varios valores en un programa), ni se manejan direcciones, ni hay saltos, la arquitectura de la máquina en la cual se ejecuta LISPKIT es muy diferente a la que utiliza una máquina VN. De hecho, es una variación de la máquina que LANDIN diseñó para la evaluación de programas escritos en LISP. La arquitectura fue diseñada para que evaluara programas y datos que ingresen como EXP-S y recibe el nombre de máquina SECD debido a las iniciales de sus 4 registros (Stack, Environment, Code y Dump).

Esta máquina procesa el código objeto generado por el compilador LISPKIT (que se verá en el capítulo 4), este código viene representado en forma de EXP-S y puede ser reducido a un conjunto de EXP-S mínimas, las cuales se dividen a su vez en átomos, los que serán una instrucción de máquina o un operando (lo que hace muy sencillo el diseño de la máquina).

Para la ejecución de un programa se alimentan a la máquina SECD el código objeto de una función y los argumentos de dicha función ambos como EXP-S, la máquina dará una EXP-S que será el resultado de aplicar la función compilada a los argumentos dados. Como se muestra en la figura 3.1

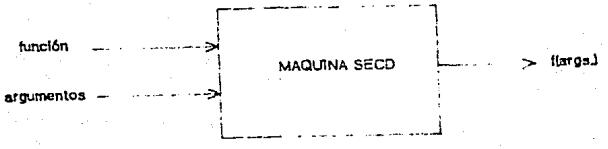


figura 3.1

Todos los datos que se almacenan en los 4 registros son EXP-S (en

realidad el registro lo que contiene es una apuntador a el área de memoria donde se halla la EXP-S). Los Estados de la máquina SECD quedan definidos con el contenido de los 4 registros, cuando hay un cambio en alguno de ellos se da una TRANSICION de MAQUINA y para representarla se usará la siguiente notación:

s e c d -----> s' e' c' d'

La cual nos muestra el estado anterior y el nuevo, si algún registro no sufrió cambios se pondrá la misma letra.

Veamos ahora las funciones específicas de cada uno de los 4 registros

Registro de Control (c)

Al iniciarse la ejecución se carga en éste registro el código objeto del programa que se está ejecutando.

Registro de Stack(s)

Este registro se usa exclusivamente para conservar los valores intermedios generados mientra se evalúa una expresión. Y al terminar de evaluarse la expresión, el resultado quedará en el tope de dicho stack.

Registro de Medio Ambiente (e)

En este registro se cargan los valores que se asociaran a las variables de la función en el curso de la ejecución

Registro de Resguardo (Dump) (d)

Este registro se usa para salvar los valores de los 3 registros anteriores cuando se llama una nueva función.

La forma de operar de la máquina SECD es la siguiente: al iniciarse el proceso se carga el código objeto de la función a ejecutarse en el registro de control, dicho código tiene la particularidad de que siempre en su primera posición tendrá un átomo numérico (aunque en la realidad se pondran mnemonicos) que es el código de una instrucción de la máquina y en las siguientes posiciones vendrán los operandos de la instrucción (si es que tiene). La ejecución de la instrucción determinará la transición de máquina que se presentará, pero al terminar dicha transición, se deberá encontrar en la primera posición del registro otro código de instrucción, de tal forma que la ejecución de un programa se traduce en un conjunto de transiciones de estado que terminan al encontrarse la instrucción STOP. Es de notarse que hay instrucciones que cambian el contenido del registro de control al llamar a una función, ya que dicha subrutina es cargada a el registro.

3.2 INSTRUCCIONES DE LA MAQUINA SECD

Para llevar a cabo sus funciones la máquina SECD dispone de un conjunto de instrucciones, las que se pueden agrupar en base a su función en los siguientes grupos:

```
-manejo de constantes y valores de las variables
        LDC                 LD
-operaciones aritméticas y lógicas
        ADD                 SUB
        MUL                 DIV
        REM                 EQ
        LEQ
-funciones primitivas
        CAR                 CDR
        ATOM                CONS
-formas condicionales
        SEL                 JOIN
-llamadas de función
        LDF                 AP
        RTN
-funciones recursivas
        RAP                 DUM
-terminación de proceso
        STOP
```

3.2.1 MANEJO DE CONSTANTES Y VALORES DE VARIABLES

Para poder manejar constantes y asignar valores a las variables de una función, la máquina SECD usa dos instrucciones que se explican a continuación:

INSTRUCCION LDC

La instrucción LDC se encarga de colocar una constante en el stack y su código en el registro de control sería:

(LDC x.c)

Donde hay tres partes, el mnemónico LDC, el operando de LDC que es la constante a cargarse en el STACK y c que indica el resto de la lista de control, de ahí el uso del punto. Entonces lo que hace LDC es cargar la constante x en el STACK obteniendose la siguiente transición:

s e (LDC x.c) d ----> (x.s) e c d

Donde vemos que despues de ejecutarse, en control solo queda el resto de lista (o sea c) y el operando x fue colocado en el stack, lo que se indica por (x.s).

Un ejemplo donde solo intervienen los registros de control y el stack es:

Sea el siguiente programa:

if(1-2*3=4) then true

```
                    else false
```

Que expresado en lenguaje fuente de LISPKIT tendrá una forma como la siguiente:

```
(EQ(
     SUB (QUOT 1) (MUL (QUOT 2) (QUOT 3)))
     (QUOT 4)
   )
   (QUOT TRUE)
   (QUOT FALSE)
)
```

El código objeto que se generaría al compilar este programa se carga al registro de control (si hubieran argumentos se cargaban al registro de environment) y la secuencia resultante sería:

```
s        e        (LDC 1 LDC 2 LDC 3 MUL SUB LDC 4 EQ)      d
;        Se carga el uno al stack
(1.s)    e        ( LDC 2 LDC 3 MUL SUB LDC 4 EQ)           d
;        Ahora se carga el 2
(2 1.s)  e        (  LDC 3 MUL SUB LDC 4 EQ)                d
;        Y ahora el 3
(3 2 1.s)         e        (  MUL SUB LDC 4 EQ)      d
```

La parte final de la ejecución se verá cuando se expliquen las operaciones aritméticas.

En este ejemplo se ve claramente como operan los registros de stack y control. El registro de control contiene el programa en código objeto, y el primer átomo de él es siempre una instrucción de máquina. La transición de estado que se realizará está determinada unicamente por esta instrucción. El stack es usado para almacenar los resultados parciales durante la evaluación de la expresión.

INSTRUCCION LD

La instrucción LD asigna valores a las variables del programa, y para esto hace uso del registro de environment(E), en este registro se cargan los valores que se asignarán a las variables en el curso de la ejecución y tiene una estructura de lista de listas (ya que es una EXP-S), por ejemplo :

```
                    (( 3 17 )  ( (a b) (c d)))
```

Sería un caso típico de el contenido de e, estos son los valores que le serán asignados a las variables de la función que se va a generar. Dichos valores serán accesados por medio de la instrucción LD, la cual tiene como operandos un par de enteros llamados "PAR INDICE".

Las sublistas de E son numeradas a partir de cero y en forma ascendente (0, 1, 2, 3...) y cada elemento de cada sublista esta también numerada en la misma forma. El "PAR INDICE" (i, j)

selecciona el jesimo miembro de la iesima sublista. Así los argumentos anteriores pueden ser indexados por (0 0) (0 1) (1 0) (1 1) para seleccionar cualquiera de sus valores. Por ejemplo:

PAR INDICE	valor
(0 0)	3
(0 1)	17
(1 0)	(a b)
(1 1)	(c d)

Veamos un ejemplo con transición de estado

s	e	c	d
nil	((3 7) (2 0))	(LD (0 1) LD(1 0) ADD)	d
(7)	((3 7) (2 0))	(LD(1 0) ADD)	d
(2 7)	((3 7) (2 0))	(ADD)	d
(9)	((3 7) (2 0))	NIL	d

El programa anterior obtiene dos valores del medio ambiente y los suma.

En un programa cada variable será asociada con una localidad en el registro E (medio ambiente). De hecho la sublista con el numero 0 corresponderá a las declaraciones en el bloque o función que utilicen primero a la variable, la sublista número 1 lo será a las declaraciones del bloque que utilice la función anterior y así sucesivamente.

3.2.2 OPERACIONES ARITMETICAS Y PREDICADOS

La máquina SECD provee de un conjunto de operaciones aritméticas y predicados, las cuales operan en forma casi identica, siendo unicamente la diferencia la operación que se realiza, por lo que solo se verá en detalle para una de ellas
La transacción para ADD es:

(a b.s) e (ADD. c) d ----> (b+a.s) e c d

Donde (a b.s) indica que el stack tiene al menos dos miembros a y b y la forma (b+a.s) indica que los miembros seran reemplazados por su suma. en los casos de EQ y LEQ, el valor resultante en el tope del stack será alguno de los átomos T o F para indicar verdadero o falso dependiendo del resultado del predicado.

Las transiciones de las restantes operaciones será:

(a b.s) e (SUB.C) d ----> (b-a.s) e c d
(a b.s) e (MUL.C) d ----> (b*a.s) e c d
(a b.s) e (DIV.C) d ----> (b/a.s) e c d
(a b.s) e (REM.C) d ----> (b REM a.s) e c d
(a b.s) e (EQ.C) d ----> (b=a.s) e c d
(a b.s) e (LEQ.C) d ----> (b=<a.s) e c d

Un ejemplo de los estados que se suceden cuando se ejecuta un programa se muestra a continuación.

Sea el programa que se mostro al explicar la instrucción LD y acabemos de ejecutarlo, se había llegado a que se colocaban en el stack 3 constantes

```
(3 2 1.s)     e     (  MUL SUB LDC 1 EQ)      d
;         Ahora se multiplica
(6 1.s)       e     (   SUB LDC 1 EQ)         d
;         Y se ejecuta la resta
(-5.s)        e           (LDC 1 EQ)          d
;         Se carga 1
(1 -5.s)      e           (EQ)                d
;         Se compara
(F.s)         e       NIL                           d
```

Notese que al final queda una "F" en el tope del stack, ya que el resultado del predicado es falso.

3.2.3 FUNCIONES PRIMITIVAS

En el capítulo 2 se vieron las operaciones primitivas sobre EXP-S, y fueron CAR, CDR, CONS y ATOM. En la máquina hay 4 instrucciones que realizan lo mismo y con el mismo nombre, todas ellas operan con el elemento que se halle en el tope del stack y asimismo dejan su resultado final en el mismo tope.
Las transiciones de estado que producen son:

```
(a b.s)          e        (CONS .c)   d ----->  ((a.b).s) e c d
((a.b).s)        e        (CAR. c)    d ----->  (a.s)      e c d
((a.b).s)        e        (CDR. c)    d ----->   (b.s)     e c d
(a.s)            e        (ATOM. c)   d ----->   (T .s)    e c d
```

La máquina SECD que implementamos fue diseñada especificamente para LISPKIT por lo tanto, las instrucciones remanentes son específicas para el manejo de programas realizados con LISPKIT.

3.2.4 FORMAS CONDICIONALES

Hay dos instrucciones que permiten el uso de formas condicionales, una es SEL, la cual selecciona una sublista de dos posibles en control dependiendo del valor que halle en el tope del stack (que podrá tener dos valores, verdadero o falso), y JOIN que sirve para retornar el control principal a la lista que efectuo la selección.
La forma exacta de operar de SEL, es la siguiente: primero checa el valor que se halla en el tope del stack, y en base a el elige la sublista a ejecutarse. El remanente del control es salvado en el registro DUMP, para que al terminar de evaluarse la sublista, (al hallarse un JOIN) se regresa dicho control a su lugar, de tal forma de que la computación pueda continuar.
Las transacción de estado para SEL es :

```
(x.s)    e  (SEL sub1 sub2.c)  d---> s    e subn(x) (c.d)
```

En la transición se observa que SEL elige la sublista 1 o 2 de acuerdo a el valor de x (T o F) y la deja en el registro de control, además salva en el registro DUMP el resto de la lista principal (c) para que al terminar de ejecutarse la sublista elegida se pueda recuperar la lista principal c.
La transición de JOIN es:

s e (JOIN) (c.d) -----> s e c d

JOIN lo único que hace es recuperar la lista c del registro DUMP, para que se siga ejecutando, por lo tanto toda sublista que sea pasada a el registro de control deberá tener un JOIN al final.

3.2.5 LLAMADAS DE FUNCION.

Dos son las instrucciones necesarias para la evaluación de las llamadas a una función, además estas sirven también cuando se compilan bloques LET

La instrucción AP (aplica) es muy importante, ya que es la forma principal en la cual los valores son establecidos en el registro E se verá que cuando una función definida por el usuario es aplicada a sus argumentos, los valores de esos argumentos son colocados en el registro E como una sublista de tal manera que ellos pueden ser accesados por el código de la función utilizando instrucciones LD

Antes de continuar veamos cual es la representación al tiempo de ejecución del valor de una función. Esta está representado por la unión de los valores de los argumentos de la función (esto es, el contexto en el cual se va a evaluar esta), y la lista de control que se halla en el registro C. Esta unión es creada por la instrucción LDF, la cual tiene a la lista de control como operando. Su transición es:

s e (LDF c'.c) d ----> ((c'.e).s) e c d

Así la unión (c'.e) es simplemente un CONS de el operando de LDF (que es la lista de control c') y el contenido de E. Por ejemplo, si la máquina esta en el estado

s	e	c	d
(O)	((3 7) (a))	(LDF(LD(1.1) RTN) LD(O.1))	nil

Al aplicar LDF el estado sig. será.

s	e	c	d
(((LD (1.1)RTN).((3 7)(a)))O)	((3 7)(a))	(LD (O.1))	nil

Donde la unión será ((LD (1.1) RTN). ((3 7)(a))), por lo general, cuando se crea una unión no se aplica inmediatamente, pero llegará el momento en que se cargue en E y será llamado un x

número de veces, usando una instrucción LD de tal forma que pueda ser aplicado a diferentes argumentos.

Cuando la función es llamada y aplicada usando una instrucción AP, debe aparecer en el stack inmediatamente arriba de la lista de parámetros actuales. Entonces la instrucción AP instalará aquellos en el registro E y pondra el código de la unión en el registro de control para ser ejecutado. El registro DUMP se usa para salvar los registros existentes tal que, puedan ser reataurados por una instrucción RTN que se halla al final del código de la unión.

La instrucción AP tiene la siguiente transición:

((c'.e')v.s) e (AP. c) d ---> nil (v.e')c' (s e c.d)

De donde se ve que AP espera que en el tope del stack se halle la unión (c'.e') y el segundo elemento de la lista sea una lista "v" de valores para los parámetros de la función representada por la unión. La evaluación de la función se inicia instalando el código c' en control y construyendo el medio ambiente (v.e'). Esta forma del medio ambiente significa que los parametros de la función serán accesados en las posiciones (0.0), (0.1), (0.2), etc y que las variables libres son accesadas en las posiciones (1.0), (1.1), ..., (2.0), (2.1), (2.2)...etc. para la ejecución del código unión empezamos con el stack vacio. El contenido del DUMP inmediatamente despues del AP es (s e c d) mostrando que el estado completo antes de la ejecución ha sido salvado. Ejemplo:

(((LD (1.1)LD(0.0) ADD RTN).((3 7)(a)))(6)0) ((2 b)) (AP STOP) d

al aplicar AP queda:

NIL ((6)(3 7)(a)) (LD(1.1)LD(0.0)ADD RTN) ((0)((2 b)) (STOP).D)

Viendo el código que ha sido instalado en el registro de control, vemos que accesa la posición (0.0) hallando un 6 (que es el primero y único parámetro actual) y en la posición (1.1) hallando un 7, que es un valor que fue proporcionada por la unión.

La instrucción RTN complementa la instrucción AP, ya que es la que restaura el estado salvado en el DUMP. El código cargado dentro del registro de control por AP deberá entonces terminar con un RTN, cuya transición de estado es:

(x) e' (RTN) (s e c.d) ----> (x.s) e c d

Se ve que RTN espera encontrar un valor x en el stack. Lo regresa a el medio ambiente que llama a la unión, insertandolo en el stack restaurado S . Ejemplo. Después de ejecutar los LD'S y el ADD en el estado anterior llegamos a :

(13) ((6)(3 7)(a)) (RTN) ((0)((2 b)) (STOP).D)

Al cual la transición RTN es aplicable llegando a:

(13 O) ((2 b)) (STOP) d

Y el valor 13 es regresado en el tope del stack como el resultado de la llamada de la función.

3.2.6 FUNCIONES RECURSIVAS

Para la evaluación de bloques recursivos se usan las instrucciones DUM y RAP. Dado que es necesario que las definiciones locales sean evaluadas en un contexto que incluya sus propios valores, esto se realiza creando un medio ambiente (MA) temporal con las definiciones locales pendientes, se evaluan las definiciones usando el MA temporal y despues se reemplazan la parte pendiente del medio ambiente por los valores de las definiciones. Así, el MA temporal incorporará cualquier unión generada cuando se evaluaron las definiciones. Despues de reemplazar las partes pendientes, las uniones seran circulares, cada una conteniendo un MA que contiene a las uniones en si mismas como valores.

La instruccion DUM crea un MA temporal con un valor pendiente Ω como su primera sublista Así cualquier intento de accesar valores en esta sublista será indefinido hasta que Ω haya sido reeplazado. La transición de DUM es

s e (DUM. c) d -----> s (Ω.e) c d

RAP es casi idéntico a AP, excepto que hace un CONS de los valores actuales de los parametros en el MA, usa la función "rplaca(x,y)" (que lo que hace es reemplazar el car de x con y, donde x debe ser un Ω) para reemplazar el Ω puesto por DUM.

((c'.e')v.s) (.e) (RAP.c) d --> NIL rplaca(e',v) c' (s e c.d)

RAP será usada siempre en un estado donde e' = (?.e), así, la unión en el tope del stack contiene un MA idéntico a el MA actual. RAP instala el codigo c' de la unión en el control y el MA e' en el registro E, habiendo reemplazado su car por la lista de valores v, el stack,el MA y el control son salvados en d, en el caso del MA, como fue expandido por DUM, solo su cdr es salvado. Por lo tanto RAP es igual a AP con la diferencia que cuando se ejecuta RAP al MA se le ha agregado Ω.

3.2.7 TERMINACION DE PROCESO

La única forma de terminar un proceso en la máquina SECD es cuando se ejecute la instrucción STOP, por lo que esta será la última de cada función.

CAPITULO 4

SEMANTICA DE LISPKIT.

La descripción formal de la semántica de un lenguaje es hoy en dia, un estado constante de desarrollo teniendo énfasis en la tecnología de definición. Existen métodos de definición semántica, los cuales estan basados bajo una técnica de modelado. Otros han desarrollado máquinas abstractas las cuales tienen un lenguaje de instrucciones que es usado para describir un proceso o un lenguaje.

A la fecha se conocen tres metodologías para la definición semántica de un lenguaje:

i) Semántica Operacional.- La cual está relacionada con las operaciones de una máquina abstracta que lleva una serie de transiciones secuenciales.

ii) Semántica denotacional.- Definición a base de notaciones o símbolos para todos los elementos de un lenguaje de programación en un dominio abstracto independiente de alguna máquina abstracta.

iii) Semántica Axiomática.- Donde la descripción de las propiedades de un lenguaje se especifican en término de los axiomas a los cuales la ejecución de un programa debe conformar.

Lispkit es considerado como semántica operacional, su especificación que posteriormente se presentará, está en una forma precisa lo cual permite que la implantación del lenguaje sea satisfactoria.

El estudio formal de la semantica se dá mediante la máquina abstracta conocida como SECD la cual se puede ver como la siguiente función:

exec (f*,a)

donde "f*" es el programa objeto en Lispkit y "a" representa la lista de argumentos de esa función. Por lo tanto se puede tomar a la función exec como una especificación formal de la sémantica del lenguaje Lispkit dado que determina un valor para cualquier programa escrito en Lispkit.

En este capitulo se analizará el mecanismo de compilación en cada una de las instrucciones del lenguaje (se verá posteriormente que también se les conoce como expresiones bien formadas). Pero antes será necesario establecer algunos conceptos importantes dentro del estudio formal que se llevará a cabo en este capitulo como son las condiciones necesarias

para tener expresiones bien formadas, reglas para transformar de forma abstracta a concreta, asociación de variables a valores,etc.

4.1 REPRESENTACION DE PROGRAMAS.

4.1.1 EXPRESIONES BIEN FORMADAS

Para poder ejecutar un programa funcional de Lispkit en la máquina SECD, es condición necesaria que cada una de las expresiones de ese programa tengan una estructura adecuada, de tal manera que cada una de ellas tenga un valor asociado . A este tipo de expresiones se les conoce como expresiones bien formadas (e.b.f.).

Ejemplos de expresiones bien formadas son las siguientes:

```
x + 1

car (cdr (s) )

if x=0 then 1 else x + f(x-1)
```

en cada uno de los ejemplos, las expresiones tendrán un valor definido cada vez que se les aplique un valor a las variables de "x" y "s". El caso contrario de expresiones mal formadas son los siguientes:

```
x +
car (cdr)
if x=0 then 1
```

aunque se le aplique un valor a la variable, se tendrá una indefinición, se podría ver en el ultimo ejemplo que no esta definido el valor para cuando x es diferente de cero.

Dentro de una expresión bien formada podemos tener varios niveles de subexpresiones bien formadas, bajo la regla de que siempre se compone una expresión bien formada en un operador y en un conjunto de operandos, donde cada operando a su vez puede ser otra expresión bien formada, en los ejemplos anteriormente citados se puede ver que x + 1 tiene como operador el "+" y como operandos a "x" y a "1" y así los siguientes:

```
car(cdr (s))          operador : car
                      operandos: cdr(s)   operador:  cdr
                                          operando:  s

if x=0 then 1 else x + f(x-1)
                      operador   :  if...then...else
                      operandos: x=0          operador:  =
                                              operandos:x,0
                                 1
                                 x+f(x-1)     operador:  +
                                              operandos:x,f(x-1)
```

Gracias a esta filosofía se puede tener un programa o función como una expresiom bien formada donde su operador es la función misma, y sus operandos son los parámetros de la función. Cada expresión biem formada se considera como una expresión simbólica, de esta maniera con expresiones simbólicas se puede construir la forma concreta de cualquier programa en Lispkit.

4.1.2 REGLAS PARA TRANSFORMAR DE FORMA ABSTRACTA A CONCRETA

Una de las pocas desventajas de Lispkit es la dificultad de poder entender un programa en su representación simbólica o también conocida como en forma concreta. Generalmente los programas escritos en un lenguaje simbólico son primero escritos en una forma abstracta como se ha venido realizando a través de todos los ejemplos descritos, esta forma abstracta es facil de entender y a la vez facil de modificar, la cual tiene una estructura semejante a la de Algol. Para transformar a su forma concreta, deberán tenerse en cuenta las siguientes reglas:

i) Debem cumplirse que todas las expresiones sean bien formadas (e.b.f.).

ii) Las mas elementales e.b.f son las constantes y las variables. Las variables serán representadas sencillamente como átomos simbólicos y a las constantes se antepondra la función básica de QUOT, así se podrá distinguir constantes de variables, ejemplo:

	Repres. Abstracta	Repres. Concreta
variables:	x,y,z	X,Y,Z
constantes:	(a,b),nil,127	(QUOTE (A B)) (QUOTE NIL) (QUOTE 127)

iii) En forma general las expresiones simbólicas estan representadas em forma concreta como una lista, siempre entre paréntesis, donde el primer elemento de esa lista, es una palabra clave que regularmente es una función básica o una función definida, por ejemplo:

f(e1, e2,....,ek)

es transformado a:

(F e1 e2 ... ek)

como se aprecia la función llamada, aparece dentro de los parentesis y no afuera. Es importante mencionar que se debe respetar la sintaxis de las e.b.f., tanto en la colocación de los paréntesis como los espacios en blanco. A continuación se presenta una tabla con las funciones o expresiones bien formadas, implantadas em Lispkit, y su representación en forma concreta:

SEMANTICA DE LISPKIT

FORMA ABSTRACTA	FORMA CONCRETA
car(e)	(CAR e)
cdr(e)	(CDR e)
cons(e1,e2)	(CONS e1 e2)
atom(e)	(ATOM e)
eq(e1,e2)	(EQ e1 e2)
e1 + e2	(ADD e1 e2)
e1 - e2	(SUB e1 e2)
e1 * e2	(MUL e1 e2)
e1 / e2	(DIV e1 e2)
e1 rem e2	(REM e1 e2)
e1 <= e2	(LEQ e1 e2)
if e1 then e2 else e3	(IF e1 e2 e3)
>(x1,...,xk) e	(LAMB (x1,...,xk) e)
{let x1=e1 and x2=e2 . . . and xk=ek e}	(LET e (x1 e1) (x2 e2) . . . (xk ek))
{letrec x1=e1 and x2=e2 . . . and xk=ek e}	(LETR e (x1 e1) (x2 e2) . . . (ex ek))
llamada de funcion	(e e1 e2 ... ek)

4.2 ASOCIACION DE VALORES A VARIABLES (BINDING).

Para poder establecer el ambiente en el que se trabaja en la máquina SECD, es necesario considerar la asociación que se tiene entre las variables y sus valores. En un lenguaje convencional, ésta asociación se realiza a través de la asignación, la cual en Lispkit no existe. Se podrá notar que una forma de asociar valores en el lenguaje funcional, es mediante el uso de funciones Lambda, a través de sus

argumentos, pero esta no es la unica forma, ya que podemos tener variables globales dentro de una definición de función, en la cual, se tenga una variable la cual no este definida como parámetro (variables globales) o bien definir una variable local dentro del cuerpo de la función por medio de una expresión LET.

Se establecen ciertas reglas dentro de Lispkit para definir la conexión que existe entre variables y valores, pero antes de explicarlas, es necesario definir el concepto de contexto.

Un contexto no es mas que la asociación entre variables y valores y que se representa a través de tablas, así cada expresión en un programa, es evaluado con respecto a un contexto. Generalmente los valores que se asocian a las variables son, expresiones simbólicas, por ejemplo:

```
x ----> ( A B C )
y ----> ( D E F )
z ----> 10
```

Una vez definido el concepto de contexto, se pueden enumerar las reglas de asociación (binding).

1) Se pueden crear nuevas asociaciones por medio de una expresión LET. De inicio todas las variables tienen un contexto, en el cual tienen definidos sus valores, cuando se tiene un bloque LET, a las variables locales definidas se les asocia un valor en base al contexto mas inmediato, por ejemplo:

Suponga que se tiene el contexto definido en el ejemplo anterior, y que se tiene el bloque LET siguiente:

```
{( let  u  =  cdr(x)
        v  =  car(y)
        w  =  z + 1
   cons( (cons(u  v)) z) }
```

El contexto sería ampliado a lo siguiente:

```
x ----> ( A B C )
y ----> ( D E F )
z ----> 10
u ----> ( B C )
v ----> D
w ----> 11
```

2) Cuando las variables locales tienen el mismo nombre, que las variables del contexto, en el cual el bloque es evaluado, la nueva asociación anula la anterior, por ejemplo:

```
( let  x  =  car(y)
     cons(x  NIL)  )
```

se tiene el siguiente contexto:

```
x ----> D
y ----> ( D E F )
z ----> 10
```

de esta manera se anula el anterior valor de 'x'. Es importante hacer notar, que la anulación, sólo tienen efecto dentro del bloque, una vez que sale de este recuperá su valor anterior.

3) Relacionada también con bloques locales, en la cual el nombre de una variable, aparece tanto en el lado izquierdo de la definción como en el lado derecho. Para que sea valido este tipo de definición local, el nombre de la variable debe haber sido definida en el contexto. Este tipo de definiciones se refleja cuando se asocia un valor numérico a la variable, por ejemplo:

```
{ let z = z+1
        z / 11 }
```

En el contexto se tendría antes de entrar al bloque el valor de 10, al evaluarse el bloque local, la variable tendría el valor de 11, cuando se sale del bloque, el valor de z sería de 10. Cabe decir, que si en un momento, se cambiará el nombre de la variable local, el resultado del bloque no cambiaría, ejemplo:

```
{ let m = z + 1
        m / 11 }
```

el bloque tendría el mismo resultado.

4.3 ASOCIACION EN EL CONTEXTO DE FUNCIONES

Para hacer un análisis de la asociación de variables, dentro de funciones, es necesario establecer dos casos:

1) Cuando no se tiene variables globales dentro del cuerpo de la función. Las funciones siempre son definidas mediante las expresiones Lambda, y generalmente el valor de esa funcion, es asociada a un nombre, por ejemplo:

```
{ let f = λ(x,y) 4 * x + y
        f(2,3) )
```

es una función aritmética, cuyo resultado es asociado a 'f', el cuerpo de la función es evaluado en el contexto de:

```
x ----> 2
y ----> 3
```

dando el valor de 11 que es asociado a 'f'. Como se podrá notar, el valor asignado a 'f' es directo y sin problema alguno pero ¿que pasa si el cuerpo de la expresión Lambda contiene variables globales?.

2)Cuando se tienen variables globales dentro del cuerpo de la función, es necesario primero reemplazar a sus valores las variables globales que se tengan, para así poder evaluar la función. Ahora surge la pregunta ¿bajo que contexto son evaluadas las variables globales?. Supongase el siguiente ejemplo:

```
{ let f = λ(x) 4 * x + y
        (let y = 4
        f(2) )}
```

donde se tiene el contexto, para la variable global de :

```
y ----> 3
```

surge la duda de si, se evalúa primero con y=3 o con y=4. Ordinariamente primero se evalúa la funcion con y=3, para establecer el "medio ambiente" de:

```
f ----> λ(x) 4 * x + 3
y ----> 4
```

entonces, evaluando la función para un argumento de 2, se tendría un valor de 11.

Para terminar con esa incertidumbre de que valor elegir, se creo el concepto de cerradura. La cerradura es una forma compuesta, la cual contiene la expresión Lambda y el contexto, el cual define los valores de las variables globales.

```
( Expresión Lambda , contexto )
```

En el ejemplo anterior se tiene entonces el siguiente contexto:

```
f ----> ( λ(x) 4 * x + y , ( y---->3 ) )
y ----> 4
```

4.4 COMPILADOR LISPKIT

Al igual que la máquina SECD, la cual se puede ver como la función exec(f,a), también el compilador de Lispkit se define como una función escrita en Lispkit. Esta función tiene como parámetro una expresión bien formada "e", la cual es el programa fuente, y que al ser compilado se obtendrá el codigo ejecutable "c" , por lo tanto, se puede establecer:

```
compile(e) = e* = c
```

Considerando esta regla, y sabiendo que el lenguaje se compone de instrucciones o expresiones simbólicas, con la característica de ser expresiones bien estructuradas, se puede compilar cada una de las expresiones del lenguaje por separado. Que se quiere decir con esto, por cada expresión que se compile el código que genera es independiente de la anterior o siguiente expresión, pero es necesario que éstas sean expresiones bien formadas.

Cada expresión que se compile, se obtendrá una lista de instrucciones ejecutables en la máquina SECD, las cuales deben cumplir con la siguiente propiedad:

Si cargamos el codigo compilado de una expresión bien formadas dentro del registro de "CONTROL", y se ejecuta ese código, el valor que se obtiene deberá estar en la parte superior de la "PILA" (STACK), teniendo en consideración que el "MEDIO AMBIENTE" fué cargada con una estructura apropiada, a la posición de los valores de cada variable global de la expresión, además, el "MEDIO AMBIENTE" y el "DUMP" deberán conservar los mismos valores, antes y despues de la ejecución. A esta propiedad que deben tener las expresiones bien formadas, se les conoce como propiedad de efecto completo.

Describiendo esta propiedad en terminos de transición de estados de la SECD (ver capítulo de la SECD) se tiene:

$$s\ e\ c\ d \Longrightarrow (x\ .\ s)\ e\ NIL\ d$$

donde "x" es el valor que se obtiene al ejecutar el código c.

Antes de describir el código que genera cada una de las expresiones bien formadas del lenguaje, es necesario establecer ciertas reglas y notaciones las cuales servirán para definirlas claramente.

Una de esas notaciones se describió anteriormente, y es la que representa la transición de estados de la maquina.

$$s\ e\ c\ d \Longrightarrow s'\ e'\ c'\ d'$$

la parte de la izquierda indica el estado de los registros antes de la ejecución de una instrucción y la parte de la derecha indica el estado después de la ejecución.

Otra notación que se utilizará, y que está relacionada con la función APPEND (función que se vio en el capítulo 2), y que significa la unión de dos o mas listas de códigos, es:

$$c1\ ;\ c2 = append\ (c1,c2)$$

es bueno recordar que la función append es:

```
append(c1,c2)= if x=NIL then y
                 else cons( car(x),append( cdr(x) , y) )
```

un ejemplo de esta notación, se tiene en la siguiente transición de estado, en el cual dos códigos se representan de la siguiente manera:

$$s\ e\ c1\ ;\ c2\ d \Longrightarrow (x1\ .\ s)\ e\ c2\ d$$

del anterior ejemplo se desprende la siguiente regla:

Si se ejecutan dos o mas códigos de expresiones, el valor que se obtiene en esas expresiones es depositado en la "PILA", desapareciendo ese código del registro de "CONTROL", y el resto de los registros permanece sin cambio, por ejemplo:

s e c1¡c2¡c' d ====> (x2 x1.s) e c' d

Ahora, se verá la notación punto (.), que une dos expresiones simbólicas por medio del registro CONS, por ejemplo si se tiene:

(x1.s) indica CONS(x1 , s)
(x1 x2.s) indica CONS(x1 , CONS(x2 , s))

Por último se tiene la siguiente notación:

e * n

donde "e" es una expresión bien formada y "n" es la "LISTA DE NOMBRES". Esto quiere decir, el código que se genera de una expresion "e", es compilada con respecto a la lista de nombres de variables globales "n".

4.5 ESTRUCTURA DE LAS VARIABLES Y ACCESO A CADA UNA

De las formas de expresión (expresiones bien formadas), que se consideran en Lispkit, la mayoria requieren de una estructura de asociación entre las variables y sus valores. En los programas en Lispkit cada subexpresión (expresiones Lambda y bloques locales), deberán ocurrir en un contexto, en el cual cada variable tiene una posición definida en el "MEDIO AMBIENTE", acorde a el campo de las variables de la expresión.

Las variables que son accesadas están arregladas, en forma de listas dentro del "MEDIO AMBIENTE", de tal manera que las variables locales son accesadas primero, antes que las variables globales. En Lispkit se define una estructura de ocurrencia de variables denominada como LISTA DE NOMBRES, éstas tienen una estructura de lista de listas de átomos, donde cada átomo viene siendo el nombre de una variable, por ejemplo:

((A) (B C) (D E F G))

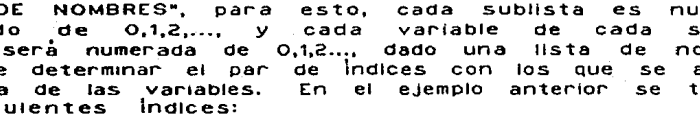
Asi cuando se representa un nuevo bloque local o una función, las variables declaradas se anexarán a la "LISTA DE NOMBRES" como una nueva sublista. En el ejemplo anterior, se tienen tres sublistas, una de ellas con cuatro miembros, otra de dos, y otra de un solo elemento; si un bloque o una función Lambda, aparece con una lista de variables locales (H I) la actual lista de nombres, aparece como:

((H I) (A) (B C) (D E F G))

se verá que la lista más a disposición de acceso dentro del

"MEDIO AMBIENTE" será (H I).

Una de las partes vitales dentro de la definición del compilador es poder accesar cualquier variable dentro de la "LISTA DE NOMBRES", para esto, cada sublista es numerada partiendo de 0,1,2,..., y cada variable de cada sublista también será numerada de 0,1,2..., dado una lista de nombres. Se puede determinar el par de índices con los que se accesan cada una de las variables. En el ejemplo anterior se tendrían los siguientes índices:



Para poder accesar cada uno de los miembros de la "LISTA DE NOMBRES" es necesario obtener el par de índices que identifican cada variable , para lograrlo, se utiliza la función LOCATION(x,n) donde "x" es la variable de búsqueda y "n" es la lista de nombres. Describiendo esta función como hasta ahora se ha venido realizando a través de pseudo-código se tiene:

```
Location (x,n)=
    if member (x,car(n)) then cons(0,position(x,car(n)))
       else
          {cons (car(z)+1 , cdr(z) )
                where z= location (x,cdr(n)))}
```

Como se puede apreciar la función es recursiva, y tiene integrada otras dos funciones "MEMBER" y "POSITION". La función regresará (0.j) si la variable requerida se encuentra en la primera sublista y en la posición "j" , si no ocurre en la primera sublista, entonces "LOCATION" es llamado recursivamente con cdr(n) hasta obtener el par de índices (i , j).

La función POSITION nos da el índice de la variable, dentro de la sublista en la cual se encuentra, en otras palabras POSITION (x,a), nos regresa el índice de "x" dada una lista de átomos "a" (sublista de "n"). Asumiendo lo anterior se tiene:

```
position(x,a) = if eq (x, car(a)) then 0 else
                  1+position(x,cdr(a))
```

Al igual que LOCATION y POSITION la función MEMBER también es una función recursiva, ésta lo que hace es checar si la variable está dentro de una sublista. Regresa un valor lógico de "T" o "F", dependiendo de si encuentre o no la variable. Describiendo esta función:

```
member (x,s) = if eq(s,NIL) then f else
                  if eq(x,car(s)) then T else
                     member(x, cdr(s))
```

"x" es la variable y "s" es la sublista.

Dentro del lenguaje de Lispkit la función de MEMBER es muy utilizada, este tipo de función se puede considerar básica dentro del lenguaje por lo tanto es importante tener este tipo de funciones en una biblioteca.

4.6 COMPILACION DE CADA UNA DE LAS EXPRESIONES DE LISPKIT

A continuación se hará una descripción, de la forma en que se genera el código de cada una de las expresiones bien formadas, que componen el lenguaje, apoyados en las reglas descritas anteriormente. En algunos casos se dará ejemplos, de como es que se compilan las diferentes expresiones con la idea de establecer la propiedad de efecto completo que guardan.

4.6.1 CONSTANTES Y VARIABLES

Las expresiones más simples que se pueden tener dentro de Lispkit, son las variables y las constantes, la compilación de ambas expresiones, genera una sola instrucción. Para el caso de la constante se genera el siguiente código:

$$(QUOTE\ s)\star n = (LDC\ s)$$

donde simplemente se carga el operando de la instruccion LDC dentro del "STACK".

El código para la variable es el siguiente:

$$x \star n = (LD\ i) \quad \text{donde } i=LOCATION(x,n)$$

aqui, el valor de "i" se obtiene de la LISTA DE NOMBRES, a través de la función LOCATION la cual retorna un par de indices.

4.6.2 EXPRESIONES ARITMETICAS

Las reglas de codificación para expresiones aritméticas, son parecidas,y lo unico que cambia es la instrucción que realiza la operación numérica, por lo tanto no es necesario describir cada una de estas expresiones, para el caso,considere la expresión de suma:

$$(ADD\ e1\ e2)$$

compilando primero las expresiones de "e1" y "e2" se obtienen los codigos de "c1" y "c2", así el código generado por la suma es:

$$e1 \star n\ ;\ e2 \star n\ ;\ (ADD) = c1\ ;\ c2\ ;\ (ADD)$$

en la ejecución, primero se calcula el valor de "e1" para ser insertado en la "PILA", posteriormente el valor de "e2" y por

último se interpreta la instrucción de suma, para dejar un solo valor en la "PILA", cumpliendose de tal manera la propiedad de efecto completo. En general, las reglas de las expresiones aritméticas son semejantes:

```
(ADD e1 e2)*n = e1*n ; e2*n ; (ADD)
(SUB e1 e2)*n = e1*n ; e2*n ; (SUB)
(MUL e1 e2)*n = e1*n ; e2*n ; (MUL)
(DIV e1 e2)*n = e1*n ; e2*n ; (DIV)
(REM e1 e2)*n = e1*n ; e2*n ; (REM)
```

A continuacion se presenta un ejemplo de como se evaluarla la siguiente expresión :

```
          expresión                    namelist
          ---------=-----------       --=--
          (ADD (CAR X) (QUOTE 1) ) * ((X Y))
  = (CAR X)*((X Y)) ; (QUOTE 1)*((X Y)) ; (ADD)
  = X*((X Y)) ; (CAR) ; (LDC 1) ; (ADD)
  = (LD (0.0)) ; (CAR) ; (LDC 1) ; (ADD)
  = (LD (0.0) CAR LDC1 ADD)
```

Los operandos son calculados y colocados en la "PILA", antes de que se aplique el operador.

4.6.3 EXPRESIONES PREDICADO.

El código generado para estas expresiones, es semejante al de las expresiones aritméticas, sólo que el valor que dejan en el "STACK", es el valor de "T" o "F" , es decir:

```
(EQ e1 e2) * n = e1*n ; e2*n ; (EQ)
(LEQ e1 e2) * n = e1*n ; e2*n ; (LEQ)
```

4.6.1 EXPRESIONES PRIMITIVAS.

En el caso de las expresiones primitivas, tienen un parámetro, a excepción de CONS que tiene dos. Todas las expresiones estructurales tienen reglas de compilación, semejantes a las aritméticas y predicados.

```
(CAR e)*n = e*n ; (CAR)
(CDR e)*n = e*n ; (CDR)
(ATOM e)*n = e*n ; (ATOM)
```

Para el caso de la expresión de CONS, cambia el orden de compilación de los parámetros como hasta el momento se venia realizando, con las expresiones de dos parámetros, primero se compila el 2do. parámetro y luego el primero; en realidad esto no tiene significancia solo que asi fué definida en la máquina SECD, la regla para CONS es:

```
(CONS e1 e2)*n = e2*n ; e1*n ; (CONS)
```

4.6.5 EXPRESION CONDICIONAL

Para la forma condicional se requiere generar, un código más elaborado, su forma es la siguiente:

(IF e1 e2 e3)

donde "e1" da un valor lógico , si es "T" se evalua la expresión de "e2" , si el valor de "e1" es "F" se evalua "e3" . Cuando se compila cada una de las expresiones que componen la condicional, a "e2" y "e3" se les anexa la instrucción de JOIN, para poder regresar al programa, despues de que se ejecute cualquiera de las expresiones de "e2" o "e3". El código que se obtiene de la condicional tiene la siguiente forma:

$$(IF\ e1\ e2\ e3)\star n = e1\star n\ ;\ (SEL\ e2\star n\ ;\ (JOIN)\ \ e3\star n\ ;\ (JOIN)\ )$$

$$= c1\ ;\ (SEL\ c2\ ;\ (JOIN)\ c3\ ;\ (JOIN))$$

Las sublistas de "c2" y "c3", vienen siendo operandos de la instrucción SEL. Al ejecutarse "c1" el valor de éste queda en el tope de la "PILA". La instruccion SEL selecciona alguna de las dos opciones, dependiendo del valor que tenga "c1", y una vez que se ejecuta el código elegido, la instrucción JOIN regresa el control al programa, sin cambiar el estado de la máquina.

Para ilustrar la expresión condicional considere el siguiente ejemplo:

(MUL K (IF (EQ L M) L (QUOTE 2))) * ((K L M))

en notación abstracta sería algo semejante a:

k * (if l=m then l else 2)

el código que se genera es:

```
( LD(0.0) LD(0.1) LD(0.2) EQ SEL ( LD(0.1) JOIN )
(LDC 2 JOIN) MUL)
```

si se analiza separadamente la forma condicional:

```
LD (0.1) LD(0.2) EQ
     SEL
        (LD (0.1) JOIN)
        (LDC 2 JOIN)
```

se tiene el producto de el valor de K, con el valor que se obtiene de la condicional (L o 2), SEL elige ya sea el valor en la posición (0.1) correspondiente a "L" o el valor de 2, todo dependiendo, de el valor lógico que se tenga en ese momento en la "PILA". Cualquiera de las dos instrucciones, restaura el control a la máquina, para así completar la instrucción de MUL.

1.6.6 EXPRESIONES LAMBDA.

Para poder compilar las expresiones Lambda, se deben tener ciertas consideraciones en cuanto a la "LISTA DE NOMBRES", dado que se introducen las variables locales, que se manejan en el cuerpo de la expresión Lambda.

(LAMB (x1 x2 ...xk) e)

la expresión "e" debe ser compilada con respecto a una nueva "LISTA DE NOMBRES", el cual debe tener la siguiente forma:

((x1 ... xk) . n)

donde "n" es la lista de nombres, conteniendo las variables globales, y las "x's" conforman la sublista de parámetros definida por Lambda. Es importante hacer notar que las variables locales, es decir los nombres de los argumentos, se ubicán en la primera posición de la "LISTA DE NOMBRES", puesto que son las primeras en accesarse. Ahora las variables globales serán referidas por un nuevo par de indices ', donde el primer elemento es incrementado por 1. El código generado por Lambda es:

(LDF c ¦ (RTN)) = (LDF e * ((x1...xk) . n) ¦ (RTN))

esta expresión se forma de la instrucción LDF y sus dos operandos, que es el código del cuerpo de la expresión de Lambda, y la instruccion de RTN. Para redondear la definición de Lambda considere el siguiente ejemplo:

Partiendo de que se tiene una LISTA DE NOMBRES con (Y Z) y el código:

(LAMB (X) (SUB (ADD X Y) Z)) * ((X.Y Z)) =
(LDF ((LD(0.0) LD(1.1) ADD) LD(1.2) SUB) RTN)

El resultado de ejecutar el código para Lambda, es solo poner la cerradura (CLOSURE), conteniendo el código en lo alto de la "PILA".

1.6.7 LLAMADA DE FUNCIONES.

Las llamadas de funciones tienen la siguiente forma:

(e e1 ... ek)

cada uno de sus componentes es compilado modularmente, para así, generar la lista de códigos c,c1,...,ck. Para que se pueda cumplir con la propiedad de efecto completo, es necesario combinar esa lista de códigos, tal que al ejecutarse el codigo de "e" ,se encuentren ya los valores que se obtienen de e1,...,ek, en la forma apropiada dentro de la "PILA". Para lograr esa lista de valores, es necesario que el código esté estructurado de la siguiente manera:

(LDC NIL) ¦ ck ¦ (CONS) ¦ ck-1 ¦ (CONS) ¦ ... ¦ c1 ¦ (CONS)

los valores de c1...ck se calculan en forma invertida, dejando así, los valores en la "PILA" como:

(v1 ... vk)

Dado que el código de c = e * n inserta el procedimiento de la función en lo alto de la "PILA", se tiene finalmente:

(e e1 ... ek) * n = (LDC NIL) ¦ ek * n ¦ (CONS) ¦ ... ¦ e1 * n ¦ (CONS) ¦ e * n ¦ (AP)

Cuando el código es ejecutado, se realizan las siguientes acciones:

i) Se construye la lista de los argumentos actuales (la lista de los valores) en la "PILA".

ii) Inserta el procedimiento de la función (Cerradura) en lo alto de la "PILA".

iii) Se identifica la instrucción de AP(aplica procedimiento), la cual salva el estado de la máquina dentro del "DUMP" excluyendo la lista de valores y el procedimiento de la función.

iv) Finalmente se ejecuta el código del procedimiento, dejando un valor en lo alto de la "PILA". Este código debe tener al final la instrucción de RTN (retorno), la cual restaura el estado de la máquina , que corresponde al código antes de hacer la llamada, con la unica diferencia que ahora existe un valor en lo alto de la "PILA", que es el valor de la función llamada.

Como ejemplo considere la siguiente llamada de INC1, que es una función que incrementa en 1 un valor dado, en este ejemplo el valor es 1, entonces se tiene:

(INC1 (QUOT 1))

la función se define a partir de la expresión Lambda:

(LAMB (X) (ADD X (QUOTE 1)))

la cerradura que representa esta expresión al momento de correr es:

B = ((LD(0.0) LDC 1 ADD RTN) . NIL)

donde :

(LD(0.0) LDC 1 ADD RTN)

es la parte de el código y NIL es la parte del "MEDIO AMBIENTE". (Notese que no tiene variables globales por lo tanto se evalua con respecto al contexto NIL).

Al compilarse la llamada, donde INC1 es la unica variable, se tiene :

```
   (INC1 (QUOTE 4) ) * ( (INC1) )
= (LDC NIL) ¡ (LDC 4) ¡ (CONS) ¡ (LD(0.0)) ¡ (AP)
= (LDC NIL LDC 4 CONS LD(0.0) AP )
```

Para demostrar que se cumple con la propiedad de efecto completo, a continuación, se presenta la ejecución de la llamada de INC1, a través de la secuencia de estados de la máquina SECD, partiendo de que el procedimiento de la expresión Lambda, se encuentra en el "MEDIO AMBIENTE":

PILA	MEDIOAMBIENTE	CONTROL	DUMP
s	((B))	(LDC NIL LDC1 CONS LD(0.0) AP)	d
(NIL.s)	((B))	(LDC1 CONS LD(0.0) AP)	d
(1 NIL.s)	((B))	(CONS LD(0.0) AP)	d
((1).s)	((B))	(LD(0.0) AP)	d
(B (1).s)	((B))	(AP)	d
NIL	((1))	(LD(0.0) LDC1 ADD RTN)	(s ((B)) NIL.d)
(1)	((1))	(LDC1 ADD RTN)	(s ((B)) NIL.d)
(1 1)	((1))	(ADD RTN)	(s ((B)) NIL.d)
(5)	((1))	(RTN)	(s ((B)) NIL.d)
(5 .s)	((B))	NIL	d

4.6.8 BLOQUE LOCALES

La regla de compilación de los bloques locales, es muy semejante a la de llamadas de funciones:

```
(LET e (x1.e1) ... (xk.ek)) = ((LAMB (x1...xk) e) e1...ek)
```

por lo tanto el código que se genera para un bloque LET tiene la siguiente forma:

```
(LET e (x1.e1) ... (xk.ek)) * n =
(LDC NIL) ¡ ek * n ¡ (CONS) ¡ ... ¡ e1 * n ¡ (CONS) ¡
(LDF e * m ¡ (RTN) AP)
```

donde "m" es el "MEDIO AMBIENTE" aumentado con los valores de:

```
m = (( x1 ... xk ) . n )
```

Como se puede apreciar el código, es parecido al código de las llamadas de funciones, solo que ahora la expresión "e", es compilada con respecto a una LISTA DE NOMBRES "m" aumentado por el número de variables locales. Tambien es importante notar, que las expresiones definiendo valores para esas variables locales son compiladas con respecto a la "LISTA DE NOMBRES" original "n", por lo tanto, cuando se ejecuta el código y se

obtienen los valores de e1, ... ,ek aún no se modifica la LISTA DE NOMBRES con las variables locales x1 ... xk, hasta que la instrucción AP es encontrada. Finalmente, dado que el código de LET tiene la combinación estructural de definiciones Lambda y Llamada de funciones, no es necesario demostrar la propiedad de efecto completo mediante algun ejemplo.

1.6.9 BLOQUES RECURSIVOS (LETR).

Las reglas de compilación de los bloques recursivos, varían muy poco en relación al de los locales. Ahora aparecen dos pequeños cambios. Primero ,las expresiones definiendo los valores de las variables, son compiladas con respecto a una LISTA DE NOMBRES aumentado (hay que recordar que en los bloques locales las variables eran compiladas con respecto a una LISTA DE NOMBRES original) y segundo, esto es realizado en base a un "MEDIO AMBIENTE" temporal, creado por la instruccion de máquina DUM, así el código que se genera es:

```
(LETR e (x1.e1) ... (xk.ek) ) A n =
     (DUM LDC NIL ¦ ek A m ¦ (CONS) ¦ ... ¦
          ¦ e1 A m ¦ (CONS) ¦ (LDF e A m ¦ (RTN) RAP )
```

donde m = ((x1 ... xk) . n)

Como ejemplo de este tipo de expresiones considere el siguiente caso:

```
(LETR (FAC (QUOT 6) )
      (FAC LAMB (X) _____))
```

donde FAC es la función de factorial. Compilando esta expresión con respecto a una LISTA DE NOMBRES vacio, se tiene:

```
(DUM LDC NIL LDF c CONS
             LDF (LDC NIL LDC 6 CONS
                  LD(0.0) AP RTN) RAP )
```

donde c es código de la funcion recursiva definida. Representando la ejecución de este código en los registros de la SECD , se tendría la siguiente secuencia de estados:

PILA	MEDIO AMBIENTE	CODIGO	RESPALDO
NIL	NIL	(DUM ... RAP)	NIL
NIL	a	(LDC NIL ... RAP)	NIL
(NIL)	a	(LDF c ... RAP)	NIL
((c.a)NIL)	a	(CONS ... RAP)	NIL
(((c.a)))	a	(LDF(LDC NIL ... RTN)RAP)	NIL
(((LDC NIL...RTN).a)((c.a)))	a	(RAP)	NIL

donde à es el 'MEDIO AMBIENTE' temporal y tiene la estructura siguiente:

$$\mathfrak{a} = (\mathfrak{L} \,.\, nil)$$

donde "$\mathfrak{L}$" indica el valor pendiente dentro del medio ambiente como se menciono en el capitulo tres, éste valor será remplazado por el valor que se obtenga al evaluar la expresión recursiva, a través de la función RPLACA. Ahora se tiene en la "PILA" una cerradura, para la expresión calificada en ese bloque, y en seguida una lista de los valores definidos (en este ejemplo es solo un valor que es la cerradura que representa a la función de FAC) . Ambas procedimientos contienen un "MEDIO AMBIENTE" temporal, así cuando se aplica la instruccion de RAP, este MEDIO AMBIENTE secundario se actualiza, tal que su car contiene la lista de los valores definidos.

Con esta última expresión bien estructurada, se concluye la descripción de las instrucciones de máquina, que genera cada una de las expresiones bien estructuradas, que componen el lenguaje de LISPKIT.

CAPITULO 5

IMPLEMENTACION DE LISPKIT

En este capitulo se explica la forma en que se implementó el sistema LISPKIT

Se principia dando una descripción de la màquina en que se implemento, y de el lenguage que se usó, se continua con las técnicas que se emplearon, el diseño del almacenamiento, así como el diseño del recolector de basura, màs adelante se pasa a explicar la programación de los diferentes módulos de que consta la implementación y las rutinas que se hicieron para cada una de los mismos.

5.1 LA MAQUINA Y EL LENGUAGE EN QUE SE IMPLEMENTO.

Se uso una minicomputadora PDP modelo 11/40 de propósito general con memoria principal de 256 Kbytes, con un sistema operativo de tiempo real con multiprogramación, llamado RSX-11M version 3.2, el código que utiliza esta minicomputadora es ASCII.

La implementación se realizó en el lenguaje FORTRAN IV. La versión de FORTRAN IV que se utilizó fue la 2.5 (de PDP), la cual provee de cuatro tipo de datos:

tipo	# de bytes
byte	1
entero	2
real	4
real*8	8

Por limitantes de capacidad de direccionamiento el tamaño màximo de memoria que puede utilizar una tarea en una 11/40 es de 64Kb (se pueden usar otros 64Kb para variables si se usa la opción de almacenamiento virtual

5.2 LAS TECNICAS UTILIZADAS

En esta sección se describen dos de las técnicas que se usaron para la implementación: recursividad y recolector de basura.

5.2.1 RECURSIVIDAD

Recursividad en computación se entiende al hecho de tener un procedimiento que se llame a si mismo (forma directa), o que llama a un segundo procedimiento, el cual en algún momento realiza una llamada al primer procedimiento (forma indirecta). Dos problemas se presentan con los procedimientos recursivos; uno, son las variables que usan, las que ocupan localidades de

memoria, y como cuando se llama a si mismo, usa las mismas localidades, se pierde su contenido previo. El otro problema es determinar la cantidad de veces que se ha llamado el procedimiento en forma directa o indirecta (llamado número de nivel, o profundidad).

Por lo anterior, para realizar programas recursivos es necesario contar con una estructura de datos (por ejemplo la "PILA(STACK)") para salvar los valores de las variables que se deben de preservar al llamarse el procedimiento recursivamente, y el número de nivel.

En este caso, se usó una PILA para salvar los valores y el número de nivel, incrementando el apuntador a la PILA en un número igual al de la cantidad de variables que se salvaron, de tal forma que al extraer de la PILA el número de nivel que se insertó al efectuar la primera llamada (cero en nuestro caso) sabemos que hemos terminado.

La PILA se definió como un vector de números enteros, cuyos elementos pueden ser insertados y extraidos solo en orden UEPS (Ultimo Entrar Primero Salir). Además se definió un apuntador a el tope de la PILA y dos banderas para indicar si se hallaba una carencia (UNDERFLOW) o un DERRAME(OVERFLOW.) de registros en la PILA. No se debe confundir la PILA que definimos para la recursividad con el que se usa en la máquina SECD ya que son totalmente diferentes.

5.2.2 RECOLECTOR DE BASURA

Por la forma en que maneja la memoria LISPKIT, se van creando registros (después se explica que es un registro) que posteriormente ya no se utilizan, (los registros que presentan esta característica se les llama basura (garbage)). En programas grandes se requiere de una gran cantidad de registros, y esto llega a provocar una terminación de los registros en memoria (al menos que se tengan cantidades muy grandes), por lo que se hace necesario proveer de un mecanismo para liberar registros de memoria que ya no se utilicen.

Existen 2 formas de recolectar registros ya no usados:

Una conocida como recolección de basura ("garbage collection"), opera en la siguiente forma : cuando se termina los registros libres, se procede a barrer todos los registros de la memoria y marcar aquellos que estan libres, y hacer otra pasada para pasar los libres a la cola de disponibles. Esta técnica tiene el inconveniente de volverse más lenta conforme se acaba la memoria disponible, y es muy lenta para aplicaciones de tiempo real y si falla en la busqueda por algún motivo se puede buscar infructuosamente.

Otra forma llamada de contadores de referencia consiste en agregar un bit a cada registro para indicar que esta ocupado, aunque esta tiene problemas ya que si se tienen varias

apuntadores a un registro es necesario un contador de referencia de la cantidad de apuntadores a tal registro. Además de que esta última técnica no sirve para listas circulares ya que el contador nunca es cero (ya que se marca a si mismo), además de que usa mucho espacio en cada nodo.

Para la implementación se eligió la de recolección de basura, aprovechando al máximo las características del diseño del área de almacenamiento con que se contaba.

La forma de realizarlo fué mediante un recorrido de todos los registros de la máquina SECD (lo que nos da todos los registros que se estan usando en ese momento), y marcarlos como ocupados, usando una mascara de bits . Posteriormente se barre todos los registros del área de almacenamiento y aquellos que no estén marcados se pasan a la lista de disponibles, eliminando la mascara de bits usada para indicar registros marcados.

5.3 ESTRUCTURA DE ALMACENAMIENTO

La estructura del almacenamiento que se utilizo para guardar los 3 tipos de registros que se manejan (simbólicos, numéricos y compuestos) conforman la memoria de la máquina SECD y se llama LIST SPACE. Y a continuación se describe como se diseño.

5.3.1 DISEÑO

Los puntos a considerar fueron:

-El LIST SPACE debe de estar en memoria principal todo el tiempo ya que de ponerlo en memoria secundaria, haria el sistema muy lento.

-El tamaño máximo de memoria que disponiamos es de 64 Kb, en nuestro caso se ocuparon para el código aproximadamente 30 Kb por lo que nos quedaban 34 Kb para el LIST SPACE.

-Los comandos de LISPKIT necesitan un máximo de 4 caracteres

-Se tenia que implementar un recolector de basura, lo que implicaba asignar espacio para marcar los registros, para poder recuperarlos mas tarde, y dicho recolector deberia ser eficiente.

-Los registros compuestos(cons) requerían de dos apuntadores a la localidad de los registros, y se deseaba manejar como 8 mil registros.

Tomando en cuenta las consideraciones anteriores se decidio que el LIST SPACE fuera un vector de registros reales (4 bytes en FORTRAN) que se ligaria al principio del proceso para simular una lista de registros disponibles.

La lista al principio tendria una configuración como la que se muestra:

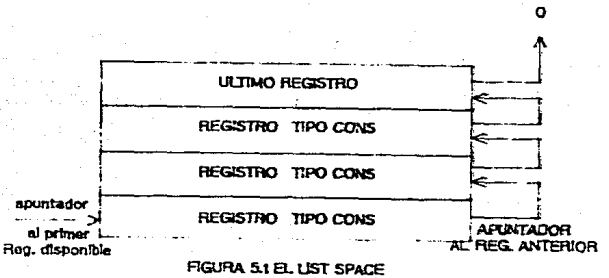


FIGURA 5.1 EL LIST SPACE

Esta lista se compone de registros del tipo CONS donde el CDR de cada uno de ellos se utiliza como apuntador al registro fisico anterior, el último registro de esta lista ligada apuntará a cero, lo que permite que se detecte facilmente el fin de la lista (o sea se acabaron los registros disponibles y habrá que llamar al recolector. Mas adelante se verá como con este vector se implementó el recolector de basura sin necesidad de usar más memoria. El vector se definió de un tamaño de 8500 registros (aprox. 33Kb), logrando así una máxima utilización de la memoria disponible y incrementando la velocidad del compilador.

Una vez que se definio el LIST SPACE como un vector ligado de tamaño "n", se pasó a definir como se utilizaria éste para almacenar los diversos tipos de registros.

5.3.2 COMO SE ALMACENAN LOS REGISTROS

Para diferenciar los tipos de registros se decidió utilizar el primer bit de los 4 bytes de los registros de LIST SPACE. La convención fue:

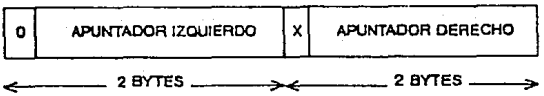
> el primer bit del primer byte servirá para determinar si un registro era CONS o alguno de los dos restantes (numérico o simbólico), de esta forma si dicho bit es 0 indica que el registro es CONS de lo contrario podria ser numérico o simbólico. Con el primer bit del tercer byte se diferencia entre un registro numérico y uno simbólico, así, si es 0 es numérico y si no es simbólico, esto siempre y cuando el primer bit del primer byte sea 1.

Con lo anterior queda la siguiente estructura de registros.

5.3.2.1 REGISTROS COMPUESTOS(CONS)

Estos registros constan de 2 apuntadores que se decidió que tuvieran una longitud de 15 bits cada uno, suficiente para direccionar las 8500 localidades del LIST SPACE.

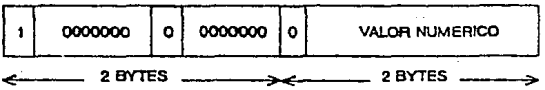
La forma de ponerlo en el registro real del LIST SPACE fue el sig:.



Es de notarse que queda 1 bit libre, el primer bit de el tercer byte.

5.3.2.2 REGISTROS NUMERICOS

Este tipo de registro tenia que albergar un número entero, y se decidio ponerlo en la parte derecha como si fuera un apuntador derecho de un registro CONS, quedando :



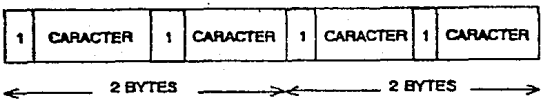
Aqui se usa el primer bit del primer byte y el primer bit de el 3 byte para identificar al registro como numérico y sus valores respectivos fueron 1 y 0.

1	el primer bit del primer byte
0	el primer bit del tercer byte

Notese que el primer bit del 2 byte se le coloca un cero, después se verá que se uso para el recolector de basura. Ademas de que los bits restantes se llenaron de ceros

5.3.2.3 REGISTROS SIMBOLICOS

Para los registros simbólicos se tenian que almacenar cuatro caracteres ASCII que utilizan 7 bits cada uno, por lo que se definió la siguiente estructura:



Aqui es de notarse que sobran cuatro bits y se decidió ponerle a cada uno un 1. Con esta estructura de registros tenemos las siguientes capacidades:

registros simbolicos	4 caracteres
reg numerico	1 entero positivo
reg compuesto	2 apuntadores con capacidad de 32767 registros.

5.3.3 COMO SE RECUPERA LA INFORMACION DE LOS REGISTROS.

Para la recuperación de la información de los registros, bastaba con quitarles las diversas mascaras de bits que se les habían colocado, por lo que no se explicarán, aunque se programaron rutinas para realizar esta función .

5.4 COMO SE HIZO EL RECOLECTOR DE BASURA

Una vez definidos los registros y como se almacenan es tiempo de pasar a describir como se marca y recupera la información en la fase de recolección del recolector de basura. Dado que había severos problemas de espacio, no fue posible crear un vector adicional de bits para marcar aquellos registros que estuvieran marcados, como sería la solución más sencilla, por lo que se opto por utilizar los bits que quedaban libres en los tres tipos de registro para efectuar el marcado de los registros ocupados.

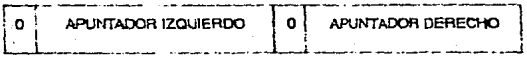
El problema de la recolección se divide en tres partes

1. el marcado de los registros ocupados
2. la recolección de aquellos que no lo fueron
3. regresar al estado original los registros marcados

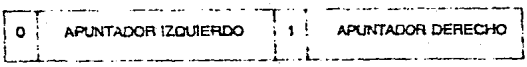
5.4.1 MARCADO

Para los registros CONS en donde solo estaba libre el primer bit del tercer byte, se opto por colocar un 1 a este bit para indicarlo como marcado.

primer bit del primer byte	primer bit del tercer byte	
0	0	reg CONS sin marcar
0	1	reg. CONS marcado



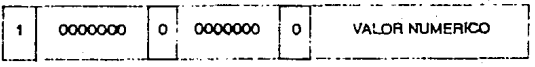
REGISTRO NO MARCADO



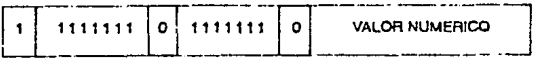
REGISTRO MARCADO

2. Para los registros numéricos, el primer bit del primer y tercer byte servirán para identificar que el registro es

numérico, y la forma de saber si esta marcado es usando los bits libres en las dos primeros bytes (los que no son usados), de tal forma que si todos estos bits son ceros es que no han sido marcados y si son unos indica que si han sido marcados, excepto, el primer bit de el segundo byte, el que por el tipo de marcado que se uso para los registros simbólicos se puso como 0

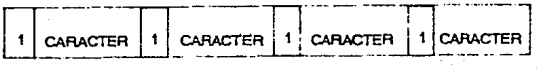


REGISTRO NO MARCADO

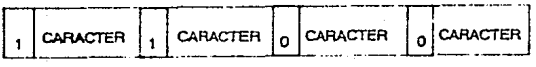


REGISTRO MARCADO

3. Para los registros simbólicos en los cuales se tienen dos bits libres (el primero de los bytes 2 y 4), se decidio identificarlos con el primer bit de los dos primeros bytes, y mostrar que estaban marcados con el primer bit de los dos ultimos bytes, quedando:



REGISTRO NO MARCADO



REGISTRO MARCADO

Esta forma de marcar resultaba totalmente segura y permite prencindir de un vector adicional para saber que registro se hallaba usado, aunque a costa de un poco mas de programación.

5.1.2 RECOLECCION DE REGISTROS

La recolección se realizó recorriendo todos los registros del LIST SPACE y pasando todos aquellos que estuvieran libres a la cola de disponibles.

5.1.3 RECUPERACION

Para poder recuperar el tipo de cada uno de ellos, se utilizó el siguiente procedimiento.

Recuperación de registros CONS

Como ya se dijo, con el primer bit del tercer byte se sabía si estaba libre, lo único que se tenía que hacer para recuperarlo era ver si el primer bit del tercer byte era 1, y si lo era se hacía cero y así, se recupera el registro.

Recuperación de registros numéricos

Aquí lo que se tenía que verificar primero era el tipo, que en el caso de numérico se hacía con el primer bit del primer y tercer byte, y si así era entonces verificar todos los bits del primer y segundo byte, si eran unos era que estaba marcado y se cambian a cero con lo que se recupera el registro original, de no ser así, se pasa a la lista de disponibles, convirtiendolo antes en registro del tipo CONS.

Recuperación de registros simbólicos

Para los simbólicos se verifican el primer bit de el primero y segundo byte si eran uno era registro simbólico, despues se debe checar si el primer bit del tercero y cuarto byte eran 0 con lo que se indica que esta marcado, de ser unos o cualquier otro se pasa a la lista de disponibles.

5.5 PROGRAMACION DE LA IMPLEMENTACION.

Una vez definidos los elementos y consideraciones que se hicieron para el diseño es tiempo de mostrar como se llevo a cabo la programación.

Para la creación de los programas se utilizaron las ideas que proporciona HENDERSON [1] en su libro, aun cuando hubo cosas que se ampliaron y otras se cambiaron. Para el diseño de los programas se aplicó diseño estructurado y programación estructurada. La explicación se hace por cada uno de los módulos principales del programa, asimismo se explica como se simuló la pila, el recolector de basura y las rutinas de uso general. Se utiliza pseudo-código a fin de que se pueda implementar en lenguages afines y no necesariamente en FORTRAN, los listados de las rutinas se pueden consultar en el anexo que se propociona.

5.5.1 MODULOS DEL PROGRAMA

El programa se divide en los siguientes módulos:

Control
Inicio
Ejecución
Depuración
Salida

[1] HENDERSON, P. (1980), Functional Programing Application and Implementation, Prentice Hall, London.

Dado que habia muchas rutinas comunes a los diversos módulos se programaron rutinas de servicio, las que se pueden dividir en dos grupos: las que se usan en mas de un módulo, que se llamaran generales; y las que son propias de un módulo especifico, que se llamaran particulares. Los commons se dividen al igual que las rutinas de servicio en generales y particulares. Las rutinas de servicio generales y los COMMONS generales se explican antes que cualquier módulo, ya que son referenciadas en la descripción de cada módulo.

La forma en que se explica cada uno de los módulos, es la siguiente: primero se explica el objeto del módulo, a continuación se indican las rutinas y los commons generales que utiliza el módulo, y finalmente se explican los commons particulares del módulo, para finalizar se proceden a explicar las diversas rutinas del módulo, poniendo énfasis en las mas importantes. A continuación en una forma "top-down" se procede a explicar las diversas partes de cada módulo, poniendo el "pseudo-código" de aquellas que son las mas importantes, finalmente se explican las variables que utilizan así como su procedencia (por common o parámetro).

5.5.2 PASO DE ARGUMENTOS

El paso de argumentos a las rutinas se hizo de dos formas, por commons y por parámetros. El criterio utilizado para cada uno de ellos fue: si los argumentos pasados a una subrutina eran enviados a otras subrutinas, entonces se usaron COMMONS. Si los argumentos no eran pasados a otras subrutinas, entonces se enviaban como parámetros para hacer mas explicitos los argumentos de la subrutina.

Rutinas de servicio generales

Las rutinas comunes a más de un módulo, se explican a continuación, se pueden dividir por su función en:

- Creación de registros
- Inserción de información en registros
- Extracción de información de registros
- Prueba de tipo de registros
- Manejo de pila
- Asignación de registros

Rutinas de servicio de creación de registros.

nombre	función
dosimb	crea un registro simbólico
docons	crea un registro CONS
donum	crea un registro numérico

Rutinas de servicio de inserción de información en registros.

nombre	función
insimb	inserta un símbolo en un registro simbólico
incar	inserta un valor en el car de un registro CONS
incdr	inserta un valor en el cdr de un registro CONS
insval	inserta un número en un registro numérico

Rutinas de servicio de extración de registros.

nombre	función
svalue	trae el símbolo de un registro simbólico
car	trae el valor de el car de un registro CONS
cdr	trae el valor de el cdr de un registro CONS
ivalue	trae el número de un registro numérico

Rutinas de servicio de prueba de registros.

nombre	función
issymb	prueba si es un registro simbólico
iscons	prueba si es un registro CONS
isnum	prueba si es un registro numérico

Rutinas de servicio de manejo de pila

nombre	función
push	coloca un valor en la pila
pop	extrae un valor de la pila

Rutinas de asignación de registros

nombre	función
newreg	asigna un registro de la pila de disponibles.

COMMONS GENERALES

nombre	variables	uso
vector	lispac	vector usado para el LIST SPACE
size	isize	tamaño de lispac
top	ff	apuntador a registro libre de lispac
stack	stack	vector entero usado para la PILA
maxs	maxstk	indica el máximo de registros de la PILA que se usaron.
kolect	csum	suma el total de registros de la pila que se usaron.
a	ipoint	apuntador a la pila
b	ifull	bandera de pila llena

d	idim	tamaño de la pila.
nil	nil	el registro NIL
res	result	pasa el resultado de la ejecución.

5.5.3 MODULO DE CONTROL

La fase de control sé lleva a cabo en el programa principal y su función consiste en efectuar las llamadas a los diversos módulos de acuerdo a lo que se vaya solicitando. Basicamente expresado en pseudo-código hace lo sig:

```
Rutina Principal
begin
        call inicio(fn, args)
        call mensaj(1)
        if ejecución then
                Call ejecuta(fn, args, result)
        call mensaj(2)
        if pidesalida then
                        Call salida
        call mensaj(3)
        if depuración then
                        Call depuración
End
```

5.5.4 MODULO DE INICIO

El módulo de inicio se utiliza para crear el LIST SPACE, y cargar en el la función y los argumentos que se leeran de el archivo FUNCIO.LIS. Regresa dos apuntadores, uno FN, apunta al inicio de la función y el otro ARGS a el inicio de los argumentos dentro del LIST SPACE.

El inicio del programa se divide en 3 submódulos:

```
creación de LIST SPACE
entrada de función
entrada de argumentos
```

y su pseudo-código sería:

```
rutina inicio (fn,args)
begin
  call inicia           ;se hace lista de disponibles
  call  donulo          ;se crea NIL
  call getexp(fn)       ;se lee y almacena la función
  call getlis(args)     ;se leen y almacenan los argumen
end
```

Las rutinas generales que usa inicio son:

nombre	función

getlin	trae una linea de la entrada
getchr	trae un caracter de la linea de entrada
gettok	trae un token de la linea de entrada
isdig	determina si un caracter es digito
islet	determina si un caracter es letra
addst	agrega un caracter a una cadena
scan	llama a gettok y anexa el fin de archivo si se da.

Los commons generales que usa INICIO son:

```
vector
size
top
```

5.5.4.1 CREACION DEL LIST SPACE

La creación del LIST SPACE se realiza con el subprograma INICIA que lo que hace es lo sig:

```
rutina INICIA
begin
 ff=0                              ;apuntador a los reg
 for i = 1,isize do
  begin
     cdr(lispac(i))=ff             ;inserto apuntador en lispac
     ff=i                          ;incremento apuntador
  end
end
```

La subroutina DONULO, tiene como objeto crear un registro nulo(NIL) que es necesario, para el proceso.

5.5.4.2 ENTRADA DE LA FUNCION

La entrada de la función se inicia con una llamada a la rutina GETCHR que lo que hace es traer un caracter de la entrada, y hace uso de la rutina GETLIN, quien es la que trae una nueva linea de entrada al agotarse la anterior. Esta llamada es necesaria para que al principio tenga algo que procesar la subroutina GETEXP que es la encargada de realizar la lectura y almacenamiento de la función.

Como la función y los argumentos de todo programa en LISPKIT se deben de colocar como EXP-S, es necesario explicar antes la sintáxis de las mismas.

El diagrama de sintáxis de una EXP-S se mostro en la figura 2.1. Del diagrama se deduce que se puede hacer un analizador del tipo llamado "recursivo descendente", que consiste en escribir un procedimiento recursivo para cada categoria de frase (EXP-S) y (LIST. de EXP-S) cuyo trabajo es barrer la entrada y reconocer la frase con la cual esta asociado.

Para lo anterior se uso una rutina llamada GETTOK (token, tipo) la cual una vez que es llamada regresa el siguiente elemento basico (TOKEN) de la linea de entrada asi como su tipo, que puede ser:

alfanumérico	(átomo simbólico)
numérico	(átomo numérico)
delimitador	(paréntesis)
endfile	(fin de entrada)

ejemplo: supongamos que tenemos de entrada la línea
[1]
(3 tristes tokens)eof

gettok regresa:

token	tipo
-----	----
(	delimitador
3	átomo numérico
tristes	átomo alfabético
tokens	átomo alfanumérico
)	delimitador
eof	end of file

La rutina getexp(e), utiliza las rutinas SCAN y GETLIS , además de las rutinas generales IVALUE, DONUM, DOSIMB y la particular TOINT (pasa un token a numérico). SCAN lo que hace es solicitar un token de entrada auxiliandose de la rutina GETTOK, y determina el tipo del mismo. Y GETLIS trae una lista de EXP-S

GETEXP tiene el sig. pseudo-código

```
rutina GETEXP(E)
; en E regresa el registro en el que queda
; el inicio de la función de entrada
;
begin
 if token = "(" then
  begin call SCAN;
        call GETLIS;
        call SCAN
  end
```

[1] eof es fin de archivo, o un caracter no definido.

```
else if TYPE = "numeric" then
        begin e::=donum;
              ivalue(e)::=toint(token);
              call SCAN
        end
      else
      begin
              e::=dosimb;
              SVALUE( e)::=token
              call scan
      end
  end
```

Notese que como la función es una EXP-S puede contener una lista de EXP-S, por lo que llama al procedimiento GETLIS que se ocupa de almacenar una lista de EXP-S

5.5.4.3 ENTRADA DE LOS ARGUMENTOS

La entrada de los argumentos que se definen como una lista de EXP-S se hace con la rutina GETLIS que tiene el sig pseudo-código.

```
rutina GETLIS(E)
; regresa en E el registro en el que se inician
; los argumentos.
;
begin
 e::=docons;                          ;se hace E un reg. cons
 GETEXP (car(e));                     ;se trae la parte izq.
 if token = "." then
  begin SCAN; GETEXP (cdr(e)) end
 else if token = ")" then
cdr(e) := nil
 else
        GETLIS (cdr(e))
end
```

Es de notarse que GETLIS es una rutina recursiva por lo que en la implemehtación se hizo uso de la PILA para simularla.

5.5.5 MODULO DE EJECUCION

En este módulo se procede a realizar la ejecución en si, de la función con los argumentos que se colocaron en el LIST SPACE en el módulo de inicio.

El módulo al principio efectua preguntas sobre la forma en que se realizará la ejecución. La forma en que se simularon los registros de la máquina SECD, las constantes y los registros de trabajo fue la siguiente: Los registros de la máquina SECD (s, e, c, d), fueron declarados como variables enteras que sirven

como apuntadores a las localidades del LIST SPACE, donde se iniciaba el contenido del registro que se simula, además se definieron variables simbólicas para los valores constantes TRUE, FALSE y NIL y dos registros de trabajo (w y w2). Estos registros se usan para guardar valores temporales durante la ejecución de cada instrucción.

Se necesitaban inicializar los registros de la máquina SECD, así como los otros registros. Por lo tanto, como se dijo que al iniciarse la ejecución en el registro S (stack), se deben hallar los argumentos de la función a evaluarse, en el registro c (control) se debe colocar la función a ejecutarse, los otros dos registros e y d se inicializan con un NIL. Además se deben inicializar las constantes TRUE, FALSE y NIL.

Cada operación en el lenguage compilado tiene un código de operación para ejecutarse en la máquina SECD (un número entero entre 1 y 21), esto facilita bastante la ejecución ya que de acuerdo al código que se tenga se efectua una llamada a la rutina correspondiente. Ademas, en la primera posición de la lista de control se halla siempre el código de una instrucción (como se vio en el capitulo 3). Por lo que al ejecutar cada instrucción de la máquina SECD simplemente lo que tenemos que hacer es tomar alguno de los 21 caminos posibles, dado que hay 21 instrucciones diferentes. Asi lo que tenemos que hacer para implementar las instrucciones es en principio tener lo siguiente:

```
ciclo:  case ivalue(car(c)) of
        begin
                1       código para ldc
                2       código para ld
                3       código para ...
                "       "
                "       "
                "       "
                21      código para stop
```

Al terminar la ejecución se debe encontrar en la primera posición de el registro s el resultado, por lo que el módulo debe regresar la posición que ocupa dicho resultado.

El módulo de ejecución se dividio en tres fases, una en la que se piden instrucciones (EXEMEN), otro en el cual se inicializan las variables y el último que es en el cual se ejecutan las instrucciones que se hallen en la función de entrada.La estructura que tendrá entonces el módulo de ejecución será:

```
subroutina ejecuta(fn, args, result)
begin
        call exemen     ;se piden mensajes
        call inivar     ; se inicializan variables
        call cycle      ;se hace el ciclo de ejecución
        result=car(s)   ; se obtiene el resultado
end
```

Las rutinas generales que utiliza ejecución son:

```
car
cdr
docons
donum
ivalue
svalue
issymb
isnum
iscons
```

los commons generales que usa son:

```
vector
size
top
stack
maxs
kolect
a
b
d
nil
res
```

5.5.5.1 RUTINA DE MENSAJES

Para dar la posibilidad de hacer un seguimiento de los pasos de la ejecución de un programa, se dio la posibilidad de poder grabar en un archivo las instrucciones de la máquina que se ejecutan, así como el valor de las cuatro registros de la máquina SECD que definen su estado (ver capítulo 3), asimismo se dió esta facilidad, pero siendo el medio la pantalla en la cual se este ejecutando el programa. La rutina se llama EXEMEN y lo que hace es solicitar alguna de las siguientes opciones al usuario:

1	se graban los pasos en el archivo EXEC.LST
2	se despliegan los pasos en la pantalla
0	no se hace ninguno de los anteriores.

Asimismo en EXEMEN se envia un mensaje indicando el número de registros que se hallan colectado en las diversas llamadas al colector de basura, de tal forma que el usuario sepa cuantos registros se recuperaron.

Asimismo se indica el número de registros totales que utilizó el programa para ejecutarse.

Rutina INIVAR

En esta se inicializan las variables necesarias para la ejecución, el código queda en la siguiente forma:

```
rutina inivar(fn, args)
begin
        s:=docons          ;sea s un registro cons
        car(s):=args
        cdr(s):=nil
        e:=fn
        c:=nil
        d:=nil
        t:=dosimb
        t:='true'
        f:=dosimb
        f:='false'
end
```

5.5.5.2 RUTINA DE PROCESO

En esta se efectua el programa que llega en los argumentos ,basicamente lo que hace es obtener el valor de el CAR del registro c (que es el código de la instrucción a ejecutar), validarla y llamar a la rutina correspondiente, además de que si se solicito impresión o despliegue de el código que se esta ejecutando, lo realiza llamando a rutinas secundarias

```
begin
cycle : case ivalue(car(c) of
        begin
        1:  ldc
        2:  ld
        3:  ...
          :
          :
          :
        21: goto endcycle;
        end;
        goto cycle;
endcycle:   end
end
```

La forma en que se implementaron las 21 primitivas de la máquina SECD se explican a continuación, cada una de las primitivas se colocó en una rutina separada y todas ellas practicamente tenian los mismos commons, y ninguna uso parámetros.

LDC

la transición de LDC que se vio en el cap. 3 fue

s e (LDC x.c) d ----> (x.s) e c d

Puede ser implementada con las siguientes instrucciones.

```
s:=docons(car(cdr(c)),s);
c:=cdr(cdr(c));
```

Esto es, se pasa la constante x la pila y en control se deja la sublista c

LD

La transición de LD es:

s e (ld i.c) d ----> (locate(i,e).s) e c d

La que se implementa con:

```
w:=e                          ;salvo e
for i:=1  until car(car(cdr(c))) do w:=cdr(w)
w:=car(w);
for i:=1 until cdr(car(cdr(c))) do w:=cdr(w)
w:=car(w)
s:=docons(w,s)
c:=cdr(cdr(c))
```

CAR

La transición es:

((a.b).s) e (CAR .c) d ----> (a.s) e c d

La cual se implementa con:

```
s:=docons(car(car(s)),cdr(s));
c:=cdr(c)
```

CDR

La implementación de CDR es casi idéntica.

```
s:=docons(cdr(car(s)),cdr(s));
c:=cdr(c)
```

ATOM

Para implementar ATOM se usa

```
if(isnum(car(s)) or issymb(car(s)) then
                    s:=docons(T,CDR(S)) else
                    s:=docons(f,cdr(s));
c:=cdr(c)
```

CONS

La instrucción CONS tiene la siguiente transición:

(a b.s) e (CONS.C) d ----> ((a.b).s) e c d

y se implementó con:

```
s:=docons(docons(car(s),car(cdr(s))),cdr(cdr(s)));
c:=cdr(c)
```

Las operaciones aritméticas ademas de requerir sus operandos, necesitan un nuevo registro para almacenar el resultado, de hecho las operaciones aritméticas se implementan en forma idéntica, por lo que solo explicaremos una de ellas.

La transicio'n de add es:

(a b.s) e (SUB .c) d ----> (b+a.s) e c d

Y se implementa con :

```
s:=docons(donum(ivalue(car(cdr(s)))-ivalue(car(s))),cdr(cdr(s)));
c:=cdr(c)
```

La implementación de LEQ es la siguiente

```
if ivalue(car(cdr(s)))= ivalue(car(s)) then
s:=docons(T , cdr(cdr(s))) else s:=docons(f,cdr(cdr(s)));
c:=cdr(c)
```

La implementación de EQ queda:

```
if issymb(car(s)) and issymb(car(cdr(s))) and
        svalue(car(s))=svalue(car(cdr(s)))
   or isnumber(car(s)) and isnumb(car(cdr(s))) and
        ivalue(car(s))=ivalue(car(cdr(s)))
then s:=docons(T, cdr(cdr(s))))
else s:=docons(F, cdr(cdr(s)));
c:=cdr(c)
```

La de LDF queda:

```
s:=docons(docons(csr(cdr(c)),e),s);
c:=cdr(cdr(c))
```

La de AP queda:

```
d:=docons(cdr(cdr(s)),docons(e,docons(cdr(c),d)));
e:=docons(car(cdr(s)),cdr(car(s)));
c:=car(car(s));
s:=nil
```

La de RTN queda:

```
s:=docons(car(s),car(d));
e:=car(cdr(d));
c:=car(cdr(cdr(d)));
d:=cdr(cdr(cdr(d)));
```

La transición de DUMP es:

```
s e (DUM .c) d ---> s  ( .e)   c  d
```

La que se implementa con :

```
e:=docons(nil,e);
c:=cdr(c);
```

la de RAP es:

```
((c'.e')v.s) ( .e) (RAP. c)  d ----> nil  rplaca(e',v)  c' (s e
 c.d)
```

que se programa con:

```
d:=docons(cdr(cdr(s)),docons(cdr(e),docons(cdr(c),d)));
e:=cdr(car(s));
car(e):=car(cdr(s));
c:=car(car(s));
s:=nil
```

La transición de SEL es:

```
(x.s)  e (SEL c(t) c(f).c)  d ---> s e c (c.d)
```

y se programa con:

```
d:=docons(cdr(cdr(cdr(c))),d);
if svalue(car(s))= "t"    then c:=car(cdr(c))
                          else  c:=car(cdr(cdr(c)));
s:=cdr(s);
```

Finalmente JOIN tiene la siguiente transición:

s e (JOIN) (c.d) ----> s e c d

que se implementa con:

```
c:=car(d);
d:=cdr(d);
```

5.5.6 MODULO DE SALIDA

La salida del sistema seran como EXP-S, por lo que se debe ver la forma de hacerla. Dado que una EXP-S puede expresarse en muchas formas dependiendo del número de puntos que se le coloquen, se eligió aquella que usaba la menor cantidad de puntos, y esto se logró haciendo que sólo se usara un punto si precedía a un átomo.

el módulo de salida se llama PUTEXP y recibe como argumento la localidad a partir de la cual se vaciará (generalmente será la del resultado). El codigo de putexp es:

```
rutina putexp(e)
begin
if issimb(e)
        then puttok(svalue(e))
     else
        if(isnumber(e)
                then puttok(tostri(ivalue(e)))
                else
                begin pointer p;puttok("(");p:=e;
                        while iscons(p) do
                            begin putexp(car(p));p=cdr(p) end;
                        if issymb(p) and svalue(p) ="nil"
                            then skip
                            else begin puttok(".");putexp(p) end;
                        puttok(")")
end
```

El algoritmo fue el siguiente:

Si es símbolo se escribe, igualmente si es número, de ser una cons, se escribe su car, y se va barriendo el cdr al dejar de ser cons, si es NIL se termina, si no se pone un punto y se hace una llamada recursiva a putexp.

Las rutinas que se usaron son:

puttok	coloca un token en la salida
tostri	convierte un número a caracter
iscons	rutina general
issymb	" "
isnum	" "
svalue	" "

Los commons generales que uso fueron:

```
vector
a
b
d
stack
```

Los commons particulares fueron

```
e       outptr  apuntador a buffer de salida
        outbuf apuntador a buffer de salida.
```

La rutina puttok tiene el sig. código:

```
rutina putok(token)
 begin
        len=lenght(token)
        for i=1 until len do putcha(token(i));
        putcha(" ")
end
```

Las rutinas que usa son:

```
lenght          determina la longitud de un token
putchr          coloca un caracter en la salida
```

5.5.7 MODULO DE DEPURACION.

Al terminar la ejecución, o antes de realizarla, se hace necesario en ocasiones, sobre todo en la fase de depuración de un programa, de revisar el contenido de localidades de la memoria, la forma en que se cargo el código, el valor de una localidad en particular, etc, para esto se creo el módulo de depuración, que nos permite:

1. vaciar un conjunto de registros de memoria
2. listar el contenido de un registro en particular

Las rutinas generales que uso fueron:

```
iscons
isnum
issymb
```

Los commons generales fueron:

```
vector
top
a
nil
```

la impresión de los registros solicitados queda en el archivo DEBUG.LST

5.5.8 IMPLEMENTACION DEL RECOLECTOR DE BASURA

Como se explicó con anterioridad el recolector de basura se realizó usando la técnica de barrer los registros de la máquina SECD para marcarlos como ocupados, y después se barre toda el LIST SPACE pasando a la lista de disponibles aquellos que no esten marcados.

Las rutinas generales usadas fueron:

```
push
pop
iscons
incdr
```

Los commons generales fueron:

```
kolect
maxs
pasos
size
top
poitrs
a
vector
```

Las rutinas que se usaron son:

nombre	funcion
garcol	rutina principal del colector
mark	marca la lista de registros asociados a uno de los 4 registros de la máquina SECD
markya	verifica si ya se marco un registro
marcar	marca un registro
colect	recorre el LIST SPACE recogiendo todos los registros que no esten marcados.
ocupad	verifica si un registro esta ocupado, si lo esta lo regresa a su estado original (quita marcas)

La primera rutina del recolector de basura es GARCOL, la cual realiza el marcado de cada uno de los registros de la máquina SECD, así como de los registros auxiliares, y al final llama a COLECT para que haga la recolección en sí.

El código de garcol es:

```
rutina garcol
begin
        call mark(s);
        call mark(e);
        call mark(c);
        call mark(d);
        call mark(w);
```

```
        call mark(w2);
        call mark(t);
        call mark(f);
        call mark(nil);
        call colect(k)  ; se hace la recoleccion
end
```

La rutina MARK se encarga de marcar todos los registros fisicos que tenga un registro, esto basicamente se puede realizar con un procedimiento recursivo (el cual se tuvo que simular), ademas como se pueden presentar listas circulares, se hacia necesario que cada registro se verificara si no se habia marcado con anterioridad. el código de mark es:

```
rutina mark(n)
 begin
        if markya(n)= false
          then
                begin
                        marcar(n)
                        if iscons(n) then
                        mark(car(n))
                        mark(cdr(n))
                end
end
```

La rutina MARCAR realiza el cambio de los bits de un registro fisico de acuerdo a la forma que se indicó en la sección 5.4.1

```
procedimiento marcar(n)
begin
        if iscons(n) then
          pon marcas de cons
        if issimb(n) then
                pon marcas de simbolo
        if isnum(n) then
                ponmarcas de número
        else error
end
```

La rutina COLECT se encarga de recolectar todos aquellos registros que se hallen libres en la siguiente forma; recorre cada registro del LIST SPACE, si esta ocupado se lo salta, si esta libre lo liga a la cola de disponibles.

```
rutina colect
begin
 for i=1 until isize do
        if ocupad(n) = false then
        cdr(i)=ff
        ff=i
        end
end
```

La rutina OCUPAD checa si un registro está ocupado y si lo está lo regresa a sus estado original (ya que se le alteraron algunos bits para indicar que estaba ocupado.

```
rutina ocupad
begin
 if iscons(n) then
        if esta ocupado then
                quita mascaras de cons
if issimb(n) then
        if esta ocupado then
          quita mascaras de simbolo
if isnum(n) then
        if esta ocupado then
                quita mascaras de numero
else error
end
```

5.5.9 IMPLEMENTACION DE LA PILA

La PILA se implantó con dos rutinas; PUSH y POP, donde la primera se ocupaba de insertar un valor entero en la PILA, y la segunda de extraerlo

Los commons que usaron fueron:

```
size
a
b
d
```

y su código fue:

```
rutina push(n)
begin
ifull=false
if n <= idim
then
        stack(ipoint)=n
        ipoint=ipoint+1
else
        ifull=true
end

rutina pop(n)
begin
iempty=false
if ipoint > 0
then
        n=stack(ipoint)
        ipoint=ipoint-1
else
        iempty=true
end
```

5.6 CREACION DE LA TAREA.

Una vez definidas las rutinas y creadas a través del editor del sistema RSX11M, el paso inmediato fue crear la tarea. Para esto era necesario compilar las diversas rutinas, y posteriormente ligarlas para obtener el código ejecutable. Para hacer más sencillos estos dos pasos se utilizaron archivos de comandos indirectos, listados de los cuales se encuentran en el manual del sistema que se anexa a esta tesis.

5.6.1 LA COMPILACION

La compilación de todas las rutinas programadas se hizo con el archivo de comandos LISPKIT.COM, el cual se usó en la siguiente forma:

```
>for @lispkit.com
```

Al término de la ejecución de este archivo se habran generado todos los código objetos de cada una de las rutinas.

5.6.2 EL GENERADOR DE TAREAS.

Para ligar la tarea se utilizó el generador de tareas del S.O. que se llama Task Builder(TKB), en el cual se usó la facilidad de "OVERLAYS", la cual se explica a continuación.

5.6.2.1 OVERLAY.

Cuando una sección de el código objeto de un programa es cargado en un area de memoria central que estaba ocupada por otra sección de el mismo programa, el proceso es llamado superposición ("overlaying") y la sección cargado se le llama "overlay". Un overlay puede contener instrucciones o datos, o una combinación de ambos.

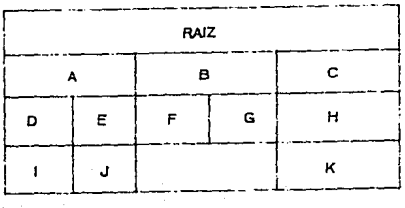
La técnica de overlay es usada para permitir a tareas correr en una computadora inclusive si sus requerimientos totales de memoria son más grandes que la cantidad de memoria disponible para el. La cantidad de memoria disponible puede ser toda la memoria central en un medio ambiente de uniprogramación, o solo ser una parte de la memoria total en un ambiente de multiprogramación.

En un programa usando overlays, una parte inicial de el programa es cargado en la memoria principal, el resto permanece en disco. Durante la ejecución del programa, instrucciones son ejecutadas para causar que toda o parte de el programa residente en memoria sea sobrepuesto por secciones especificas del programa en disco.

La PDP usa un método muy sencillo de overlays llamado encadenamiento. los overlays encadenados son relativamente independientes y auto-contenidos. Una liga de la cadena podra

ser cargada a memoria y ejecutada hasta que se termine por la carga de otra liga sobre si misma. La siguiente liga se ejecutará y terminará en la misma forma cargando un overlay sobre si misma y dando control a el nuevo overlay. Una área comun de datos no sobreposicionable en memoria declarada por la declaración common, da la única comunicación entre las ligas de el programa encadenado.

En un overlay se especifica un segmento raiz el cual reside en memoria durante la ejecución del programa. El área que se inicia donde acaba la raiz puede ser ocupada por A, B, o C. Si es ocupada por A entonces el área que se inicia donde termina A puede ser ocupada por D o E, sin embargo ni D o E pueden estar en memoria sin estar A. Una referencia en la raiz a una localidad de C, hará que C se carge a memoria, removiendo a A y sus niveles inferiores.



Un ejemplo de como se reduce la memoria con overlays es el siguiente:

Supongamos que tenemos tres rutinas llamadas PRINCI, INICIO, PROCESO y SALIDA con los siguientes tamaños:

SEGMENTOS	TAMANO
CONTROL	1000
INICIO	2000
PROCESO	2500
SALIDA	2000

Si se arma la tarea sin overlay se usarían 7500 bytes, en cambio usando overlays solamente se usarán 3500 bytes(1000 y 2500 aprox., porque la estructura de overlay usa un poco de memoria).

5.6.2.2 EL LENGUAJE ODL

Para definir Overlays se usa un lenguage llamado Overlay Description Language(ODL), una descripción de overlay se hace por medio de un archivo con una serie de directivas odl, una por linea.

Cada linea de odl tiene la sig. sintaxis:

etiqueta: directiva argumentos;comentarios

las directivas que nos interesan son:

```
.root
.fctr
.end
```

Las directivas root indica que segmentos estaran en la raiz y que ramas tendrá la raiz, hace uso de los operadores siguientes (también los usa FCTR)

El guion (-) indica la concatenación de segmentos, o sea, indica los segmentos que compartirán memoria.

La coma (,) la sobreposición de segmentos en memoria real, o sea, aquellos que se sobrepondran entre si.

por ejemplo:

```
(x,y)
```

indica que x e y compartirán la misma memoria

La directiva FCTR sirve para crear árboles grandes en forma clara donde la etiqueta es alguna referenciada por ROOT o FCTR

la directiva END indica el fin del overlay

ejem:

```
.root raiz-(A,B,C); raiz con tres ramas(A,B,C)
A: .fctr (D,E)              ;rama con dos subramas
B: .fctr (F,G)              ;rama con 2 subramas
C: .fctr (H,I)              ;rama con 2 subramas
.end
```

CAPITULO 6

EJEMPLOS

En este capitulo se muestran ejemplos de uso del compilador LISPKIT, se inicia con ejemplos muy sencillos para mostrar las capacidades del lenguaje. Los ejemplos avanzan en complejidad hasta que finalmente se muestra el compilador que se utilizó.

La forma en que se presentan los ejemplos es la siguiente:

1. explicación del problema
2. pseudocódigo
3. programa en lenguaje LISPKIT
4. código objeto obtenido del compilador
5. argumentos
6. resultados de aplicar la función a los argumentos.

6.1 OPERADORES ARIMETICOS

Mostraremos como se llevan a cabo operaciones aritméticas, específicamente se realizará la operación:

a+b×c

el pseudo-código seria :

```
a+b×c
```

Expresado como una función (en LISPKIT todo entra como función) quedaria:

```
( lamb (a b c)
  (add a (mul b c))
)
```

Y su código objeto al pasar por el compilador instalado en la máquina SECD seria

```
( 3 ( 1 ( 0 . 0 ) 1 ( 0 . 1 ) 1 ( 0 . 2 ) 17 15 5 )
 4 21 )
```

Si se dieran como argumentos los siguientes valores:

```
1 2 3
```

Obtendriamos la siguiente salida:

```
7
```

Es de notarse que los valores no entran entre paréntesis, ya que la función no espera una lista, sino solo átomos.

6.2 USO DE SELECTORES

Veamos ahora un ejemplo para mostrar el uso de los selectores CAR y CDR. si se desee obtener el segundo elemento de una lista se tendría lo siguiente:

```
second(a)=car(cdr(a))
```

Que expresado en LISPkIT quedaría:

```
( lamb (a )
        ( car (cdr a))
)
```

El código objeto resultante sería:

```
( 3 ( 1 ( 0 . 0 ) 11 10 5 ) 4 21 )
```

Y si la lista de entrada "a" tuviera los siguientes valores:

```
(a b c d e )
```

Obtendríamos lo siguiente:

```
b
```

6.3 FUNCIONES RECURSIVAS

Para evaluar funciones recursivas podríamos usar el siguiente ejemplo en el cual se determina la longitud de una lista X, el algoritmo es:

```
long(x)= if eq (x,nil) then 0
              else long(cdr(x)+1)
```

Expresado en LISPKIT sería:

```
(letr long
 (long lamb (x)
       (if (eq x (quot nil)) (quot 0)
           (add (long (cdr x)  ) (quot 1))
       )
 )
)
```

Cuyo código objeto es:

```
( 6 2 nil 3 ( 1 ( 0 . 0 ) 2 nil 14 8 ( 2 1 9 ) ( 2 nil
 1 (

      0 . 0 ) 11 13 1 ( 1 . 0 ) 4 2 1 15 9 ) 5 ) 13 3 (
1 (

    0 . 0 ) 5 ) 7 4 21 )
```

Y si la lista "x" es igual a:

(a b c d e f g)

El resultado sería:

7

Hagamos una función que agrege a una lista los elementos de otra.

```
[append whererec
        append =(x, y) if x = nil then y
                                  else cons(car(x), append(cdr(x), y))
```

En LISPKIT queda:

```
(letr appe
     (appe  lamb (x y)
        (if (eq x (quot nil)) y
            (cons (car x)
                (appe (cdr x) y)))))
```

Cuyo código objeto queda:

```
( 6 2 nil 3 ( 1 ( 0 . 0 ) 2 nil 14 8 ( 1 ( 0 . 1 ) 9
) (
    2 nil 1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 11 13 1 ( 1 . 0 )
 4
   1 ( 0 . 0 ) 10 13 9 ) 5 ) 3 ( 1 ( 0 . 0 ) 5 ) 7
4 21 )
```

Si los parámetros X y Y tienen los siguientes valores:

(a) (b)

El resultado será:

(a b)

6.4 DEFINICIONES LOCALES

Definamos una función en donde se usan definiciones locales, esto es, valores que son validos solo durante el cuerpo delimitado por una instrucción LET, esta función hará la suma y el producto de los elementos numéricos de 2 listas, y hará uso de la función twolist, que une dos elementos en una lista, el algoritmo es:

```
sumprod(x)= if eq(x,NIL)) then
                twolist
                else
                        [let z = sumprod(cdr(x))
                        twolist(car(z)+car(x),car(cdr(z))*car(x))]
```

Donde twolist es:

```
twolist(x,y)=CONS(X,CONS(Y,NIL))
```

El código en LISPKIT es:

```
(letr supr
   (supr lamb (x)
      (if (eq x (quot nil)) (twol (quot 0) (quot 1))
         (let (twol (add n (car z)) (mul n (car(cdr z))))
              (n car x)
              (z supr (cdr x)))))
     (twol lamb (i j)
           (cons i (cons j (quot nil)))))
```

Y su código objeto es:

```
( 6 2 nil 3 ( 2 nil 1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 13 5 )
13
   3 ( 1 ( 0 . 0 ) 2 nil 14 8 ( 2 nil 2 1 13 2 0 13 1 (
1 .
    1 ) 4 9 ) ( 2 nil 2 nil 1 ( 0 . 0 ) 11 13 1 ( 1 .
0 )
    4 13 1 ( 0 . 0 ) 10 13 3 ( 2 nil 1 ( 0 . 0 ) 1 ( 0
.
  1 ) 11 10 17 13 1 ( 0 . 0 ) 1 ( 0 . 1 ) 10 15 13 1 ( 2
.
   1 ) 4 5 ) 4 9 ) 5 ) 13 3 ( 1 ( 0 . 0 ) 5 ) 7 4 21 )
```

Si la lista X es la siguiente:

(4 5 6)

el resultado será:

(15 120)

6.5 DIFERENCIACION

Veamos un ejemplo en el cual se define una función que permite derivar formulas que contengan variables, constantes, y las operaciones + y *.

las reglas serían:

```
D x(x)=1
D x(y)=0          y <> de x (y constante o variable)
D x(e1+e2)= D x(e1) + D x(e2)
D x(e1*e2)= e1 * D x(e2) + D x(e1) * e2
```

El código en LISPKIT sera :

```
(letr  diff
  (diff lamb (e)
     (if (atom e) (if(eq e (quot x)) (quot  1)
                  (quot 0))
     (if (eq (car e) (quot add))
         (let (sum (diff e1) (diff e2))
               (e1 car (cdr e))
               (e2 car (cdr (cdr e))))
     (if (eq (car e) (quot mul))
         (let (sum (prod e1 (diff e2))
                   (prod (diff e1) e2))
               (e1 car (cdr e))
               (e2 car (cdr (cdr e))))
     (quot mal)))))
  (sum  lamb (u v)
        (cons (quot add)
              (cons u (cons v (quot nil)))))
  (prod lamb (u v)
        (cons (quot mul)
              (cons u (cons v (quot nil))))))
```

El código objeto seria:

```
( 6 2 nil  3 ( 2 nil  1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 13 2
mul
  13 5 ) 13 3 ( 2 nil  1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 13 2
add
   13 5 ) 13 3 ( 1 ( 0 . 0 ) 12 8 ( 1 ( 0 . 0 ) 2 x 14
8 (
    2 1 9 ) ( 2 0 9 ) 9 ) ( 1 ( 0 . 0 ) 10 2 add 14 8
(
  2 nil  1 ( 0 . 0 ) 11 11 10 13 1 ( 0 . 0 ) 11 10 13 3 (
2 nil
      2 nil  1 ( 0 . 1 ) 13 1 ( 2 . 0 ) 4 13 2 nil  1 (
0 .
    0 ) 13 1 ( 2 . 0 ) 4 13 1 ( 2 . 1 ) 4 5 ) 4 9 ) (
1 (
    0 . 0 ) 10 2 mul  14 8 ( 2 nil  1 ( 0 . 0 ) 11 11 10
13 1 (
     0 . 0 ) 11 10 13 3 ( 2 nil  2 nil  1 ( 0 . 1 ) 13 2
nil
    1 ( 0 . 0 ) 13 1 ( 2 . 0 ) 4 13 1 ( 2 . 2 ) 4 13 2
nil
    2 nil  1 ( 0 . 1 ) 13 1 ( 2 . 0 ) 4 13 1 ( 0 . 0 )
13
  1 ( 2 . 2 ) 4 13 1 ( 2 . 1 ) 4 5 ) 4 9 ) ( 2 mal  9 )
```

```
 9 )
      9 )  5 )  13 3 (  1 (  0 .  0 )  5 )  7 1 21 )
```

Si los argumentos de entrada fueran :

```
(add (add (mul x x) (mul x 2)) 1)
```

Los cuales representan:

$$x*x + 2x + 1$$

La salida obtenida con los parámetros originales sería :

```
( add ( add ( add ( mul x 1 ) ( mul 1 x ) ) ( add (
   mul x 0 ) ( mul 1 2 ) ) ) 0 )
```

La cual es igual a :

$$x + x + 2 + 0 = 2x + 2$$

6.7 EL COMPILADOR LISPKIT

A continuación se muestra el código fuente y objeto del compilador LISPKIT.

```
;   código fuente checado de  liskit
;
(letr comp
  (comp lamb (e)
    (com e (quot nil) (quot (4 21))))


  (com lamb (e n c)
    (if (atom e)
        (cons (quot 1) (cons (loca e n) c))

    (if (eq (car e) (quot quot))
        (cons (quot 2) (cons (car (cdr e)) c))

    (if (eq (car e) (quot add))
        (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 15)

        c)))

    (if (eq (car e) (quot sub))
        (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 16)
        c)))
```

```
   (if (eq (car e) (quot mul))
       (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 17)
       c)))

   (if (eq (car e) (quot div))
       (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 18)
       c)))

   (if (eq (car e) (quot rem))
       (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 19)
       c)))


   (if (eq (car e) (quot leq))
       (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 20)
       c)))

   (if (eq (car e) (quot eq))
       (com (car (cdr e)) n (com (car (cdr (cdr e))) n (cons
(quot 14)
       c)))

   (if (eq (car e) (quot car))
       (com (car (cdr e)) n (cons (quot 10) c))

   (if (eq (car e) (quot cdr))
       (com (car (cdr e)) n (cons (quot 11) c))

   (if (eq (car e) (quot atom))
       (com (car (cdr e)) n (cons (quot 12) c))

   (if (eq (car e) (quot cons))
       (com (car (cdr (cdr e))) n (com (car (cdr e)) n (cons
(quot 13)
       c)))

   (if (eq (car e) (quot if))
       (let (com (car (cdr e)) n (cons (quot 8)
                                       (cons then (cons else c))))
         (then com (car (cdr (cdr e))) n (quot (9)))
         (else com (car (cdr (cdr (cdr e)))) n (quot (9)))  )

   (if (eq (car e) (quot lamb))
       (let (cons (quot 3) (cons body c))
         (body com (car (cdr (cdr e))) (cons (car (cdr e)) n)
                                       (quot (5)))  )
```

```
   (if (eq (car e) (quot let))
       (let  (let (cols args n (cons (quot 3)
                     (cons body (cons (quot 4) c))))
                  (body com (car  (cdr e)) m (quot (5))))

          (m cons (vars (cdr (cdr e))) n)
          (args expr (cdr (cdr e))))

    (if (eq (car e) (quot letr))
        (let  (let  (cons (quot 6) (cols args m
                          (cons (quot 3) (cons body (cons
(quot 7) c)
                          ))))
                 (body com (car (cdr e)) m (quot (5))))
           (m cons (vars (cdr (cdr e))) n)
           (args expr (cdr (cdr e))))
    (cols (cdr e) n (com (car e) n (cons (quot 4)
c))))))))))))))))))))
    )


  (cols lamb (e n c)
    (if (eq e (quot nil)) (cons (quot 2) (cons (quot nil) c))

        (cols (cdr e) n (com (car e) n (cons (quot 13) c)))))


  (loca lamb (e n)
    (letr
      (if (memb e (car n)) (cons (quot 0) (posn e (car n)))
          (inca (loca e (cdr n))))


    (memb lamb (e n)
        (if (eq n (quot nil)) (quot f)
        (if (eq e (car n)) (quot t) (memb e (cdr n)))))


    (posn lamb (e n)
      (if (eq e (car n)) (quot 0) (add (quot 1) (posn e (cdr
n)))))
    (inca lamb (l) (cons (add (quot 1) (car l)) (cdr l)))))


  (vars lamb (d)

    (if (eq d (quot nil)) (quot nil)
        (cons (car (car d)) (vars (cdr d)))))


  (expr lamb (d)
    (if (eq d (quot nil)) (quot nil)
        (cons (cdr (car d)) (expr (cdr d)))))
```

Y este es el código objeto .

```
:   código objeto comprobado de liskit
:
(6 2 nil 3 (1 (0 . 0) 2 nil 14 8 (2 nil 9) (2 nil 1 (0 . 0) 11 13
 1 (1 .
 5) 4 1

(0 . 0) 10 11 13 9) 5) 13 3 (1 (0 . 0) 2 nil 14 8 (2 nil 9) (2
 nil 1 (0

13 1 (1 . 4) 4 1 (0 . 0) 10 10 13 9) 5) 13 3 (6 2 nil 3 (1 (0 .
 0) 11 2
1 1 (0

 nil 1
(0 . 1)

11 13 1 (0 . 0) 13 1 (1 . 1) 4 15 9) 5) 13 3 (1 (0 . 1) 2 nil 14
 8 (2 f
9) (1

(0 . 0) 1 (0 . 1) 10 14 8 (2 t 9) (2 nil 1 (0 . 1) 11 13 1 (0 .
 0) 13 1
(1 . 0)
4 9) 9) 5) 13 3 (2 nil 1 (1 . 1) 10 13 1 (1 . 0) 13 1 (0 . 0) 4 8
 (2 nil
 1 (1 .

1) 10 13 1 (1 . 0) 13 1 (0 . 1) 4 2 0 13 9) (2 nil 2 nil 1 (1 .
 1) 11 13
 1 (1 .

0) 13 1 (2 . 3) 4 13 1 (0 . 2) 4 9) 5) 7 5) 13 3 (1 (0 . 0) 2 nil
 14 8 (
1 (0 .

2) 2 nil 13 2 2 13 9) (2 nil 2 nil 1 (0 . 2) 2 13 13 13 1 (0 . 1)
 13 1 (
0 . 0)

10 13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 13 1 (1 . 2) 4 9)
 5) 13 3
 (1 (0

 13 2 1
13 9)

(1 (0 . 0) 10 2 quot 14 8 (1 (0 . 2) 1 (0 . 0) 11 10 13 2 2 13 9)
 (1 (0

10 2  add 14 8 (2 nil 2 nil 1 (0 . 2) 2 15 13 13 1 (0 . 1) 13 1
 (0 . 0)
11 11 10
```

```
13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10
 13 1 (1 . 1) 4 9) (1 (0 .
 0) 10

2 sub 14 8 (2 nil 2 nil 1 (0 . 2) 2 16 13 13 1 (0 . 1) 13 1 (0 .
 0) 11
11 10 13

1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1
 (0 . 0)
 10 2

mul 14 8 (2 nil 2 nil 1 (0 . 2) 2 17 13 13 1 (0 . 1) 13 1 (0 . 0)
 11 11
10 13 1

(1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0
 . 0)
10 2 div

14 8 (2 nil 2 nil 1 (0 . 2) 2 18 13 13 1 (0 . 1) 13 1 (0 . 0) 11
 11 10
13 1 (1

 0) 10 2
 rem 14

8 (2 nil 2 nil 1 (0 . 2) 2 19 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11
 10 13 1
 (1 .

1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0)
 10 2
leq 14 8

(2 nil 2 nil 1 (0 . 2) 2 20 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10
 13 1 (
1 . 1)


4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10
 2 eq
14 8 (2

nil 2 nil 1 (0 . 2) 2 14 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13
 1 (1 .
 1) 4

13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2
 car 14
 8 (2

nil 1 (0 . 2) 2 10 13 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 .
 1) 4 9)
```

```
(1 (0 .

0) 10  2 cdr 14 8 (2 nil 1 (0 . 2) 2 11 13 13 1 (0 . 1) 13 1 (0 .
 0) 11
10 13 1

(1 . 1) 4 9) (1 (0 . 0) 10 2 atom 14 8 (2 nil 1 (0 . 2) 2 12 13
 13 1 (0

1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 cons 14 8 (2
 nil 2 nil
 1 (0 .

2) 2 13 13 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 13 1 (0
 . 1)
13 1 (0

 nil 2 (
9) 13 1

(0 . 1) 13 1 (0 . 0) 11 11 11 10 13 1 (1 . 1) 4 13 2 nil 2 (9) 13
 1 (0 .
 1) 13

1 (0 . 0) 11 11 10 13 1 (1 . 1) 4 13 3 (2 nil 1 (1 . 2) 1 (0 . 1)
 13 1 (
0 . 0)

13 2 8 13 13 1 (1 . 1) 13 1 (1 . 0) 11 10 13 1 (2 . 1) 4 5) 4 9)
 (1 (0 .
 0) 10

2 lamb 14 8 (2 nil 2 nil 2 (5) 13 1 (0 . 1) 1 (0 . 0) 11 10 13 13
 1 (0 .
 0)

11 11 10 13 1 (1 . 1) 4 13 3 (1 (1 . 2) 1 (0 . 0) 13 2 3 13 5) 4
 9) (1 (
0 . 0)

10 2 let 14 8 (2 nil 2 nil 1 (0 . 0) 11 11 13 1 (1 . 5) 4 13 1 (0
 . 1) 2
 nil 1

(0 . 0) 11 11 13 1 (1 . 4) 4 13 13 3 (2 nil 2 nil 2 (5) 13 1 (0 .
 0) 13
1(1 .

0) 11 10 13 1 (2 . 1) 4 13 3 (2 nil 1 (2 . 2) 2 4 13 1 (0 . 0) 13
 2 3 13
 13 1

(2 . 1) 13 1 (1 . 1) 13 1 (3 . 2) 4 5) 4 5) 4 9) (1 (0 . 0) 10 2
 letr 14
 8 (2
```

```
nil 2 nil 1 (0 . 0) 11 11 13 1 (1 . 5) 4 13 1 (0 . 1) 2 nil 1 (0
 . 0) 11
 11 13

1 (1 . 4) 4 13 13 3 (2 nil 2 nil 2 (5) 13 1 (0 . 0) 13 1 (1 . 0)
 11 10
13 1 (2

 0) 13
1 (1 .

1) 13 1 (3 . 2) 4 2 6 13 5) 4 5) 4 9) (2 nil 2 nil 1 (0 . 2) 2 4
 13 13 1
 (0 .

1) 13 1 (0 . 0) 10 13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 13
 1 (1 .
 2) 4
9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 5) 13 3 (2 nil
 2 (4
21) 13 2
nil 13 1 (0 . 0) 13 1 (1 . 1) 4 5) 13 3 (1 (0 . 0) 5) 7 4 21)
```

De esta forma creemos haber mostrado algunas de las capacidades de el compilador Lispkit.

CONCLUSIONES

Una vez logrado el objetivo que nos propusimos al elaborar esta tesis, el cual fue implementar en una máquina convencional una máquina virtual capaz de ejecutar programas escritos en el lenguaje funcional LISPKIT, creemos que la implementación realizada nos permitio mostrar lo siguiente:

-La teoría de los lenguajes funcionales.

-Las principales características de los lenguajes funcionales y sus ventajas respecto a los convencionales.

-La estructura de un lenguaje funcional.

-Como opera un compilador de un lenguaje funcional

-Una forma de implantar en una máquina convencional un lenguaje funcional

Asimismo este trabajo nos permitió :

-Entender mejor la semántica de los lenguajes de cómputo.

-Conocer las conexiones entre un lenguaje y su implementación .

-Conjugar un conjunto de conocimientos de otras áreas de la computación, como son : teoría de programación, compiladores, lenguajes, estructuras de datos y arquitectura de computadoras para realizar el proyecto.

-En general amplió nuestra experiencia, y nos impulsa a profundizar en los temas vistos.

Dentro de las propiedades de los lenguajes funcionales la implementación nos permitio mostrar:

-Uso de funciones simples

-"Composición de funciones" para crear nuevas funciones

-No computar por efecto

-Capacidad de estructuración de datos

-Uso de funciones de alto nivel

-Ser jerárquicos.

-Programas fáciles de entender, y por lo tanto de mantener

Los programas se hacen más rapidamente

Lamentablemente sus características de manejo de memoria hace que su aplicabilidad halla sido limitada, pero se hará más grande conforme la industria provea de mayores cantidades de memoria (lo cual se está logrando).

Los costos de software se incrementan cada vez más y tienden a usar la mayor parte de los costos totales de uso de equipos de cómputo, el precio por bit de memoria cada vez es más barato, y el volumen disponible por máquina cada vez es mayor.

Por lo anterior concluimos que los lenguajes funcionales son una magnífica alternativa para los lenguajes convencionales, y en el futuro los lenguajes funcionales, o algunos similares deben desplazar a los lenguajes convencionales.

MANUAL de USUARIO

Este manual pretende enseñar el manejo del sistema LISPKIT. El cual permite compilar y ejecutar programas realizados en el lenguaje funcional LISPKIT en una máquina simulada llamada SECD.

Para el uso adecuado de este manual es prerequisito el conocer la tesis y el sistema operativo RSX11M versión 3.2 de DEC, así como los programas de utileria EDI y PIP.

1. INTRODUCCION

El primer paso para usar el sistema es definir un programa funcional siguiendo los lineamientos dados en la tesis. Una vez definido se debe teclear el código, creando un archivo por medio del editor de texto EDI, el nombre de este archivo puede ser el que eliga el usuario.

DIAGRAMA DE PROCESO

El diagrama de proceso del sistema se muestra en la figura 1

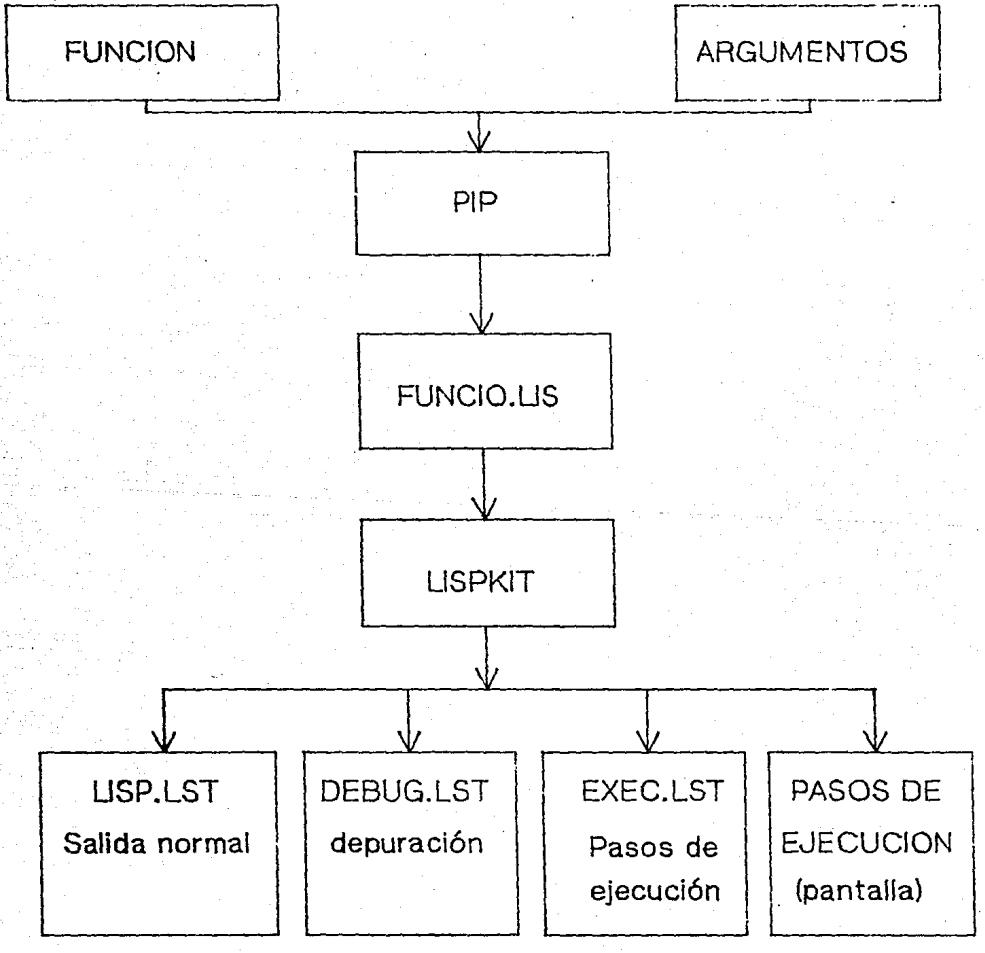


figura A.1 DIAGRAMA DE PROCESO

Por ejemplo, supongamos que se tecleo el siguiente programa para derivar que se muestra en la tesis

```
(letr  diff
  (diff  lamb (e)
     (if (atom e) (if(eq e (quot x)) (quot  1)
                      (quot 0))
     (if (eq (car e) (quot add))
         (let (sum (diff e1) (diff e2))
              (e1 car (cdr e))
              (e2 car (cdr (cdr e))))
     (if (eq (car e) (quot mul))
         (let (sum (prod e1 (diff e2))
                   (prod (diff e1) e2))
              (e1 car (cdr e))
              (e2 car (cdr (cdr e))))
     (quot mal)))))
  (sum   lamb (u v)
         (cons (quot add)
               (cons u (cons v (quot nil)))))
  (prod lamb (u v)
         (cons (quot mul)
               (cons u (cons v (quot nil))))))
```

Y que el archivo que contiene este programa se llame DIFF.LIS Para ejecutar el programa son necesarios dos pasos:

a) compilar el programa

b) ejecutarlo con sus argumentos

Para esto se ha provisto de un archivo de comandos llamado LISPKIT.CMD que se explica a continuación.

1. EL ARCHIVO DE COMANDOS LISPKIT

Por medio de este archivo se compilan y ejecutan las diversas funciones que el usuario programe la forma de ejecutarlo es invocar desde MCR.

```
>@LISKIT
```

En donde se solicita lo siguiente.

1 COMPILAR UNA FUNCION

2 EJECUTAR UNA FUNCION

<CR> TERMINAR

SELECCIONE OPCION :

Responda según sus requerimientos. Si su respuesta es un 1 pase pase a la sección 1.1, si es un 2 pase a la sección 2.1 , si es <CR> se termina el proceso.

1.1 COMPILACION

Este módulo le permite compilar una función, usando el código objeto del compilador que se implantó, cuyo nombre es OBJETO.LIS, que es transparente para el usuario normal. Lo único que se debe proporcionar es el nombre de el programa a compilar que se solicita en la siguiente forma:

NOMBRE DE ARCHIVO A COMPILAR :

Teclee el nombre del archivo donde se halla la función

ejem:

NOMBRE DEL ARCHIVO A COMPILAR : DIFF.LIS

El proceso unirá su archivo con el código objeto del compilador LISPKIT, creando un archivo llamado FUNCIO.LIS, el cual es la entrada al programa LISPKIT. Una vez creado el archivo FUNCIO.LIS, se invoca el programa LISK.TSK, que es el que realiza la ejecución.

Primero se mostrará en pantalla lo sig:.

TAMANO DE MEMORIA A USAR (MAX 8500)(CR> = 8500)

Responda con la cantidad de memoria que considera se usará para compilar su programa, recordando que el código del compilador ocupa 2566 registros al cargarse en memoria (si se va a compilar por primera vez use el máximo de memoria. Para evitar que no se pueda terminar la compilación por falta de memoria).

Una vez que lo halla hecho el programa procede a cargar en memoria el código del compilador y el de su programa, al terminar se despliega lo siguiente :

EL CODIGO OBJETO SE INICIA EN : XXXX
Y OCUPA XXXX REGISTROS

LOS ARGUMENTOS SE INICIAN EN : XXXX
Y OCUPAN XXXX REGISTROS

Y se hará la siguiente pregunta:

DESEA EJECUCION [S/N]

Si su respuesta es negativa pase a la sección 1.3

1.2 EJECUCION

En el módulo de ejecución se procede a ejecutar el código recibido, y se da la posibilidad de obtener un listado de los pasos de la ejecución o en forma opcional un desplegado en pantalla de los mismos.

Al ingresar a ejecución, se despliega:

1	SE GRABAN LOS PASOS DE LA EJECUCION
2	SOLO SE MUESTRAN EN PANTALLA
<CR>	NINGUNA DE LAS ANTERIORES

SELECCIONE OPCION :

Si se elige la opción 1, los pasos de ejecución se grabarán en el archivo EXEC.LST como se muestra en la figura

INST.	S	E	C	D	FF
DUM	8057	8500	8499	8500	8056
LDC	8057	8056	8497	8500	8055
LDF	8055	8056	8493	8500	8054
CONS	8053	8056	8462	8500	8052
LDF	8051	8056	8460	8500	8050
CONS	8049	8056	8429	8500	8048
LDF	8047	8056	8427	8500	8046
CONS	8045	8056	8100	8500	8044
LDF	8043	8056	8098	8500	8042
RAP	8041	8056	8087	8500	8040
LD	8500	8056	8095	8038	8037
RTN	8037	8056	8089	8038	8036
AP	8036	8500	8085	8500	8035

Cuadro 1 ejemplo de resultados en el archivo EXEC.LST

O sea se indica la instrucción que se ejecutó, así como el valor de los 4 registros principales de la máquina SECD y el valor del apuntador a el tope de la PILA, para que si se detecta un error en el programa, se pueda determinar el origen del problema y saber hasta que localidad de memoria se tenía ocupada, esta

salida se origina aunque la terminación termine en forma anormal, indicandose en tal caso hasta el penúltimo paso, ya que el último no se puede detectar.

Si se eligió la opción 2 entonces lo anterior se muestra pero en pantalla, recomendandose solo para depuraciones muy breves que se puedan corregir de inmediato.

Al terminar la ejecución se procede con la sección de salida.

1.2.1 MANEJO DE ERRORES

Si al realizarse la ejecución se halla un error, el sistema avisa indicando que se desea realizar una operación invalida y permite abortar dicha ejecución, aun cuando se pueden continuar con los siguientes módulos para poder determinar el origen del error.

1.3 SALIDA

Al iniciarse el módulo de salida se pregunta:

DESEA SALIDA [S/N]

Si no se elige salida pase a la sección 1.4

Si se elige salida, el sistema pregunta:

LA SALIDA SE INICIA EN : XXXX

DIGITE REGISTRO DESDE EL CUAL SE VACIA O
CR> SI DESEA LA SALIDA NORMAL.

La salida se produce con el formato de EXP-S, colocando puntos solo cuando se hallan dos átomos juntos para que la salida sea más legible, la salida se graba en el archivo LISP.LST, por lo que este archivo contendrá el código objeto del programa que se compilo.

ejem:

```
( 6 2 nil 3 ( 2 nil 1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 13 2
mul
  13 5 ) 13 3 ( 2 nil 1 ( 0 . 1 ) 13 1 ( 0 . 0 ) 13 2
add
   13 5 ) 13 3 ( 1 ( 0 . 0 ) 12 8 ( 1 ( 0 . 0 ) 2 x 14
8 (
     2 1 9 ) ( 2 0 9 ) 9 ) ( 1 ( 0 . 0 ) 10 2 add 14 8
(
```

```
  2 nil  1 ( 0 . 0 )  11 11 10 13 1 ( 0 . 0 )  11 10 13 3 ( 2 nil
      2 nil  1 ( 0 . 1 )  13 1 ( 2 . 0 )  4 13 2 nil  1 ( 0 .
    0 )  13 1 ( 2 . 0 )  4 13 1 ( 2 . 1 )  4 5 )  4 9 )  ( 1 (
    0 . 0 )  10 2 mul  14 8 ( 2 nil  1 ( 0 . 0 )  11 11 10
13 1 (
      0 . 0 )  11 10 13 3 ( 2 nil  2 nil  1 ( 0 . 1 )  13 2
nil
    1 ( 0 . 0 )  13 1 ( 2 . 0 )  4 13 1 ( 2 . 2 )  4 13 2
nil
    2 nil  1 ( 0 . 1 )  13 1 ( 2 . 0 )  4 13 1 ( 0 . 0 )
13
  1 ( 2 . 2 )  4 13 1 ( 2 . 1 )  4 5 )  4 9 )  ( 2 mal  9 )
 9 )
      9 )  5 )  13 3 ( 1 ( 0 . 0 )  5 )  7 4 21 )
```

Cuadro 1.2 ejemplo de código objeto grabado en LISP.LST

Al terminar la producir la salida seleccionada, se pregunta:

DESEA MAS [S/N]

Esta opción permite que se obtenga una salida en formato de EXP-S de cualquier sección de la memoria, y se aconseja su uso cuando se tienen dudas sobre la forma en que el sistema almacena la información, por ejemplo si por error se colocaran TABS en el archivo fuente, estos son interpretados por la máquina como caracteres lo que provacaría errores al momento de ejecutarlo.

Si la respuesta es negativa pase a la sección 1.4

Si la respuesta es afirmativa, se despliega:

DAME DIRECCION INICIAL (CR> SI YA NO) :

Y al darla pide :

DAME DIRECCION FINAL :

De esta forma se podrán listar varias secciones de memoria hasta que se responda con un CR> a la primera pregunta. Con lo que se pasará a la siguiente sección.

1.4 DEPURACION

.El módulo de depuración se realizó para que se pueda observar la

forma en que se almacenan los registros en la memoria de la máquina SECD, la salida de este módulo se graba en el archivo DEBUG.LST, y existen dos posibles formatos de salida, uno en el cual se determina el tipo de registro que se halla almacenado en determinado registro físico de la memoria y se muestra el tipo, y su contenido, como se muestra en la figura.

```
ff=     2 ipoint=     0  fn    99 args    17result     0
reg:     25 ===>cons car =    24 cdr=   100
reg:     24 ===>numerico  el número es:     5
reg:     23 ===>cons car =    22 cdr=    21
reg:     22 ===>numerico  el número es:     7
reg:     21 ===>cons car =    20 cdr=    19
reg:     20 ===>numerico  el número es:     4
reg:     19 ===>cons car =    18 cdr=   100
reg:     18 ===>numerico  el número es:    21
reg:     17 ===>cons car =    16 cdr=   100
reg:     16 ===>cons car =    15 cdr=    14
reg:     15 ===>symbolo  el símbolo es: a
reg:     14 ===>cons car =    13 cdr=    12
reg:     13 ===>symbolo  el símbolo es: b
reg:     12 ===>cons car =    11 cdr=    10
reg:     11 ===>symbolo  el símbolo es: c
reg:     10 ===>cons car =     9 cdr=     8
```

Cuadro 1.3 Salida obtenida en el archivo DEBUG.LST

En esta salida se muestran al principio los valores principales de la ejecución, como son:

ff	apuntador a la cola de disponibles
ipoint	apuntador a la pila para recursividad
fn	apuntador al inicio de la función
args	apuntador al inicio de los argumentos
result	apuntador al inicio del resultado

En el caso que se utiliza, se muestra la salida de pedir el vaciado del conjunto de registros de 25 al 8 indicando sus partes respectivas, como son: el número de registro en memoria, el tipo de registro, y su contenido.

Para esta primera opción el sistema pregunta:

```
NUMERO DE REGISTRO MAYOR (CR> TERMINA):
NUMERO DE REGISTRO MENOR:
```

Responda según necesite.

Otro formato es el llamado OCTAL, en el cual se vacia en formato

octal el contenido de algun registro en particular (eliminando los ceros a la izquierda), esto sirve cuando se ha determinado que algo extraño se ha insertado y se desea saber que es. la pregunta de esta opción se hace en la sig. manera:

DESEA ALGUN REGISTRO EN CODIGO OCTAL [S/N]

Si contesta una S se le pedirá.

NUMERO DE REGISTRO :

Y se regresará a la pregunta anterior hasta que se diga que no.

En la figura 3.2 se muestra la salida que se obtendría al solicitar el vaciado octal de los registros 7899 y 7898. (este vaciado se hace en forma octal agrupando por palabras (2 bytes) y son dos por registro de LIST SPACE, (ya que un registro es de 4 bytes) y se da asimismo el tipo de registro con sus partes respectivas

```
reg:  7899 ===>cons car = 8046 cdr= 7900
     en octal 17556 17334
reg:  7898 ===>cons car = 8183 cdr= 7910
     en octal 17767 17346
```

Cuadro 1.3 formato de salida OCTAL

A continuación el sistema procederá a desplegar dos estadísticas importantes que nos permiten determinar el uso de la memoria en la sig. forma :

NUMERO MAXIMO DE REGISTROS DE LA PILA USADOS : XXXX
NUMERO TOTAL DE REGISTROS RECOLECTADOS : XXXX

Donde la primera cantidad nos indica la cantidad máxima de registros de la pila que se llego a utilizar en un momento dado. Y la segunda la cantidad total de registros que se colectaron en las diversas llamadas que se hallan realizado a el colector de basura.

Y terminará el proceso de ejecución, regresandolo a la sección 1.1.

2.1 EJECUCION

En este módulo del sistema se permite la ejecución de el código objeto de una función (código que se debe obtener con la opción 1), este código se debe hallar en un archivo, y antes de ejecutarlo se debe crear un archivo con EDI conteniendo los argumentos.

Siguiendo con el ejemplo de derivadas podemos suponer que el

archivo LISP.LST que se obtendría en la fase de compilación de la función de derivación, se renombrará como DIFF.LIS y haber capturado un archivo llamado DIF.ARG que tenga lo siguiente:

```
;parámetros originales
(add (add (mul x x) (mul x 2)) 1)
```

Entonces se podría usar la opción de ejecución para obtener el resultado de aplicar la función a los argumentos dados.

La opción solicita lo siguiente:

NOMBRE DEL ARCHIVO CON EL CODIGO OBJETO A EJECUTAR :DIFF.LIS

Responda con el nombre apropiado, después la máquina solicita :

NOMBRE DEL ARCHIVO CON LOS ARGUMENTOS DE LA FUNCION :DIF.ARG

Teclee el nombre del archivo. Con lo anterior se procede a crear el archivo FUNCION.LIS para posteriormente invocar a el programa LISPKIT con lo cual se repiten las mismas preguntas que en la sección 1.1, ya que es el mismo proceso, solo que en el caso de la compilación, el código objeto es el del compilador LISPKIT y los argumentos el el código fuente de la función

La salida obtenida al ejecutar la función que estamos usando se ejemplo se muestran en la figura 2.1

```
; salida obtenida con los parámetros originales
;
( add ( add ( add ( mul x 1 ) ( mul 1 x ) ) ) (
 add (
    mul x 0 ) ( mul 1 2 ) ) ) 0 )
```

figura 2.1 salida obtenida de la función ejemplo

Al terminar la ejecución, el archivo de comandos hace la pregunta inicial para permitir continuar compilando o ejecutando funciones.

MANUAL de SISTEMA

INTRODUCCION

El objetivo de este manual es explicar los componentes de el programa LISPKIT, asi como su estructura, el overlay que se utilizó para crearlo y los archivos necesarios para armar la tarea, para que aquellos que lo deseen modificar lo hagan facilmente.

Son prerequisitos para entenderlo el haber leido la tesis y conocer el sistema operativo RSX11M de una computadora PDP, sobre todo la parte de editor de textos y de armado de tareas TKB.

ESTRUCTURA DEL SISTEMA

El programa del lenguaje LISPKIT se realizó con un compilador FORTRAN IV version 2.5 de DIGITAL EQUIPMENT CORPORATION en una minicomputadora PDP modelo 11/40. y se divide en los siguientes módulos:

principal
entrada de función y argumentos
ejecución
salida
depuración
colector de basura
stack

Cada uno de estos módulos consta de una o varias subrutinas, que se explican en cada uno de los módulos. Dado que hay rutinas que se utilizan en mas de un módulo estas se dividen en dos tipos:

rutinas generales
rutinas particulares

Las rutinas generales son aquellas que se usan en más de un módulo, y por lo tanto se explican antes que los módulos. y las particulares en solo uno de ellos aunque en varias partes de dicho módulo. por lo que se explican junto con el módulo en que se utilizan.

Compilación de las rutinas

Para la compilación de las rutinas se hizo un archivo de comandos indirectos que se muestra a continuación, para usarlo, se digita

```
>@lisk.com
```

El contenido de el archivo lisk.com es el siguiente:

```
princi=princi
        mensaj=mensaj
;       inicio
inicia=inicia
        inisia=inisia
        inisi1=inisi1
        inisi2=inisi2
        getchr=getchr
        getlin=getlin
        scan=scan
        gettok=gettok
        islet=islet
        isdig=isdig
        addst=addst
        getexp=getexp
        getlis=getlis
toint=toint
        decim=decim
;       ejecucion
exec=exec
  exemen=exemen
 lres=lres
  exec1=exec1
cycle=cycle
        mencyc=mencyc
        newgar=newgar
        garcol=garcol
        mark=mark
        markya=markya
        marcar=marcar
        colect=colect
        ocupad=ocupad
        next=next
        ld=ld
        ldc=ldc
        ldf=ldf
        ap=ap
        rtn=rtn
        dum=dum
        rap=rap
        sel=sel
        join=join
        carlis=carlis
        cdrlis=cdrlis
        atom=atom
        consll=consll
        eq=eq
        add=add
        sub=sub
        mul=mul
        div=div
        rem=rem
```

```
        leq=leq
        rutinas generales
        cons=cons
          car=car
          cdr=cdr
          newreg=newreg
          incdr=incdr
        incar=incar
        push=push
        pop=pop
        rsnum=rsnum
        rscons=rscons
        rssymb=rssymb
        docons=docons
        donum=donum
        insimb=insimb
        svalue=svalue
        insval=insval
        rvalue=rvalue
        dosimb=dosimb
;       para putexp
putout=putout
putfin=putfin
putexp=putexp
puttok=puttok
tostr=tostr
addstr=addstr
lenght=lenght
putcar=putcar
putlin=putlin
putnum=putnum
;       salida
salida=salida
```

ARMADO de la tarea

Para el armado de la tarea se utilizaron dos archivos de comandos, uno de ellos contiene las instrucciones para el TKB y se llama lisk.cmd, el otro contiene la descripción del overlay y se llama lisk.odl la forma de "armar" la tarea es la sig.:

```
>tkb @lisk
```

El contenido del archivo LISK.CMD es el siguiente.

```
lisk/cp/fp=lisk/mp
units=5
asg=sy0:1,tl0:2
actfil=2
stack=1024
//
```

El contenido del archivo LISK.ODL es :

```
.root princi-x(inicio,mensaj,ejecu,putex,salid)
```

```
;
;         inicio
;
inicio: .fctr inisia-(inici, inis1, inis2, inis3)
inici:      .fctr inicia-incdr
inis1:      .fctr inisi1-(dosimb-newreg-cdr, insimb,, getchr-getlin))
inis2:      .fctr inisi2-scan-(gettok-(getc, #sdig,, addst,, #slet))
getc:             .fctr getchr-getlin
inis3:      .fctr getexp-gettok-getchr-getlin-#sdig-#slet-addst-(mas)
mas:          .fctr getlis-docons-cdr-newreg-scan-(mas1)
mas1:           .fctr
 (t, donum, insval, dosimb, insimb, push, pop, incar,, incdr))
t:                .fctr toint-decim
;
;           Módulo de ejecución
;
ejecu:   .fctr exec-car-cdr-(e1, e2, res, exemen)
e1:         .fctr exec1-newreg-(dosimb, insimb,, docons,, incar, incdr)
;                 esto de aqui es garbage colector
e2:           .fctr
 cons-iscons-incar-incdr-docons-newgar-garcol-mark-(e20)
e20:              .fctr markya-marcar-colect-ocupad-push-pop-(e22)
e22:          .fctr
 cycle-(1nex, 1men, 11d, 1ldc, 1ldf, 1ap, 1rtn, 1dum,, 1rap,, mor)
mor:          .fctr 1sel, join, 1car, 1cdr, 1atom, consl, 1eq,, 1add,, 1mul,, more1
more1:            .fctr   1sub, 1div, 1rem, 1leq
1nex:                      .fctr next-(ivalue)
1men:                      .fctr mencyc
11d:                       .fctr ld-(ivalue)
1ldc:                      .fctr ldc
1ldf:                      .fctr ldf
1ap:                       .fctr ap
1rtn:                      .fctr rtn
1dum:                      .fctr dum
1rap:                      .fctr rap
1sel:                      .fctr sel-(svalue)
1join:                     .fctr join
1car:                      .fctr carlis
1cdr:                      .fctr cdrlis
1atom:                     .fctr atom-(isnum, #ssymb))
consl:                     .fctr consli
1eq:                       .fctr eq-(issymb, #snum,, svalue,, ivalue))
1add:                      .fctr add-(ivalue, donum,, insval))
1sub:                      .fctr sub-(ivalue, donum,, insval))
1mul:                      .fctr mul-(ivalue, donum,, insval))
1div:                      .fctr div-(ivalue, donum,, insval))
1rem:                      .fctr rem-(ivalue, donum,, insval))
1leq:                      .fctr leq-(ivalue)
;
;         Módulo de edición de salida
;
putex:   .fctr putout-(putexp-(etiq1), putfim-((putlim)))
etiq1:     .fctr
 (push, #ssymb, puttok-(etiq2), #snum, tostr1-(addstr),, mas2))
mas2:             .fctr (putnum-(etiq3), #scons,, car,, cdr,, svalue,, ivalue,, pop))
```

```
etiq2:                          .fctr lenght,etiq3
etiq3:                                  .fctr putcar-putlin
;
;       Modulo de depuracion
;
salid:  .fctr salida-(isnum,iscons,issymb)
        .end
```

REFERENCIA INMEDIATA

CONVENCIONES

x, x1, . . . , xk	representación de átomos.
e, e1, . . . , ek	representación de expresiones-S
s	cualquier expresión-S
T	si se cumple una condición de verdadero, se especifica mediante el átomo de 'T'.
NIL	cuando la condición es falsa se representa con el átomo de NIL, que también es indicador de fin de lista.
(e1 . e2)	representa la unión de dos expresiones formando una sola expresión.

INSTRUCCION	SINTAXIS	DESCRIPCION
x	x	Indica una variable.
QUOTE	(QUOTE s)	Definición de una constante.
ADD	(ADD e1 e2)	Suma de dos expresiones.
SUB	(SUB e1 e2)	Resta de dos expresiones.
MUL	(MUL e1 e2)	Multiplicación de dos exprs.
DIV	(DIV e1 e2)	División de dos exprs.
REM	(REM e1 e2)	Residuo o remanente que surge de la división entre e1 y e2.
CAR	(CAR e)	Se obtiene el 1er. átomo de la expresión o lista 'e'.
CDR	(CDR e)	Se obtiene la lista 'e' menos el 1er. elemento de esa lista.
CONS	(CONS e1 e2)	Se forma una sola lista a partir de las listas e1 y e2.
ATOM	(ATOM e)	Predicado el cual indica si la expresión es un átomo o no (T o NIL).
EQ	(EQ e1 e2)	Predicado que indica si las expresiones e1 y e2 son iguales (T o NIL).
LEQ	(LEQ e1 e2)	Predicado el cual establece si e1 es menor que e2 (T o NIL).
IF	(IF e1 e2 e3)	Forma condicional donde e1 es el predicado; si e1 es 'T' se ejecuta e2 si e1 es 'NIL' se ejecuta e3
LAMBDA	(LAMBDA (x1...xk) e)	Definición de funciones donde x1...xk son los parámetros de la expresión 'e'.
Llamada de Funciones	(e e1...ek)	Cuando es invocada una función las expresiones e1...ek son evaluadas para determinar los argumentos de la función 'e'.
LET	(LET e (x1.e1) ... (xk.ek))	

Definición de bloques locales
donde x1...xk son definidas
como variables locales dentro
de la expresión 'e'.

LETREC (LETREC e (x1 . e1) ... (xk . ek))
Definición de bloques
recursivos donde las x1...xk
pueden estar involucradas tanto
en 'e' como en e1...ek.

BIBLIOGRAFIA

BACKUS, J(1978), Can Programing be liberated from the von Neumman style?. Comm. ACM 21(8), 613-41.

BAJAR, Victoria(1980), Introducción al Sistema Operativo RSX11M para DIGITAL PDP-11, Editorial Limusa, Mexico.

BURGE, W.H.(1975),Recursive Programing Techniques, Addison-Wesley, Reading(Mass.)

DAHL, O., DIJSTRA, E.W. y HOARE, C.A.R.(1972), Structured Programing, Academic Press, London.

DONOVAN, J., Systems Programing, Mcgraw-Hill, New York, N.Y.(1972).

GEAR, J., Computer Organization, Mcgraw-Hill, New York, N.Y.(1970)

GEREZ, V, Grijalva M., El Enfoque de Sistemas, Editorial Limusa, Mex(1978).

GRIES, David, Construcción de Compiladores, Editorial Paraninfo, Madrid(1975).

KNUTH, D.E.(1968), The Art of Computer Programing : Vol. 1 , Fundamental Algorithms, Addison-Wesley, Reading(Mass.).

HENDERSON, P.(1980), Functional Programing application and implementation, Prentice Hall, London.

LANDIN, P.J.(1963), The Mechanical Evaluation of Expresions, Computer Journal, 6(4), 308-20.

LANDIN, P.J.(1965), A correspondence Between Algol 60 and Church's Lambda Calculus, Comm.ACM 8(3), 158-65

LANDIN, P.J.(1966), The Next 700 Programing Lamguages, Comm.ACM 9(3), 157-64

MCCARTHY, J.(1960),Recursive Functions of Symbolic expresions and their computation by Machine, Comm. ACM, 3(4), 184-95.

RALSTON, A(1976),Encyclopedia of Computer Science, Van Nostrand, NY.

TRACTON, Ken(1980),Programer's Guide to LISP, TAB Books, PA.

TREMBLAY, J y SORENSON, P,An introduction to data structures with applications, Mcgraw-Hill, New York, N.Y.(1976).

BIBLIOGRAFIA

WINSTON, P y HORN P.(1981), LISP,ADDISON-WESLEY, Reading(Mass.)

WIRTH, N(1976), Algorithms + Data Structures = Programs, Prentice Hall, Englewood Cliffs (N.J).

E.I.ORGANICK,A.I.FORSYTHE, R.P.PLUMMER Programming Lenguaje Structure

JONHATAN AMSTERDAM Building a Computer in Software, Byte,October-85

RICHARD BRONSON Computer Simulation "What it is and How it's Done" Byte,Marzo-84.

MANUALES:

PDP-11 01/31/45/55 Procesor Handbook 1976,DEC, Maynard(Mass).

Peripheral Handbook 1976

(1975), Introduction to RSX11M, DEC,MAYNARD(MASS.)

(1975), MCR Operations Manual, DEC,MAYNARD(MASS.)

(1975), Utilities Manual, DEC,MAYNARD(MASS.)

(1975), RSX11M/M-PLUS, DEC,MAYNARD(MASS.)

(1975), Task Builder Manual, DEC,MAYNARD(MASS.)

DIGITAL EQUIPMENT(1975), PDP 11 Fortran Language Reference Manual, DEC,MAYNARD(MASS.)

DIGITAL EQUIPMENT(1975), PDP 11 Fortran User's Guide, DEC,MAYNARD(MASS.)