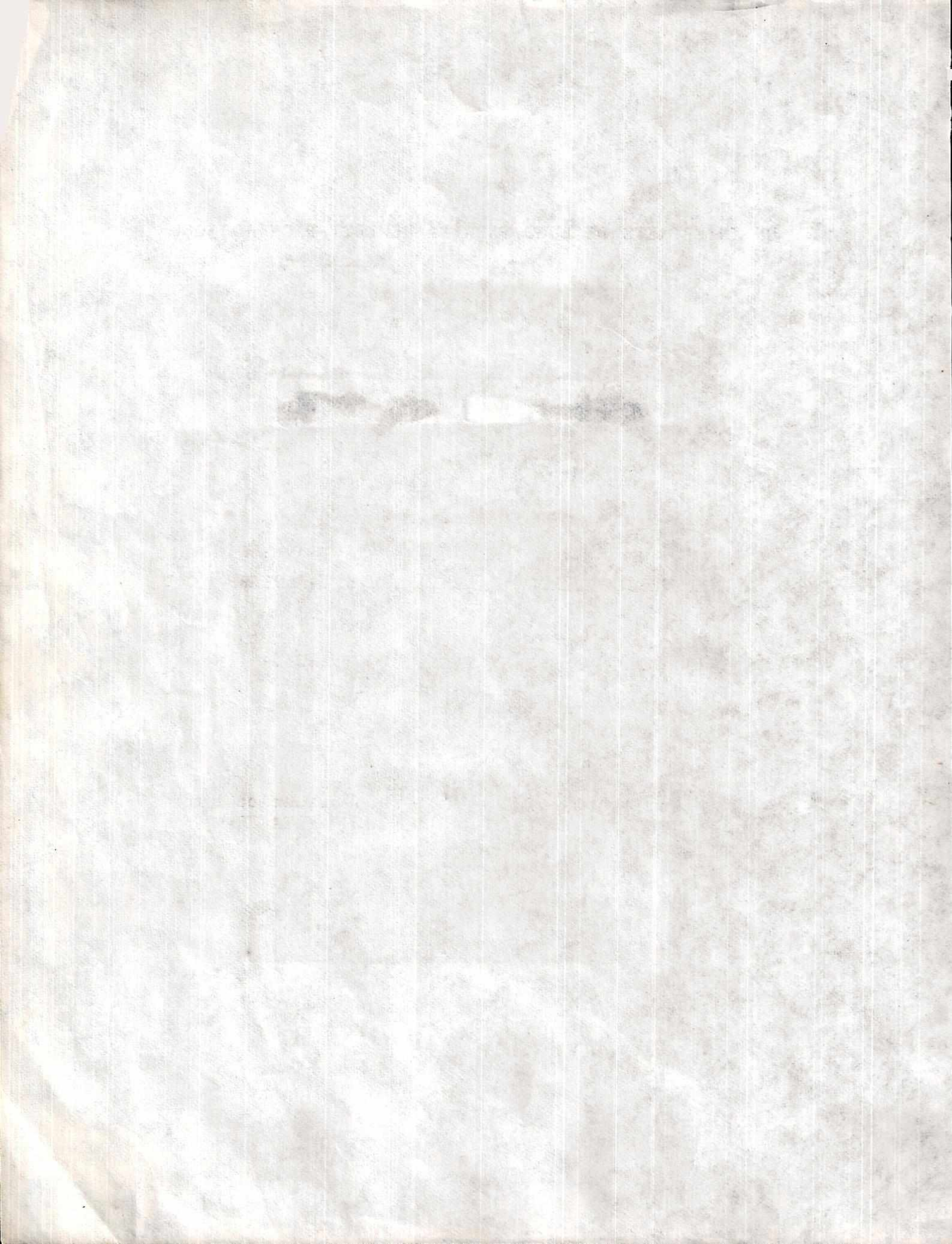
BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA

# UN INTERPRETE DE LOGO PARA LA HEWLETT-PACKARD 3000

BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA



UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

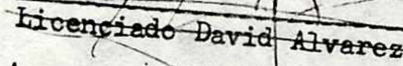
# UN INTERPRETE DE LOGO PARA LA HEWLETT-PACKARD 3000

DOUGLAS LEONEL BARRIOS GONZALEZ.

Trabajo de investigación
presentado para optar al grado académico de
LICENCIADO EN CIENCIAS DE LA COMPUTACION

Guatemala
1985

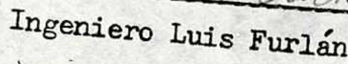
Vo. Bo.

(f) 

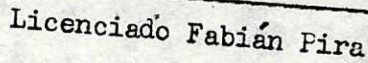
Licenciado David Alvarez

Asesor

Tribunal:

(f) 

Ingeniero Luis Furlán

(f) 

Licenciado Fabián Pira

(f)

Licenciado David Alvarez

Fecha de aprobación: 30 de octubre de 1985.

Páginas

CONTENIDO

0. Prefacio........................................................1
0.1 Lenguajes de alto nivel.........................................1
0.1.1 Necesidad.......................................................1
0.1.2 Origen y evolución..............................................1
0.1.3 Traductores.....................................................3
0.2 La computadora en la educación..................................4
0.3 Objetivos.......................................................4
0.4 Alcances........................................................5
1. Introducción.....................................................7
1.1 Que es el Logo..................................................7
1.1.1 Logo y educación................................................7
1.1.2 Raices del Logo.................................................9
1.1.3 Relación con el Lisp............................................9
1.2 Computadoras virtuales..........................................10
1.2.1 Intérpretes.....................................................11
1.2.2 Lenguajes y computadoras virtuales..............................12
1.2.3 Jerarquía de computadoras.......................................12
1.2.4 Estructura y operación de una computadora.......................14
1.3 Descripción del informe.........................................15
2. Máquina virtual Logo.............................................17
2.2 Estructura de un programa escrito en Logo.......................17
2.3 Datos...........................................................19
2.3.1 Uso de palabras.................................................20
2.4 Operaciones.....................................................21
2.4.1 El mundo de la tortuga..........................................22
2.4.2 Manejo de listas y palabras.....................................23
2.4.3 Manejo de números...............................................24
2.4.4 Operaciones Lógicas.............................................25
2.4.5 Manejo de la lista de propiedades...............................26
2.4.6 Entrada y salida de datos.......................................26
2.4.7 Asignación de valores...........................................27
2.4.8 Traducción y ejecución de datos.................................28
2.5 Control de secuencia............................................28
2.5.1 Recursión.......................................................30
2.6 Control de Datos................................................30
2.6.1 Procedimientos y parámetros.....................................31
2.6.2 Ambiente de referencia y reglas de alcance......................33
2.7 Manejo de memoria...............................................35
2.8 Ambiente de trabajo.............................................36
2.8.1 Guardar y recuperar programas...................................36
2.8.2 Edición.........................................................36
3. Máquina virtual del intérprete Logo..............................39
3.1 Datos...........................................................39

Páginas


3.1.1 Expresiones simbólicas........39
3.1.2 Representación de S-Expr........41
3.1.2.1 Descriptor de un átomo no numérico........42
3.1.3 Lista de propiedades........45
3.1.4 Procedimientos........46
3.1.5 Pila de ejecución........48
3.2 Operaciones........48
3.2.1 Manejo de S-Expr........49
3.2.2 Proposiciones........51
3.2.3 Manejo de lista de propiedades........52
3.2.4 Manejo de la pila de ejecución........52
3.2.5 Ejecución de una instrucción Logo........53
3.3 Control de secuencia........54
3.3.1 Expresiones........54
3.3.2 Procedimientos........55
3.3.3 Regreso de un procedimiento........56
3.3.4 Operaciones de decisión........57
3.3.4.1 Decisión IF........57
3.3.4.2 Decisión TEST........59
3.3.5 Operacion de repetición........60
3.4 Control de datos........62
3.4.1 Ambiente de referencia........62
3.4.2 Transmisión de parámetros reales........63
3.5 El intérprete........64
3.5.1 EVAL_LST........66
3.5.2 APPLY_LST........68
3.5.3 Repetición de nivel superior........70
3.5.4 Ejemplo del funcionamiento del intérprete........70
3.6 Manejo de memoria........72
3.6.1 Memoria libre........73
3.6.2 Memoria utilizada........74
3.6.3 Recolector de basura........75
3.7 Ambiente de trabajo........77
3.7.1 Guardar y recuperar una sesión Logo........77
3.7.2 Edición........79
4. Estructura del intérprete........81
4.1 Análisis de Léxico........81
4.1.1 Estructura del analizador de léxico........83
4.1.2 Interacción con el manejo de tablas........89
4.1.3 Ejemplo de análisis de léxico........90
4.2 Análisis de sintaxis y generación de código........90
4.2.1 Sintaxis del lenguaje Logo........91
4.2.2 Estructura del analizador de sintaxis........93
4.2.3 Ejemplo de análisis de sintaxis........100
4.2.4 Ejemplo y representacion de código producido........104
4.2.5 Interacción con el manejo de tablas........104
4.3 Manejo de tablas........105

Páginas

4.3.1 Instalar un descriptor.............................106
4.3.2 Busqueda en el diccionario central...................107
4.3.3 Instalar funciones propias del Logo..................109
Apéndice A. Sintaxis del lenguaje Logo.......................111
Apéndice B. Un programa Logo.................................113
Apéndice C. Instrucciones Logo...............................117

LISTA DE GRAFICAS


Gráfica Página

3.0.1 Estructura general del Logo....................40
3.1.1 Tipos de nodos....................41
3.1.2 Descriptor de átomo no numérico....................43
3.1.3 Diccionario central....................44
3.1.4 Representación abreviada de S-Exprs....................46
3.1.5.a Procedimiento del usuario....................47
3.1.5.b Procedimiento Logo....................48
3.1.6 Pila de ejecución....................49
3.3.1 Recorrido de S-expr Logo....................55
3.3.2 Expresión IF....................58
3.3.3 Expresion TIZ....................60
3.4.1 Valores asociados....................63
3.5.1 Funcionamiento del intérprete....................72
3.6.1 Lista de nodos disponibles....................73
3.6.2 Obtener un nodo....................74
3.6.3 S-Expr utilizadas....................75
4.0.1 Estructura general del intérprete....................82
4.1.1 Analizador de léxico....................84
4.1.2 Ejemplo de análisis de léxico....................90
4.2.1 Analizador de sintaxis....................94
4.2.2 [PRINT WORD :A B]....................101
4.2.3 [5 + 3 * 4] ....................102
4.2.4 [IF A THEN[C] ELSE[B]]....................103
4.2.5 Arboles resumidos de sintaxis....................105

# 0. Prefacio.

## 0.1 Lenguajes de alto nivel.

### 0.1.1 Necesidad.

La computadora es una herramienta muy útil para la resolución de una gran variedad de problemas. Las personas que la emplean no son sólo técnicos en computación, sino ingenieros, médicos, maestros, investigadores sociales, etc.

Todas estas personas necesitan alguna forma de comunicarse con la computadora, de plantear la solución de problemas en una forma que ella comprenda. Para ello se emplea un lenguaje de programación.

### 0.1.2 Origen y evolución.

En los primeros tiempos de la computación se empleó el lenguaje de máquina. Este consiste en una serie de ceros y unos que indican a la computadora como utilizar sus componentes físicos. Para emplear este lenguaje es necesario conocer muy bien todas las partes y el funcionamiento de la computadora, por eso se requiere que los usuarios sean técnicos de computación.

El escribir un programa en lenguaje de máquina es sumamente tedioso y hacerle modificaciones es casi imposible. Además cada computadora posee su propio lenguaje de máquina.

Ante estas dificultades surgieron los lenguajes de ensamblador, en los que las instrucciones de lenguaje de máquina son representados por códigos mnemónicos. Por ejemplo si 0100010 significa sumar en determinado lenguaje de máquina, en el ensamblador se usa el mnemónico ADD. Este lenguaje libera al usuario de la necesidad de manejar una inmensa cantidad de ceros y unos. Pero aún se debe hablar en términos de registros, buses y demás componentes físicos. Una computadora no puede ejecutar directamente los programas escritos en lenguaje de ensamblador, para ello necesita traducirlo a lenguaje de máquina.

El tener que conocer los detalles físicos de una computadora para programar es una limitación muy grande. Por eso fue necesario desarrollar lenguajes que fueran independientes de la máquina y más naturales de usar para una persona.

En este tipo de lenguajes de alto nivel se pueden expresar los problemas independientemente de la computadora en que se esta trabajando. Asi se puede expresar la instrucción A+B, para sumar dos números, sin conocer los detalles físicos de la máquina, ni la serie de ceros y unos para

indicarla, o el código del ensamblador para realizar esta operación.

Estas herramientas de programación permiten que se emplee la computadora en más actividades. A pesar de esto los problemas que enfrenta un ingeniero no son los mismos que existen para un contador, un médico o el operario de una fábrica. Cada uno de ellos desea expresar las soluciones a sus problemas en lenguajes más idóneos a las disciplinas en que trabajan.

Existe una serie de lenguajes de alto nivel orientados a aplicaciones especificas como: COBOL para los negocios, SIMULA para simulación de modelos, SNOBOL para manejo de cadenas de símbolos, etc.

0.1.3 Traductores.

Todos estos lenguajes de alto nivel permiten a las personas comunicarse con la computadora en una forma más natural al tipo de aplicación especifica a que se dedica.

La computadora sólo puede ejecutar en forma directa los programas escritos en un lenguaje de máquina. Debido a esto es necesario traducir los programas escritos en un lenguaje de alto nivel a un programa equivalente en lenguaje de máquina. La técnica para la construcción de este tipo de

traductores esta bastante desarrollada y existe mucha investigación para mejorarla.

## 0.2 La computadora en la educación.

El empleo de la computadora en educación no es nuevo. Al principio se uso para tareas muy rudimentarias tales como: instrucción programada, hacer preguntas a grupos de alumnos y comprobar respuestas.

Las computadoras pueden ofrecer mucho más. Es posible emplearlas como una herramienta activa del proceso de enseñanza-aprendizaje, para que el alumno descubra y maneje nuevos conocimientos.

Para ello es necesario un lenguaje de alto nivel que permita utilizar adecuadamente la computadora en educación. Este lenguaje es el Logo. En el presente trabajo se explica la forma en que se construye.

## 0.3 Objetivos.

Los objetivos del presente trabajo de graduación son:

1. Construir un intérprete para el lenguaje Logo en la minicomputadora HP-3000 y el sistema operativo MPE.

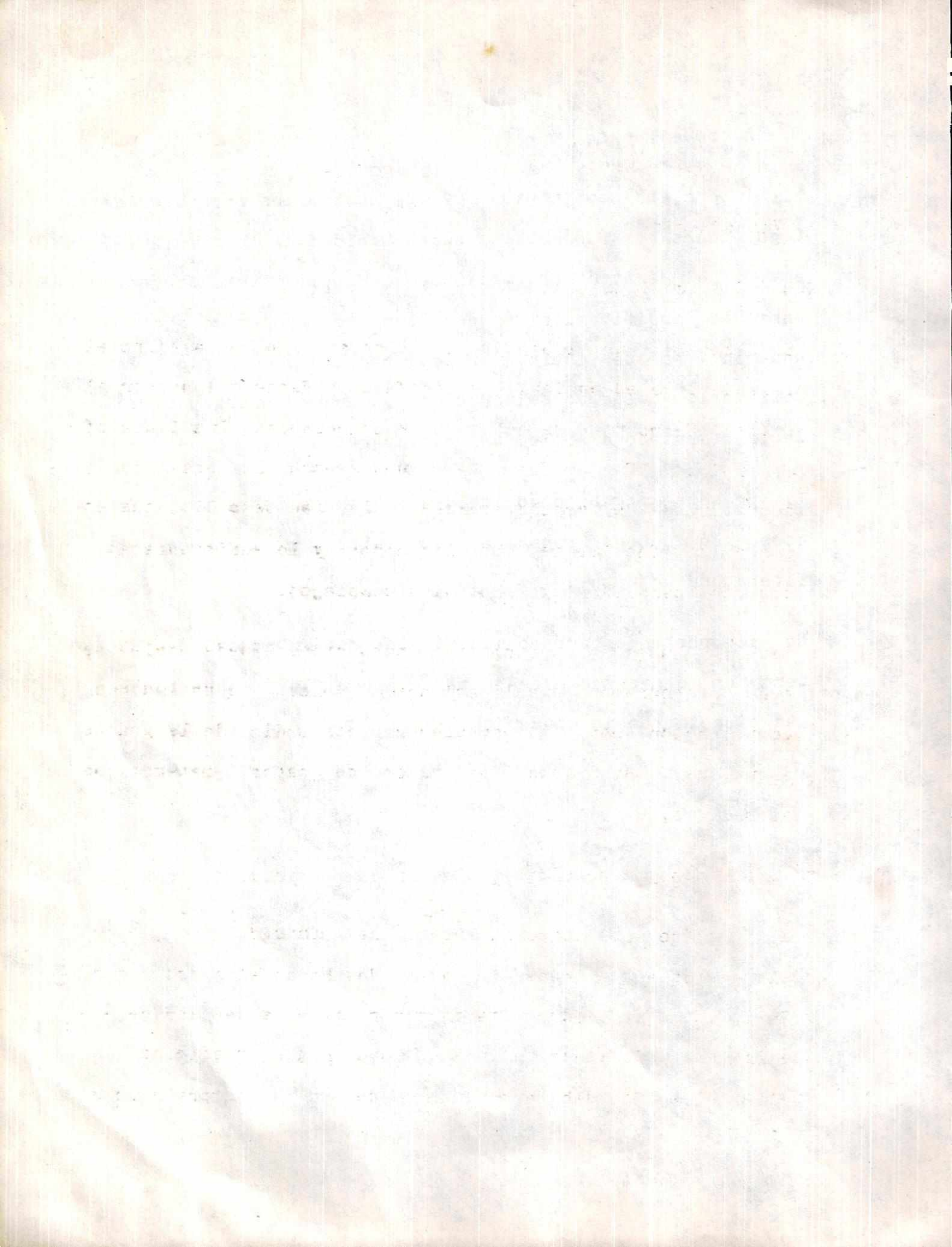
2. Crear un ambiente de trabajo para la utilización y desarrollo de programas en Logo.

## 0.4 Alcances.

El intérprete construido acepta todas las instrucciones Logo necesarias para la construcción de programas. Se emplea el editor proporcionado por el sistema operativo para la modificación o creación de procedimientos del usuario. No se incluye ningun manual de usuario. Para utilizarlo puede consultar el manual de referecia del Logo de la computadora Apple, ver bibliografía.

El presente trabajo no incluye un curso sobre Logo. Para conocerlo deberá consultar los libros mencionados en la literatura adjunta.

No se pretende tampoco mostrar su uso en educación. Este tema es demasiado extenso para incluirlo y corresponde a las ciencias de la educación.



## 1. Introducción.

### 1.1 Que es el Logo.

El Logo es un lenguaje de programación de alto nivel orientado a educación. Fue diseñado y desarrollado por el Doctor Seymour Papert en el Massachusetts Institute of Technology. Es empleado en muchas escuelas primarias y secundarias de varios paises. Es tan sencillo que es aprendido fácilmente por los niños y lo suficientemente poderoso para escribir programas complejos.

Este lenguaje no es exclusivo para niños. Puede emplearse para programar aplicaciones como: compiladores, comunicación con la computadora por medio de lenguajes naturales, consultas a bases de datos, matemáticas simbólicas, etc.

#### 1.1.1 Logo y educación.

El uso de las computadoras se ha difundido mucho, desde Universidades e industrias avanzadas hasta el mismo hogar. Su precio se reduce tanto que a finales de esta década existirá prácticamente una en cada hogar y oficina. Por ello el niño debe conocerla desde su más temprana edad.

Muchas escuelas estan adquiriendo computadoras para todos los niveles de educación.

En la mayoría de los casos la computadora se utiliza para hacer preguntas al niño, comprobar sus respuestas, reforzar los puntos débiles y distribuir información.

En este ambiente el niño es "programado" por la computadora. Aun los juegos educativos que existen son muy rígidos. En un sistema Logo la situación se invierte: el niño, aun en edad muy temprana, programa a la computadora. Al enseñar a la máquina, los niños se embarcan en una exploración para descubrir como piensan ellos mismos. Eso los ayuda mucho para la resolución de problemas de la vida diaria. El niño adquiere la herramienta del algoritmo y el modelo de como trabajaría una computadora, para resolver problemas.

El Logo se convierte en un instrumento muy valioso en diferentes campos de educación, como la enseñanza de: aritmética, geometría, física, gramática, etc.

Existen varias implementaciones de este lenguaje disponibles en el mercado. Pero no existe ninguna para la computadora HP-3000, por lo que se realiza el presente trabajo para su construcción en esa máquina.

1.1.2 Raices del Logo.

Para el diseño y desarrollo del Logo el doctor Seymour Papert se basó principalmente en:

a) Teorías del aprendizaje y conocimiento de Piaget. b) Teorías de inteligencia artificial.

El primer punto corresponde a la Psicología, por lo que no se hablará sobre él.

La inteligencia artificial es un campo de la ciencia de la computación, que trata de las técnicas para extender la capacidad de las computadoras para realizar funciones que son consideradas inteligentes si se efectuan por humanos. Por ejemplo: entender el lenguaje natural, aprender, traducción entre lenguajes naturales, etc.

El presente informe se suscribe a la forma en que se implementa el Logo y no trata de explicar la influencia de los estudios de inteligencia artificial en su desarrollo.

1.1.3 Relación con el Lisp.

Las computadoras pueden hacer cálculos matemáticos a velocidades asombrosas. Pero existen otras aplicaciones en las que se necesita trabajar con el significado expresado por palabras, oraciones y párrafos.

La forma de representar las palabras y oraciones para entender sus relaciones y significado se denomina expresion-simbólica. Es por ello que un lenguaje que las maneje puede resolver problemas como diagnosticos de enfermedades, comunicación con lenguaje natural, búsqueda de información, etc.

El Lisp es uno de los lenguajes más usados para el manejo de expresiones simbólicas. Por ello es muy empleado en los estudios de inteligencia artificial.

El Logo toma muchas ideas y experiencias ganadas por el Lisp en el manejo de expresiones-simbólicas.

1.2 Computadoras virtuales.

Una computadora es un conjunto integrado de algoritmos y estructuras de datos, capaz de guardar y ejecutar programas. Puede ser implementada en dispositivos físicos como circuitos electrónicos, ello forma una computadora en "hardware", sinembargo se puede construir también por medio de programas ejecutados en otra computadora, es decir construirla en "software".

Una computadora parcial o totalmente implementada en "software" o microprogramación es lo que se llama una computadora virtual.

La gran ventaja de un lenguaje de alto nivel es el poder expresar las soluciones a problemas en una forma natural a la aplicación del usuario. Pero este lenguaje no puede ser ejecutado directamente por la computadora ya sea virtual o física. Por eso la implementación de un lenguaje de alto nivel consiste en la construcción de un traductor que toma este lenguaje y lo convierte a un lenguaje comprendido por la computadora física o virtual.

1.2.1 Intérpretes.

Si el traductor acepta programas en lenguaje de alto nivel y produce un programa equivalente en lenguaje de máquina directamente ejecutable por una computadora, física o virtual entonces se denomina compilador.

Otra forma de ejecutar programas escritos en un lenguaje de alto nivel es crear un intérprete. Este simula, por medio programas en otra computadora huésped, una computadora cuyo lenguaje de máquina es el lenguaje de alto nivel. Esto implica construir en el lenguaje de máquina de la computadora huésped los algoritmos y estructuras de datos necesarios para la ejecución del lenguaje de alto nivel.

La implementación del lenguaje Logo se logra construyendo un intérprete. En este informe se explican las estructuras

de datos y algoritmos necesarios para la ejecución de programas escritos en Logo.

### 1.2.2 Lenguajes y computadoras virtuales.

Las estructuras de datos y algoritmos usados en la ejecución de un programa escrito en un lenguaje de alto nivel, estan definidos por la implementación del lenguaje y crean una computadora virtual. El lenguaje de máquina de esta computadora es el programa en forma ejecutable producido por un compilador, o una estructura de datos alternativa, si es interpretado.

La computadora virtual asociada con la implementación de un lenguaje es claramente distinta de la computadora virtual definida al momento de utilizar un programa escrito en ese lenguaje, ya que diferentes implementaciones del mismo lenguaje de alto nivel pueden definir diferentes máquinas virtuales.

### 1.2.3 Jerarquía de computadoras.

El usuario de una computadora no trabaja casi nunca con la computadora física, más bien usa una computadora virtual situada en un nivel más elevado que los elementos y detalles físicos de la computadora huésped.

Los niveles de computadoras que utiliza una persona forman una jerarquía. Esta jerarquía comienza con la computadora física y no tiene ningun final definido, ya que siempre se puede construir y usar una computadora virtual en un nivel superior.

El usuario del Logo trabaja con la computadora virtual definida por este lenguaje. Pero para tener una idea más clara de este informe se describe aqui la posición que ocupa la computadora logo.

a) Computadora física: en el nivel más inferior se encuentra la verdadera computadora física, construida con circuitos electrónicos, orientada al manejo de "stack", etc. Es una minicomputadora HP-3000.

b) Sistema operativo: muy pocas personas programan al nivel de la computadora física. Se necesitan ayudas para manejar archivos, distribución de recursos como memoria, dispositivos de entrada/salida. Estas nuevas facilidades son proporcionadas por el sistema operativo, el cual tambien elimina las operaciones primitivas de entrada/salida para que el usuario no las emplee. El sistema operativo utilizado es el MPE que permite multiprogramación, trabajos en tiempo compartido y en tandas.

c) Máquina virtual Pascal. El inteprete de Logo es implementado usando el lenguaje de propósito general llamado Pascal. Así se utilizó la computadora virtual definida por el Pascal, los programas fueron escritos en lenguaje de máquina y estructuras de datos de la computadora Pascal.

d) Máquina virtual del inteprete Logo. La implementación del intérprete Logo crea una computadora virtual que tiene todas las estructuras y algoritmos capaces de ejecutar programas escritos en Logo. Estas estructuras no son visibles para el programador de Logo.

e) Máquina virtual logo. Es la computadora con la que trabaja el usuario del Logo. Posee todas las estructuras y algoritmos para ejecutar programas escritos para sus aplicaciones específicas.

1.2.4 Estructura y operación de una computadora.

Cuando hablamos de una computadora debemos especificar el nivel en el que estamos trabajando. Pero hay ciertas características que son comunes a todos los niveles de la jerarquía de máquinas. Estos son:

a) datos: la clase de datos que existen disponibles para el usuario como enteros, reales, letras, etc.

b) Operaciones primitivas: las operaciones que puede realizar la computadora como sumar, leer un dato, etc.

c) Secuencia de control: la forma en que se determina el orden de ejecución de las operaciones primitivas.

d) Control de datos: la forma en que la computadora controla que los operandos de las operaciones sean del tipo adecuado.

e) Manejo de almacenamiento: como guarda los datos en memoria, reconoce las instrucciones y programas y puede localizarlos fácilmente.

f) ambiente de operación.

Asi al hablar de una computadora se deberan indicar estos elementos para comprenderla y conocer su operación. No importa el nivel en que estamos trabajando. En el presente informe se hace referencia a estas características para la descripción de las computadoras virtuales creadas.

1.3 Descripción del informe.

En el capítulo 2 se describe la maquina virtual del lenguaje Logo. Contiene los principales elementos del lenguaje empleados por los usuarios en la construcción de programas.

El capitulo 3 explica la máquina virtual definida por la construcción del presente intérprete. Indica la forma en que las ordenes Logo son ejecutadas para producir los resultados esperados por el usuario.

En el capitulo 4 se explica el análisis de léxico y sintaxis que sufre una instrucción Logo para determinar si corresponde o no al lenguaje.

El apendice A contiene la sintáxis del lenguaje en notación BNF.

El apendice B presenta el listado de un programa escrito en Logo.

El apendice C muestra un listado de las instrucciones Logo que puede ejecutar el intérprete que se costruyo.

## 2. Máquina virtual Logo.

Este capitulo muestra una visión general del lenguaje Logo. No pretende ser un curso para su aprendizaje ni mostrar todas las operaciones que posee. Se analizan los datos, operaciones primitivas, control de secuencia, control de datos y manejo de memoria que forman la computadora definida por el lenguaje y que es utilizada por los programadores de Logo.

Cuando sea necesario distinguir entre las instrucciones Logo y los resultados de las mismas, se emplean las siguientes convenciones. El signo de interrogación indica que a continuación el usuario escribe una o más ordenes Logo. El signo de mayor ">" indica que se esta escribiendo un procedimiento. Cuando la linea no esta precedida por alguno de estos signos se trata de una respuesta del Logo.

### 2.2 Estructura de un programa escrito en Logo.

Un programa escrito en el lenguaje Logo esta formado por una serie de procedimientos. Un procedimiento puede ser propio del Logo, es decir una instruccion primitiva, o definido por el usuario.

Un procedimiento puede ser entendido como una o más instrucciones que ejecutan una tarea específica dentro del programa. No existe ninguna estructura física que una los procedimientos. La única relación entre ellos ocurre al momento de ejecutar el programa a través de invocaciones a los mismos.

Ejemplo de programa:

```
?TO PREGUNTAR
>   PRINT [ CUAL ES LA CAPITAL DE GUATEMALA?]
>   MAKE "RESPUESTA READWORD
>   IF :RESPUESTA = "GUATEMALA
>      THEN [APROBAR]
>      ELSE [DESAPROBAR]
> END

? TO APROBAR
>    PRINT [MUY BIEN]
> END
? TO DESAPROBAR
>    PRINT [SU RESPUESTA ES INCORRECTA, -
>            PRUEBE DE NUEVO]
>    PREGUNTAR
>  END

?PREGUNTAR
```

Este programa consta de tres procedimientos definidos por el usuario: PREGUNTAR, APROBAR y DESAPROBAR. Los procedimientos propios del Logo son: TO, PRINT, MAKE, READWORD, IF, =. No importa el orden en que se definan los procedimientos del usuario. El programa pregunta "CUAL ES LA CAPITAL DE GUATEMALA?" Lee la respuesta y la compara con "GUATEMALA". Si es correcta se utiliza el procedimiento APROBAR, en caso contrario se llama a DESAPROBAR. En APROBAR se escribe "MUY BIEN". En desaprobar se escribe "SU RESPUESTA ES INCORRECTA, PRUEBE DE NUEVO" y se repite el procedimiento PREGUNTAR.

2.3 Datos.

El Logo trabaja con dos tipos de datos: palabras y listas. Una palabra esta formada por una serie de símbolos: letras, dígitos y signos de puntuación. Los números son una clase especial de palabras compuestas sólo por dígitos. Ejemplos de palabras son: UNIVERSIDAD, 324, D17MAX.

Una lista esta compuesta por palabras u otras listas. Las listas se muestran encerradas entre [ ]. Por ejemplo: [1 2 3], [ESTA ES UNA [SUBLISTA ] MUY GRANDE].

Los programas del usuario estan formados por estos dos tipos de datos. Esto permite ejecutar datos como si se tratase de programas y modificar programas como si fuesen

datos. Una lista de mucha importancia es la lista de propiedades.

La lista de propiedades esta formada por una serie de nombres de propiedades y valores para ellas. Ejemplo: [ NOMBRE [DOUGLAS BARRIOS] UNIVERSIDAD [DEL VALLE] EDAD 24 ]. Esta lista contiene tres propiedades NOMBRE, UNIVERSIDAD y EDAD y los valores para ellas son: [DOUGLAS BARRIOS], [DEL VALLE], 24. Cada palabra no numérica tiene asociada una lista de propiedades. Los procedimientos son palabras que tiene la propiedad EXPR y como valor las instrucciones que la forman.

Cada palabra no numérica puede tener asociado un valor, es decir se comporta como una variable en otros lenguajes.

2.3.1 Uso de palabras.

Los dos puntos y las comillas modifican el significado de una palabra cuando se colocan inmediatamente delante de ella.

Si una palabra tiene dos puntos adelante significa que se pretende usar el valor que tiene asociado. Si esta precedida por comillas usamos la palabra misma. Cuando ninguno de estos símbolos aparece, la palabra es interpretada como el nombre de un procedimiento.

Por ejemplo la palabra NOMBRE tiene asociado el valor DOUGLAS y además identifica un procedimiento definido en la siguiente forma:

```
?TO NOMBRE
>  OUTPUT [JUAN CHAPIN ]
>END
```

Nótese el resultado del procedimiento PRINT en los siguientes casos. El resultado se muestra inmediatamente abajo de cada PRINT.

```
?PRINT "NOMBRE
        NOMBRE

?PRINT :NOMBRE
       DOUGLAS

?PRINT NOMBRE
       [JUAN CHAPIN]
```

Las palabras que forman una lista se comportan como si existiera comillas adelante de ellas.

2.4 Operaciones.

Todas las operaciones proporcionadas por el Logo estan definidas como procedimientos. A continuación se mencionan las más importantes.

2.4.1 El mundo de la tortuga.

Una parte del lenguaje Logo esta diseñado para proporcionar medios simples para el tratamiento de problemas geométricos. Para ello se utiliza una tortuga que al moverse en la pantalla hace dibujos. Las ordenes para indicar los movimientos que debe ejecutar se denominan "geometria de la tortuga" y la pantalla donde los realiza es el "mundo de la tortuga".

Las instrucciones que forman la geometria de la tortuga son muy simples. Le indican avanzar o retroceder determinadas unidades de espacio o girar un ángulo cualquiera. Con estas ordenes se puede llegar a hacer dibujos muy complejos y visualizar las reglas de la geometria. Entre ellas estan:

FORWARD n : mueve la tortuga n pasos hacia adelante.

BACK n : mueve la tortuga n pasos hacia atras.

LEFT n : gira la tortuga hacia la izquierda, en sentido contrario a las agujas del reloj, n grados.

RIGHT n : gira la tortuga n grados hacia la derecha.

PENUP : indica que la tortuga no debe realizar ningún trazo al moverse.

POS : regresa una lista indicando las coordenadas de la posición de la tortuga.

SETPOS [X Y] : mueve la tortuga al punto (X,Y) del plano cartesiano.

2.4.2 Manejo de listas y palabras.

Hay operaciones para unir, separar y examinar palabras y listas. Algunas de ellas son:

FIRST objeto : proporciona el primer elemento del objeto. FIRST de una lista puede ser una palabra u otra lista. FIRST de una palabra es su primer simbolo. Ejemplo: FIRST [LUNES MARTES JUEVES ] es LUNES FIRST "LUNES es L.

LAST objeto : produce el último elemento del objeto. Ejemplo: LAST [LUNES MARTES JUEVES ] es JUEVES. LAST "JUEVES es S.

LIST objeto1 objeto2: produce una lista cuyos elementos son objeto1 y objeto2. Ejemplo: LIST [ME LLAMO] "DOUGLAS produce: [[ME LLAMO] DOUGLAS]

WORD palabra1 palabra2 : forma una palabra compuesta por la unión de palabra1 y palabra2. Ejemplo: WORD "LU "NES produce "LUNES.

LISTP objeto: regresa: "TRUE si el objeto es una lista, en caso contrario "FALSE. Ejemplo: LISTP "LUNES produce "FALSE. LISTP [LUNES] produce "TRUE.

NUMBERP objeto: regresa: "TRUE si el objeto es una palabra numérica, en caso contrario "FALSE.

EQUALP objeto1 objeto2 : produce "TRUE si los dos obejtos son iguales y "FALSE si no lo son. Ejemplo: EQUALP "LUNES [LUNES] produce "FALSE. EQUALP [LUNES] [LUNES] produce "TRUE.

2.4.3 Manejo de números.

El Logo proporciona procedimientos para efectuar operaciones aritméticas, trigonométricas y raíz cuadrada. Otras instrucciones sirven para comparar si un número es mayor, menor o igual a otro.

Las operaciones de suma, resta, multiplicación y división son proporcionadas en notacion entrefija y prefija. A continuación se muestran en su forma prefija.

SUM a b : produce la suma de a más b. Ejemplo: SUM 5 7 produce 12.

DIFERENCE a b : produce la diferencia a menos b. Ejemplo: DIFERENCE 7 5 produce 2.

BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA



PRODUCT a b : calcula el producto de a por b.

Ejemplo: PRODUCT 4 5 produce 20.

Otras operaciones son:

SQRT a : calcula la raiz cuadrada de a.

RANDOM a : produce un número al azar menor que a y mayor o igual a cero.

INT a: elimina la parte decimal de a.

2.4.4 Operaciones Lógicas.

Una proposición es un procedimiento cuyo resultado puede ser sólo "TRUE (verdadero) o "FALSE (falso). Ejemplo de proposición es el procedimiento LISTP descrito en las operaciones de listas y palabras.

Una operación lógica es la que sirve para manipular proposiciones. Su resultado es también "TRUE o "FALSE. Las tres operaciones más importantes son AND, OR y NOT.

AND prop1 prop2 : produce "TRUE si las dos proposiciones son "TRUE. En caso contrario regresa "FALSE.

OR prop1 prop2 : regresa "FALSE si y sólo si las dos proposiciones son falsas.

NOT prop : produce la negación de la proposición.

BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA

2.4.5 Manejo de la lista de propiedades.

Logo proporciona procedimientos para agregar, borrar y buscar una propiedad específica en la lista de propiedades de una palabra.

Supongamos que la palabra "CLASE tiene asociada la siguiente lista de propiedades : [SALON 217 HORA 8.00 DIAS [LUNES MIERCOLES VIERNES] ]. Las operaciones para manejar una lista de propiedades son:

GPROP pal prop : busca el valor de la propiedad prop asociada con la palabra pal. Ejemplo: GPROP "CLASE "SALON produce 217. GPROP "CLASE "COLOR produce [].

PPROP pal prop val : coloca en la lista asociada con la palabra pal la propiedad llamada prop con valor val. Ejemplo: PPROP "CLASE "MATERIA "COMPUTACION coloca la propiedad "MATERIA con valor "COMPUTACION.

REMPROP pal prop : elimina la propiedad prop de la lista asociada a la palabra pal. Ejemplo: REMPROP "CLASE "DIAS elimina la propiedad "DIAS de la lista asociada a "CLASE.

2.4.6 Entrada y salida de datos.

La entrada y salida de información es muy sencilla ya que no cuenta con instrucciones para asignar formatos. Los principales procedimientos son:

PRINT dat : escribe el dato dat. Si dat es una lista no se imprimen los [ ] mas externos. Ejemplo: PRINT "A A ?PRINT [LUNES MARTES] LUNES MARTES

READLIST : produce como resultado la lista leida.

READCHAR : regresa en forma de palabra el símbolo que se lee.

2.4.7 Asignación de valores.

El procedimiento para crear una relación entre una palabra y un valor es MAKE.

MAKE pal val : crea una relación entre la palabra pal y el valor val. Ejemplo: MAKE "HOY "LUNES MAKE "NOMBRE [JUAN CHAPIN]

Al imprimir el contenido de estas palabras obtenemos:

```
?PRINT :HOY
        LUNES
?PRINT :NOMBRE
        JUAN CHAPIN
```

2.4.8 Traducción y ejecución de datos.

Cada procedimiento del usuario es guardado en forma de lista, es decir con la misma estructura que se representa un dato. Esto permite tratar los procedimientos como si fuesen datos. Asi se pueden ejecutar los datos leidos. Ello se logra mediante la instruccion RUN.

RUN lista1 : traduce y ejecuta las ordenes contenidas en lista1. Ejemplo: ?MAKE "OPERACION [PRINT SUM 5 6] ?RUN :OPERACION 11

2.5 Control de secuencia.

El control de secuencia se refiere a la forma de modificar el orden en que se ejecutan las instrucciones. Existen tres formas principales de control: condicionales, iteración y terminación de un procedimiento.

Las formas condicionales se refieren a la ejecución o no de las instrucciones dependiendo del cumplimiento de una condición. La mas importante de ellas es el procedimiento IF.

IF prop THEN list1 ELSE list2 : si la proposición prop es "TRUE se ejecutan las instrucciones contenida en list1, en caso contrario se efectuan las instrucciones de list2.

EJEMPLO: IF EQUALP :DIA "LUNES
THEN [PRINT [HOY ES LUNES]]
ELSE [PRINT [NO ES LUNES]]

La iteración consiste en repetir una o más veces determinado conjunto de instrucciones. Esto se logra con el procedimiento REPEAT.

REPEAT n lista1 : se repite n veces las instrucciones guardadas en lista1. Ejemplo: REPEAT 4 [FORWARD 5 RIGHT 90]

Los procedimientos pueden regresar o no algun resultado. Por ejemplo el procedimiento SQRT 25 produce como resultado 5. Por el contrario, PRINT "A, no regresa ningun valor, simplemente ejecuta una acción.

La instrucción OUTPUT sirve para terminar un procedimiento e indicar cual es el resultado que produce. Ejemplo:

```
?TO DIA
>    OUTPUT "LUNES
>END
```

Al invocar el procedimiento DIA se obtendrá como resultado "LUNES.

Para terminar un procedimiento que no produce ningun resultado se utiliza STOP. Ejemplo:

```
?TO CONTAR :N
> IF :N = 0 THEN [STOP]
> CONTAR :N - 1
>END
```

2.5.1 Recursión.

Otra forma de repetir una serie de procedimientos es la recursión. Consiste en usar un procedimiento determinado en su propia definición. Ejemplo:

```
TO CIRCULO
   FORWARD 1
   RIGHT 1
   CIRCULO
END
```

Al momento de definir el procedimiento CIRCULO, se emplea el mismo como una de sus instrucciones.

2.6 Control de Datos.

Cada palabra no numérica tiene asociada un valor. En terminología de otros lenguajes, cada palabra es una variable que puede guardar un valor. El problema de control

de datos; consiste en dar el valor correcto de la palabra dependiendo de donde es usada.

Las Variables en Logo son globales, a menos que se indique lo contrario. Esto quiere decir que el valor de una palabra esta disponible en todo el programa. Ejemplo:

```
?TO INICIO
>  MAKE "DATO "UNO
>  SEGUNDO
>END

?TO SEGUNDO
>    PRINT :DATO
>END
```

```
?INICIO
 UNO
```

Asi se observa que el valor asociado con DATO esta disponible para los procedimientos INICIO y SEGUNDO, es decir Todo el programa.

2.6.1 Procedimientos y parámetros.

Los procedimientos tienen datos de entrada que son utilizados por ellos. Ejemplo en la orden FORWARD 5, FORWARD es el procedimiento y 5 es un dato de entrada.

Cuando el usuario define sus procedimientos, en la linea en que le designa su nombre se indican los datos de entrada o parámetros. Ejemplo:

```
?TO SUMAR :N
>   PRINT :N + 1
END
```

SUMAR es el nombre del procedimiento y :N es el parámetro.

Al ejecutarse un procedimiento se crea una relación entre cada parámetro y los valores con que se invocó el procedimiento. Asi al decir:

```
?SUMAR 5
```

se crea una relación entre N y el valor 5.

Si la palabra que actua de parámetro ya tenía un valor asociado antes de usar el procedimiento, entonces esa relación es guardada durante la ejecución del procedimiento. Al finalizar esta, se restaura la antigua relación.

Ejemplo:

```
?TO SUMAR :N
>   MAKE "N  :N + 1
>   PRINT [EL VALOR DE N EN EL PROCEDIMIENTO ES ]
```

```
>   PRINT :N
>END

?MAKE "N 5                                    A.
?SUMAR 7                                      B.
 EL VALOR DE N EN EL PROCEDIMIENTO ES
 8
?PRINT :N                                     C.
 5
```

En la linea A se forma la relación entre N y el valor 5. En la linea B, N es asociada con 7, ya que es parámetro del procedimiento SUMAR. La relación entre N y 5 es guardada. Al finalizar el procedimiento SUMAR se restaura esa relación, como se muestra en la linea C.

2.6.2 Ambiente de referencia y reglas de alcance.

Observemos el siguiente ejemplo:

```
?TO PROC1 :PARM1
 >  PROC2
 >END

?TO PROC2
>   MAKE "PARM1 :PARM1 + 1
>   PRINT [EL VALOR DE PARM1 EN PROC2 ES ]
```

```
>  PRINT :PARM1
>END

?MAKE "PARM1 5                              A.
?PROC1 7                                    I.1
 EL VALOR DE PARM1 EN PROC2 ES              I.2
 8                                          I.3
?PRINT :PARM1                               B
 5
?PROC2                                      II.1
 EL VALOR DE PARM1 EN PROC2 ES              II.2
 6                                          II.3
```

Este ejemplo muestra que valor de PARM1 es el que se emplea en el procedimiento PROC2. En la linea A se crea una asociación entre PARM1 y 5. Al ser invocado el procedimiento PROC1 se crea la asociacion PARM1 y 7, linea I.1, asi el valor que usa PROC2 es 7 e imprime un 8, linea I.3.

Al final de PROC2 se restablece la relación anterior entre PARM1 y 5, por lo que al imprimir su contenido se obtiene el cinco, linea B.

Al emplear independiente PROC2 la asociación que se utiliza es PARM1 y 5, por lo que el valor que imprime es 6, linea II.3.

Como se observa PROC2 siempre emplea el último valor asociado con PARM1.

El conjunto de todas las asociaciones entre cada palabra y valor, en un momento determinado, es lo que constituye el ambiente de referencia. El problema de determinar que ambiente de referencia se debe utilizar en un momento dado constituye una regla de alcance.

En la linea A el ambiente de referencia esta formado por la relacion PARM1 y el valor cinco. En la linea I.1, el ambiente de referencia es PARM1 relacionado con siete, porque PARM1 es un parametro de PROC1. Al finalizar PROC1, linea B, el ambiente es de nuevo PARM1 relacionado con cinco.

Esta forma de determinar el ambiente de referencia, usando la última asociación entre una palabra y su valor, se denomina una regla de alcance dinámica. Esa es la forma empleada por el Logo para determinar el valor que una palabra tiene asociada al momento de ejecutar el programa.

## 2.7 Manejo de memoria.

El usuario de Logo no debe indicar por adelantado el número de palabras o listas que necesitará, no debe reservar la memoria que empleará, tal como sucede en otros lenguajes. El Logo automáticamente proporciona la memoria necesaria

para que se construyan las listas y palabras. El usuario no tiene ningun control sobre el mecanismo de asignación de memoria.

## 2.8 Ambiente de trabajo.

El ambiente de trabajo se refiere a las ayudas que posee el usuario de Logo para desarrollar sus programas. Las principales son el guardar y recuperar programas y datos.

### 2.8.1 Guardar y recuperar programas.

Los programas son guardados en archivos secuenciales en forma de símbolos del alfabeto ASCII. La orden SAVE "ARCHIV guarda todos los procedimientos definidos por el usuario, y los valores y listas de propiedades asociadas con cada palabra, en el archivo llamado ARCHIV.

Para recuperar esta información se emplea el procedimiento LOAD "ARCHIV. Este lee cada linea del archivo llamado ARCHIV, tal como si fuesen escritas directamente en la pantalla.

### 2.8.2 Edición.

Los procedimientos escritos por el usuario pueden ser modificados empleando un editor. Por medio de este se modifican, agregan o eliminan lineas. Al finalizar los cambios, Logo lee las lineas del procedimiento como si

fuesen escritas por el programador en la pantalla. El comando para editar al procedimiento llamado PROC1 es EDIT "PROC1.

# 3. Máquina virtual del intérprete Logo.

El conjunto de algoritmos y estructuras de datos necesarios para que se ejecuten programas escritos en Logo forman la computadora virtual del intérprete. En la figura 3.0.1 se muestra la estructura general del intérprete. En este capítulo se describen la ejecución de instrucciones Logo, el manejo de memoria y la interface con el ambiente.

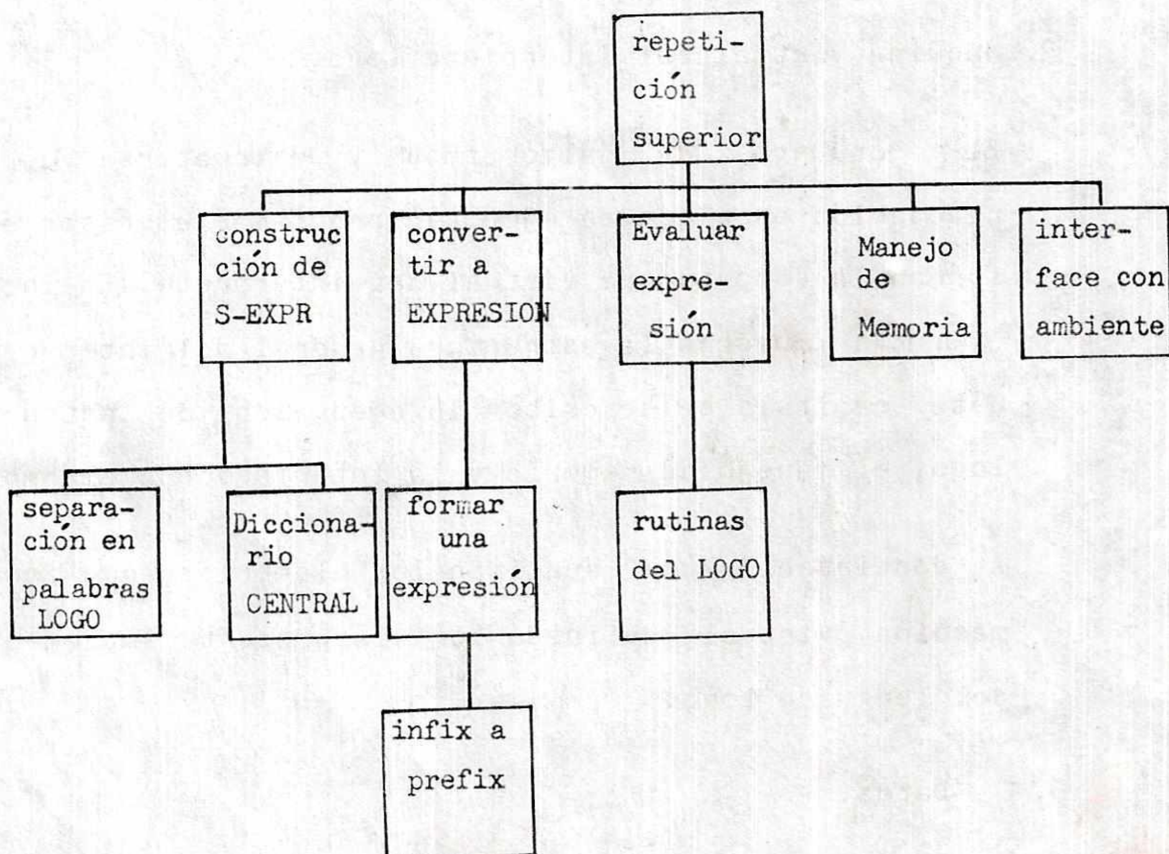
A continuación se analizan los elementos que forman la máquina virtual definida por la presente implementación del lenguaje Logo.

## 3.1 Datos.

### 3.1.1 Expresiones simbólicas.

El intérprete representa tanto los programas como los datos en forma de expresiones simbólicas, de aqui en adelante abreviadas S-Expr. Existen dos tipos de S-Expr: átomos y listas.

Los átomos son los elementos básicos. Estan formados por una serie de no más de quince símbolos alfanuméricos. Si el átomo esta compuesto exclusivamente de dígitos y el punto decimal se dice que es un átomo numérico. Ejemplos de átomos:




3.0.1 Estructura general del Logo

ABC

HOY

LOGO

1.25

Una lista es un conjunto de elementos encerrados entre [ ]. Los elementos que forman la lista pueden ser átomos u otras listas. Ejemplos de S-Expr:

HOY

[EL VALOR DE PI ES 3.1415]

[ [ESTA ES UNA LISTA] [ESTA ES OTRA LISTA]]

3.1.2 Representación de S-Expr.

Los S-Expr son representados en forma de listas encadenas compuestas por nodos. Existen tres formatos de nodo como se muestra en la figura 3.1.1. Todos ellos tienen en común dos campos:

| marca de utilización | tipo de nodo | CAR | CDR |
|---|---|---|---|

a) Nodo de tipo lista.

| marca de utilización | tipo de nodo | tipo de átomo | valor |
|---|---|---|---|

b) Nodo de tipo átomo no numérico.

| marca de utilización | tipo de nodo | valor |
|---|---|---|

c) Nodo de átomo numérico.

3.1.1 Tipos de nodos

a)Marca de utilización: indica si el nodo esta empleado o no.

b)Marca de tipo de nodo: contiene una señal del tipo de nodo: lista, átomo no numérico y átomo numérico.

Los nodos de tipo lista contienen además:

a)CAR: guarda el puntero a un elemento de la lista.

b)CDR: contiene el puntero al siguiente nodo de la lista o la dirección nula, NIL, cuando no existen más elementos.

Los campos adicionales de un nodo para un átomo no numérico son:

a)Tipo de átomo: indica si el átomo esta precedido por comillas, dos puntos o ninguno de estos dos signos.

b)Valor: contiene un puntero a un descriptor.

Los nodos de átomos no numéricos sólo constan de un campo más que contiene su valor.

3.1.2.1 Descriptor de un átomo no numérico.

Los átomos no numéricos tienen un descriptor, ver figura 3.1.2, el cual contiene:

| representació alfanumérica | puntero a lista de propiedades | puntero a lista de valores | puntero a proximo descriptor |
|---|---|---|---|

3.1.2 Descriptor de átomo no numérico

a)Representacion alfanumérica del átomo.

b)Puntero a la lista de propiedades del átomo.

c)Puntero a la lista de valores asociados con el átomo.

d)Puntero al próximo descriptor.

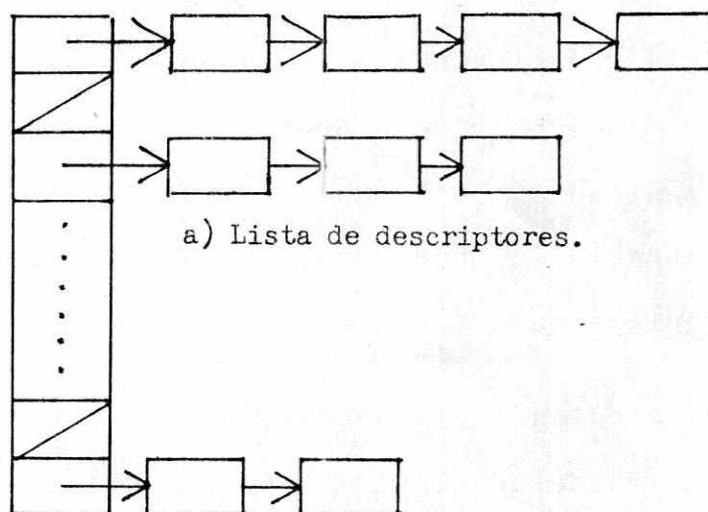
Este descriptor es único para cada átomo no numérico lo cual permite lograr gran ahorro de memoria si el átomo es empleado en más de una lista.

Estos descriptores deben ser localizados con mucha rápidez al momento de crearse una S-Expr. Por eso estan organizados en una estructura denominada Diccionario Central. La búsqueda en el se efectua por medio de técnicas de desmenuzamiento.

El diccionario central esta construido de acuerdo a la técnica de desmenuzamiento abierto, ver figura 3.1.3. Esta formado por una tabla de colisiones, en la cual cada entrada contiene el puntero a una lista encadenada de descriptores que poseen el mismo valor de hash. La función

hash es aplicada a la representación alfanumérica del átomo y proporciona el indice en la tabla de colisiones que contiene el puntero a la lista de descriptores. Esta lista es recorrida hasta encontrar el descriptor deseado. Si la función es eficiente y el número de entradas en la tabla es lo suficientemente grande, se garantiza que la lista de descriptores será de uno o dos elementos. Esto permite efectuar búsquedas muy rápidas.

En la figura 3.1.4 se muestra la representación abreviada de las S-Expr: [A B C D] y [A [16 [A 2.5] ] [B] [21] ]. Para los átomos no numéricos no se representa su nodo, simplemente se escribe el campo del descriptor que



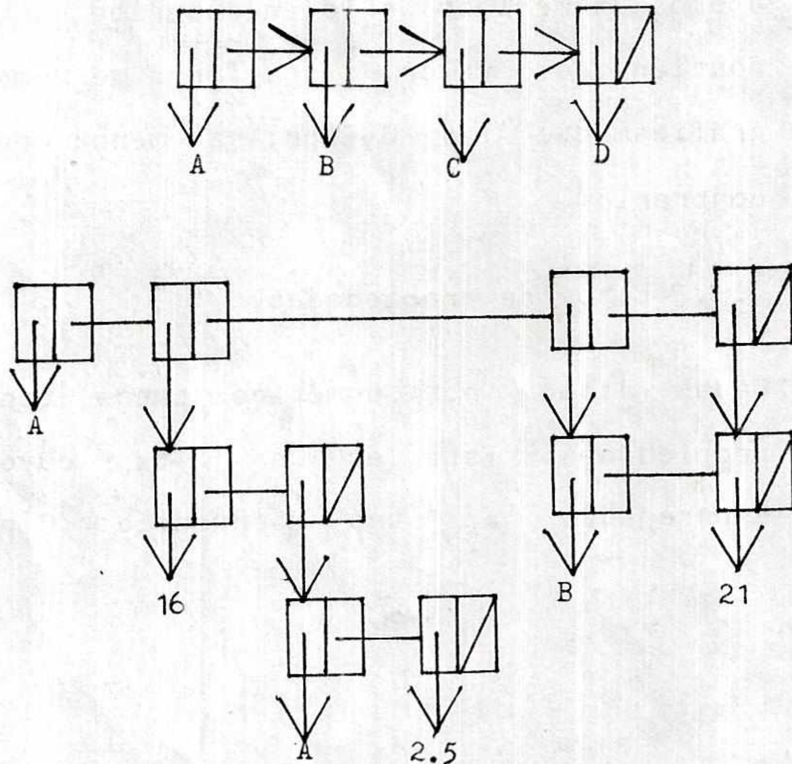

b) Tabla de colisiones

3.1.3 Diccionario central

contiene la representación alfanumérica del mismo. En los átomos numéricos sólo se escribe el campo del nodo que contiene el valor. Esta forma se empleará para mostrar gráficamente las S-Expr, a menos que se indique lo contrario.

3.1.3 Lista de propiedades.

Cada átomo no numérico puede tener una lista de propiedades. Esta es una S-Expr cuyos elementos estan emparejados en una secuencia: [propiedad1 valor1




3. 1. 4 Representación abreviada de S-Exprs

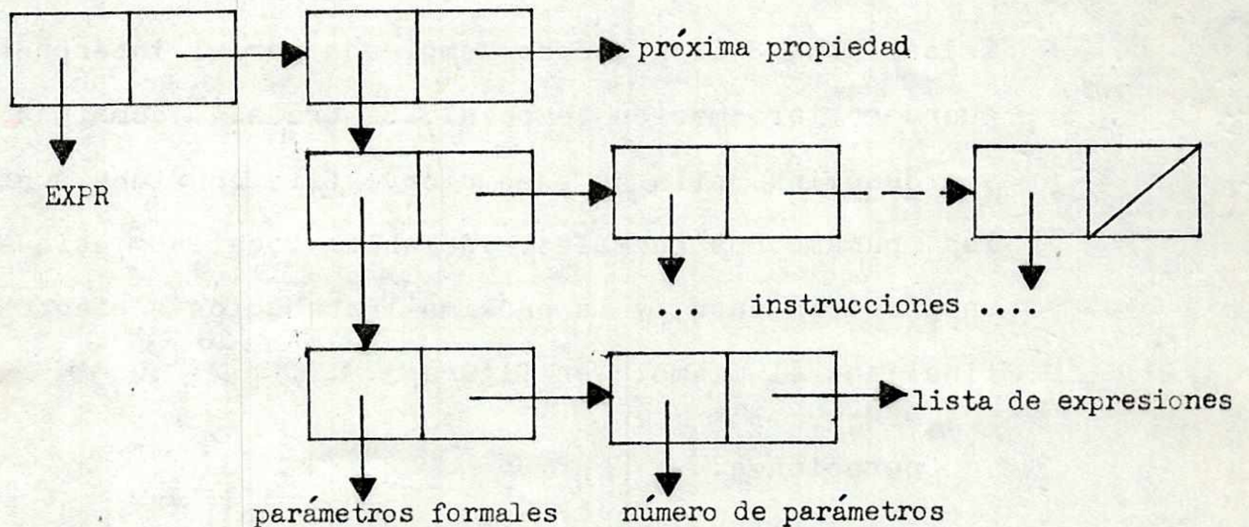
propiedad2 valor2 ... ]. El puntero a esta S-Expr se encuentran en el descriptor del átomo, tal como se muestra en la figura 3. 1. 2.

3. 1. 4 Procedimientos.

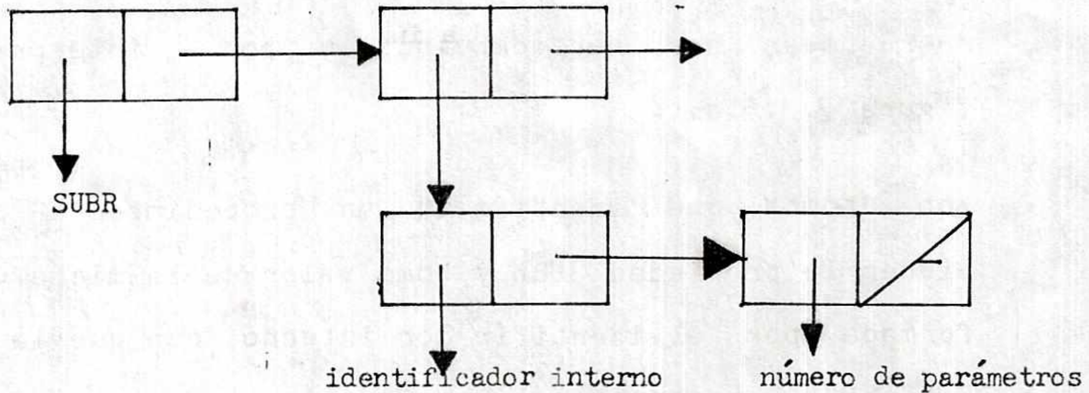
Las instrucciones que forman un procedimiento escrito por el usuario son guardadas en forma de S-Expr y colocadas en la lista de propiedades del átomo que identifica al procedimiento, con la propiedad EXPR. La primera sublista contiene información sobre los parámetros formales y su

número. Contiene además el puntero a las instrucciones reordenadas para ser comprendidas por el intérprete. Ver figura 3.1.5.a.

Los átomos que identifican un procedimiento del Logo tienen la propiedad SUBR y como valor de la misma un lista formada por el identificador interno con que la funcion FUNC_SUBR reconoce al procedimiento y el número de




3.1.5.a Procedimiento del usuario




b) procedimiento del Logo.

3.1.5.b Procedimiento Logo

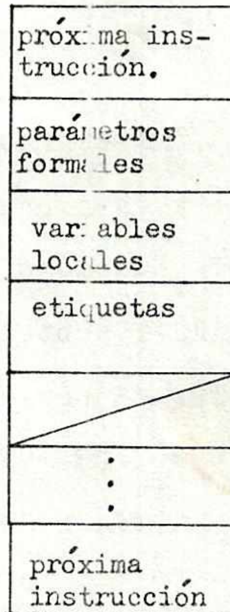
parámetros necesarios. Ver figura 3.1.5.b.

3.1.5 Pila de ejecución.

Existe una S-Expr que es empleada por el intérprete para guardar información temporal. Es tratada como una pila y se denomina pila de ejecución. Ella contiene punteros a los parámetros formales, variables locales y etiquetas de un procedimiento, y la próxima instrucción a efectuarse al finalizar el mismo. Ver figura 3.1.6.

3.2 Operaciones.

A continuación se muestran las operaciones necesarias para que funcione el intérprete.




3.1.6 Pila de ejecución

3.2.1 Manejo de S-Expr.

CAR_LST(s-expr1) : obtiene el primer elemento de la lista s-expr1. Ello es logrado tomando el campo CAR del primer nodo de s-expr1. Ejemplo: CAR_LST([A B C]) es A CAR_LST([ [A B] C]) es [A B]

CDR_LST(s-expr1) : obtiene todos los elementos que siguen al primero de la lista s-expr1. Esto se hace tomando el campo CDR del primer nodo de la s-expr1. El resultado es siempre una lista. Ejemplo: CDR_LST([A B C]) es [B C] CDR_LST([A [B

C]]) es [ [B C]] CDR_LST( CAR_LST([A B C] X])) es [B C]

CONS_LST(s-expr1, s-expr2): forma una s-expr con la unión s-expr1 y s-expr2. S-expr2 debe ser una lista. CONS_LST funciona tomando un nodo y colocando el campo CAR el puntero a s-expr1 y en CDR el puntero a s-expr2. Ejemplo: CONS_LST([A B C],[X Y Z]) es [ [A B C] X Y Z] CONS_LST(CAR_LST([A B]),CDR_LST([A B])) es [A B]

LIST(s-expr1,s-expr2) : forma una lista compuesta por s-expr1 como primer elemento y s-expr2 como segundo. Ejemplo: LIST([A B], [X Y]) es [ [A B] [X Y]]

LIST funciona de la siguiente forma:

a)Obtiene un nodo "A" y un nodo "B".

b)En el campo CAR del nodo "A" se coloca el puntero a s-expr1.

c)En el campo CAR del nodo "B" se coloca el puntero a s-expr2.

d)En CDR del nodo "A" se coloca el puntero al nodo "B".

e)En CDR del nodo "B" se coloca la direccion nula, NIL.

3.2.2 Proposiciones.

Las proposiciones sólo pueden producir como resultado verdadero o falso. La empleadas por el inteprete son:

NULL(s-expr1): es verdadero cuando s-expr1 es la lista nula. Ejemplo: NULL([A B]) es falso. NULL([]) es verdadero.

EQUAL(s-expr1, s-expr2): es verdadero cuando las dos S-Expr son iguales. Ejemplo: EQUAL([A B], [A B]) es verdadero EQUAL([A B], [B C]) es falso.

Las siguientes proposiciones miran el contenido del tipo de marca del nodo:

ATOM_LST(s-expr1): es verdadera cuando s-expr1 es un átomo. Ejemplo: ATOM_LST("A) es verdadero. ATOM_LST([A]) es falso.

NUMBERP(s-expr1): es verdadero cuando s-expr1 es un átomo numérico. Ejemplo: NUMBERP(5) es verdadero. NUMBERP([5]) es falso.

Las proposiciones que indican el contenido del tipo de átomo son:

QUOTED(s-expr1): es verdadero cuando s-expr1 es un átomo no numérico que comienza con comillas. Ejemplo: QUOTED("ABC) es verdadero. QUOTES(:ABC) es falso.

DOTED(s-expr1): es verdadero si s-expr1 es un átomo no numérico que comienza con dos puntos. Ejemplo: DOTED(:ABC) es verdadero. DOTED(XYZ) es falso.

3.2.3 Manejo de lista de propiedades.

GET(s-expr1, prop): obtiene el valor de la propiedad llamada prop colocada en la lista de propiedades asociada con el átomo s-expr1. Ejemplo: si la lista asociada con "LIBRO es [NOMBRE MINDSTORMS AUTOR [SEYMOUR PAPERT] TEMA LOGO], entonces GET("LIBRO,"NOMBRE) es MINDSTORMS.

DEF_LST(s-expr1,prop,val_de_prop): coloca en la lista de propiedades asociada con el átomo s_expr1 una propiedad llamada prop con el valor val_de_prop. Ejemplo: si la lista asociada con "LIBRO es la mostrada en el ejemplo anterior, entonces DEF_LST("LIBRO,"PAGINAS,230) deja la lista: [PAGINAS 230 NOMBRE MINDSTORMS AUTOR [SEYMOUR PAPERT] TEMA LOGO].

3.2.4 Manejo de la pila de ejecución.

Las instrucciones para agregar y eliminar valores de la pila de ejecución son:

PUSH(s-expr1) : coloca s-expr1 al principio de la pila de ejecución. Esta operación es lograda efectuando CONS_LST( s-expr1,[contenido de la pila de ejecución]).

POP : retira el primer elemento de la s-expr formada por la pila de ejecución.

3.2.5 Ejecución de una instrucción Logo.

Existe una operación para ejecutar cada instrucción propia del Logo. Para determinar la operación adecuada se debe usar el identificador interno del procedimiento Logo y la función FUNC_SUBR.

FUNC_SUBR(ident_int,args): determina la operacion u operaciones necesarias para efectuar el procedimiento Logo cuyo identificador interno es ident_int. Args es una S-Expr cuyos elementos son los operandos necesarios para efectuar el procedimiento Logo. Ejemplo: GL2 es el identificador interno de SUM, por lo que:

FUNC_SUBR(GL2,[5 6 1]) produce 12.

El emplear un identificador interno en la funcion FUNC_SUBR permite que exista independencia entre el nombre usado por Logo y el reconocido por el intérprete. Asi se

pueden traducir las instrucciones Logo al español o cualquier otro idioma y no cambiar el intérprete. Los nombres internos son mostrados en el apendice B.

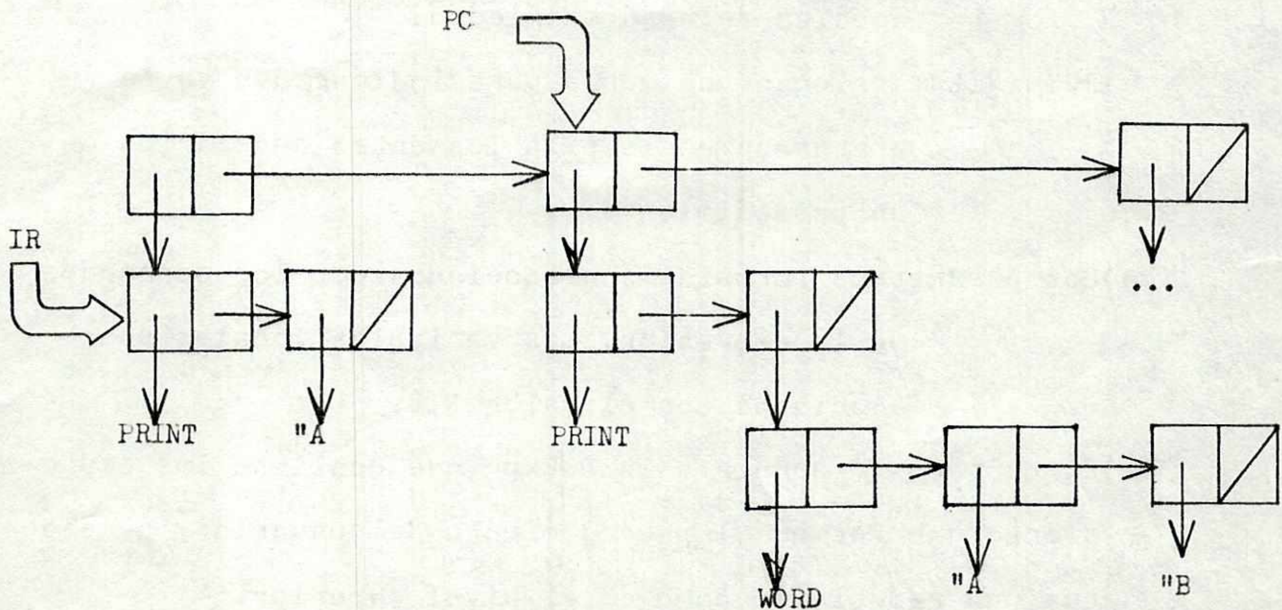
## 3.3 Control de secuencia.

Las instrucciones que comprende el intérprete estan escritas en forma de expresiones. Estas son colocadas en una S-Expr. El intérprete recorre esta S-Expr y ejecuta una por una las las expresiones. Al final de la S-Expr se espera que el usuario ingrese otra serie de instrucciones. Esta repetición es llamada la repetición de nivel superior.

El recorrido de la S-Expr de expresiones es auxiliado por dos punteros. Ver figura 3.3.1. El puntero PC siempre apunta al próximo nodo de la S-Expr. El puntero IR apunta a la expresión que debe ser ejecutada en ese momento.

### 3.3.1 Expresiones.

Una expresión es una S-Expr formada por el nombre de un procedimiento y los operandos que requiere. Estas expresiones se encuentran en notación polaca cambridge. Los operandos pueden ser a su vez otras expresiones. En este caso se ejecutan primero los operandos y luego se efectua el procedimiento con los resultados obtenidos. Ejemplo de expresión: [ PRINT [WORDP "A ] ]. El




3.3.1 Recorrido de S-expr Logo

procedimiento es PRINT y el operando es [WORDP "A] que a su vez es expresión.

3.3.2 Procedimientos.

Si el procedimiento de la expresión que se ejecuta es propio del Logo se emplea FUNC_SUBR para efectuar la operación solicitada con los operandos dados.

Cuando se trata de un procedimieno definido por el usuario se efectuan los siguientes pasos:

a)PUSH(PC): guardar el puntero a la expresión que se debe efecutar al regresar del procedimiento.

b)PUSH(PARMS): guardar los parámetros formales y variables

locales del procedimiento.

c)PUSH(NIL): colocar una marca para indicar que ya se guardaron en la pila los datos necesarios de un procedimiento.

d)Los parámetros formales son asociados con los operandos de la expresión. Las variables locales son asociadas con el valor NIL.

e)El puntero PC apunta a la S-Expr que contiene las expresiones que forman el procedimiento del usuario y se efectua una repetición como en el nivel superior.

3.3.3 Regreso de un procedimiento.

La instrucción STOP termina la ejecución de un procedimiento definido por el usuario. Existe siempre un STOP colocado en el último nodo de la S-Expr que contiene las expresiones del procedimiento. Se puede colocar además en cualquier otro lugar donde se desee terminar la ejecución del procedimiento.

Los efectos de STOP son:

a)Eliminar los nodos de la pila hasta encontrar uno cuyo CAR contenga NIL. Esto deja en la pila la información del procedimiento.

b)Hacer POP: la lista obtenida contiene los parámetros formales y variables locales a los que se les debe asociar con los valores que tenían antes de entrar al procedimiento.

c)Hacer POP: la lista que se obtiene contiene las expresiones que seguían al procedimiento. Por lo que se debe apuntar PC a ellas.

De esta manera se obtiene la proxima expresión a efectuar al finalizar el procedimiento.

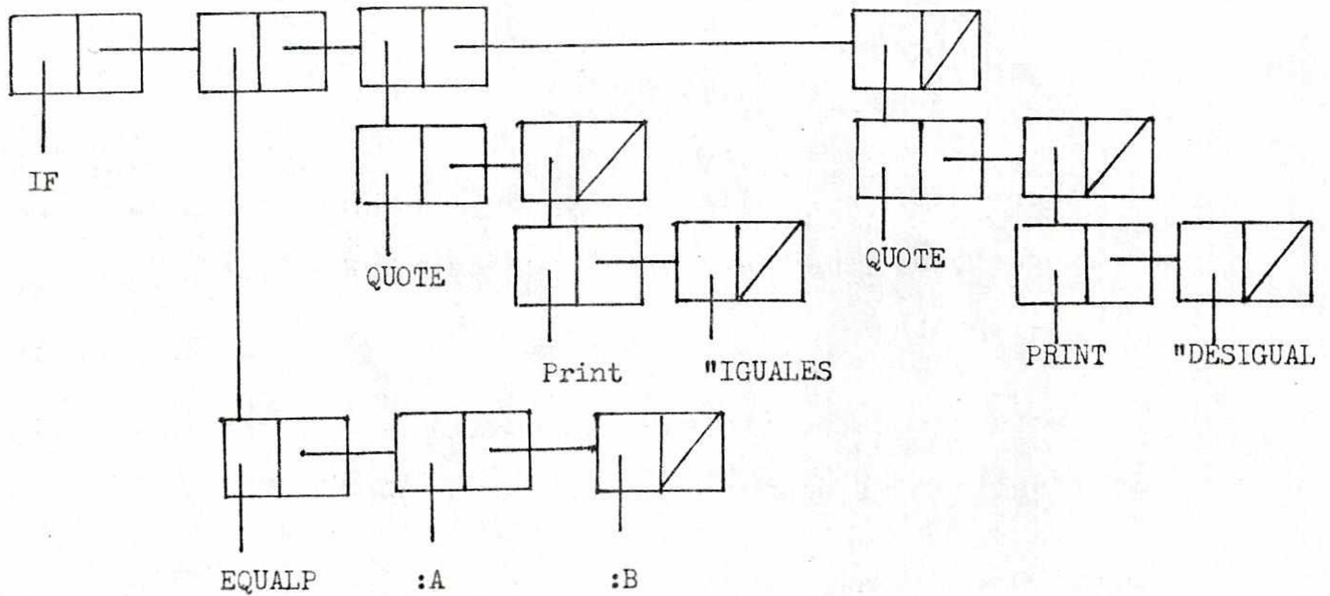
3.3.4 Operaciones de decisión.

Las operaciones de decisión permiten efectuar condicionalmente un grupo de expresiones. Existen dos tipos: decisión IF y decisión TEST.

3.3.4.1 Decisión IF.

Una decisión IF es requerida con una expresión que tiene la forma: [IF [cond-expr] [then-expr] [else-expr] ]. Ver figura 3.3.2. [cond-expr] es un predicado, si produce el valor "TRUE se ejecuta [then-expr] en caso contrario se efectuan las expresiones de [else-expr].

El mecanismo que sigue la decisión IF es:




3.3.2 Expresión IF

a)PUSH(PC): guardar el puntero a la próxima expresión después del IF.

b)Ejecutar [cond-expr].

c)Si [cond-expr] regresa "TRUE

entonces PC apunta a CAR_LST([then-expr])

en caso contrario PC apunta a CAR_LST([else-expr])

La lista señalada por PC es recorrida y ejecutadas todas las expresiones que estan contenidas en ella. Al final de

la lista se recupera de la pila el puntero a la expresión que seguía a IF.

3.3.4.2 Decisión TEST.

La expresión [TEST [cond-expr] ] evalua el predicado contenido en [cond-expr] y asocia el valor resultante a la variable "TEST. Esta es local a cada procedimiento del usuario por lo que siempre esta presente en la lista de variables locales. Ya efectuada la expresión TEST se puede realizar una decisión por medio de las expresiones IFTRUE o IFFALSE.

La operacion VER_TEST( tvalor, [s-expr1]) consulta el valor asociado a la variable TEST, si es igual a tvalor se efectua:

a)PUSH(PC): guardar el puntero a la próxima expresión.

b)PC apunta a [s-expr1].

c)Se ejecutan las expresiones contenidas en s-expr1.

d)POP: se recupera el puntero a la próxima expresión y se guarda en PC.

La expresión [IFTRUE [s-exprtrue] ] es convertida en una llamada a la operación VER_TEST("TRUE, [s-exprtrue] ).

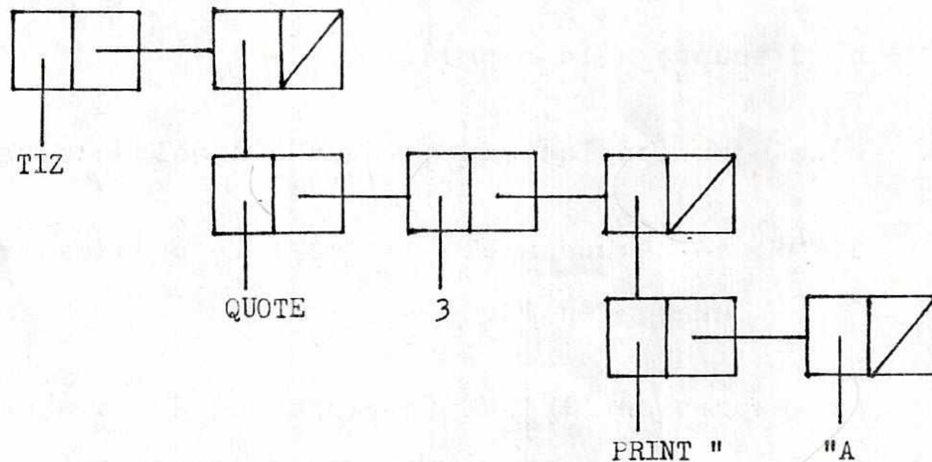
Para la expresión [IFFALSE [s-exprfalse] ] se utiliza la operación VER_TEST("FALSE, [s-exprfalse]).

3.3.5 Operacion de repetición.

La expresión [REPEAT n [s-expr1] ] permite ejecutar n veces las instrucciones contenidas en [s-expr1]. Para ello es empleada la expresión auxiliar [TIZ [ m [s-expr1] ] ], ver figura 3.3.3. En ella m indica el número de veces que hace falta efectuar las expresiones de [s-expr1].

El proceso de repetición es el siguiente:

a)Operar la expresión REPEAT.




3.3.3 Expresion TIZ

Si n es mayor que cero

entonces

PUSH(PC): guardar la proxima expresión después del REPEAT.

m := n - 1;

PC apunta a [s-expr 1]

formar la expresión TIZ

PUSH(TIZ): guardar en la pila la expresión TIZ

Efectuar las expresiones de [s-expr 1] al final recuperar de la pila la expresión TIZ y ejecutarla.

b)Efectuar la expresión TIZ:

Si m es mayor que cero

entonces:

m := m - 1

PUSH(TIZ) : guardar TIZ actualizada

PC apunta a [s-expr 1]

efectuar las expresiones contenidas en [s-expr 1] al final recuperar de la pila la expresión TIZ y regresar al paso b).

en caso contrario:

recuperar a PC de la pila y continuar ejecutando las expresiones apuntadas por PC.

## 3.4 Control de datos.

### 3.4.1 Ambiente de referencia.

Cada átomo no numérico tiene una S-Expr cuyos elementos son los valores que tiene asociados. El valor más reciente esta colocado al principio de la S-Expr y el más antiguo al final. Es decir que la S-Expr es manejada como pila. Ver figura 3.4.1. El puntero a esta S-Expr se encuentra en el descriptor del átomo.

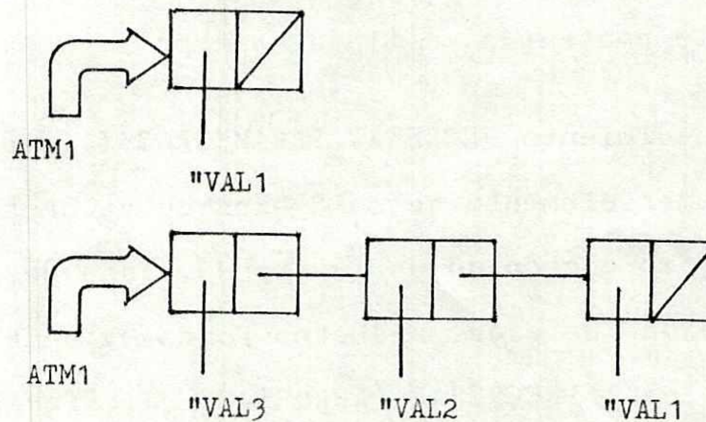
La expresión [MAKE "ATM1 s-expr1] asocia s-expr1 como nuevo valor del átomo "ATM1. El proceso es el siguiente:

a) Se elimina el primer elemento de la s-expr de valores asociados.

b) Se agrega s-expr1 como primer elemento de la s-expr de valores asociados con el átomo.

En la presente implementación para mayor rapidez y ahorro de memoria se modifica el campo CAR del primer nodo de la S-Expr de valores asociados para que apunte a s-expr1. De esta manera se reemplaza el valor más recientemente asociado por el nuevo valor que se desea asociar.

El procedimiento GET_VALUE("ATOM) proporciona el puntero al valor más recientemente asociado al átomo "ATOM.




3.4.1 Valores asociados

3.4.2 Transmisión de parámetros reales.

Los parámetros siempre son transmitidos por valor. Cada parámetro real es evaluado y su resultado es colocado al principio de la S-Expr de valores asociados de su respectivo parámetro formal. La S-Expr nula NIL es colocada al principio de la S-Expr de valores asociados de cada variable local al iniciarse el procedimiento del usuario.

El procedimiento PASAR_PARAMETROS ( [formales-expr], [reales-expr] ) realiza la asociación de cada parámetro

real de [reales-expr] con su respectivo parámetro formal de [formales-expr] como se indico en el párrafo anterior.

ASOC_LOCALES([locales-expr]) asocia el valor NIL a cada variable contenida en [locales-expr].

El procedimiento REGRESAR_PARAMETROS ( [s-expr1] ) elimina el primer elemento de la S-expr de valores asociados para cada átomo contenido en [s-expr1]. Para destruir la última asociación de cada parámetro formal y cada variable local se emplean: REGRESAR_PARAMETROS ( [formales-expr] ) y REGRESAR_PARAMETROS ( [locales-expr]), respectivamente.

Al finalizar el procedimiento del usuario los parámetros formales y variables locales recuperan la última asociación que tenian antes de efectuarse el procedimiento. De esta forma el ambiente de referencia es guardado antes de entrar al procedimiento y es restaurado al fin del mismo.

## 3.5 El intérprete.

El intérprete es el encargado de ejecutar las expresiones que representan instrucciones Logo. Esta formado por dos procedimientos principales: EVAL_LST y APPLY_LST. Ellos interactuan recursivamente para efectuar una expresión.

Los procedimientos necesarios para el funcionamiento del intérprete fueron presentados en el inciso 3.2. Los procedimientos CONS y PARSE son descritos en detalle en el capítulo 4.

EVAL_LST(s-expr1) : clasifica a s-expr1 y decide la forma en que debe ser evaluada. Si s-expr1 es un átomo que comienza con dos puntos busca su asociación más reciente. Si s-expr1 es una lista, se asume que es representa una expresión, se utiliza EVLIS para evaluar cada uno de los operandos y solicita a APPLY_LST que ejecute la expresión con los operandos ya evaluados.

APPLY_LST(func, args): ejecuta una expresión, la cual viene dividida en func que es el procedimiento a efectuar y args, que es una S-Expr que contienen los operandos. Determina si el procedimiento es primitivo del Logo o definido por el usuario para escoger la forma de ejecutarlo.

Además se emplea el procedimiento auxiliar EVLIS( [operand-expr]) el cual evalua cada elemento de [operando-expr] y forma una nueva S-Expr con los resultados de la evaluación. EVLIS funciona de la siguiente forma:

```
SI NULL([operand-expr])
   entonces regresar(NIL)
   de otra manera regresar(
            CONS_LST( EVAL_LST(CAR_LST([operand-expr])),
                      EVLIS(CDR_LST([operand-expr]))
                    )
```

A continuación se explican detalladamente los procedimientos EVAL_LST y APPLY_LST.

3.5.1 EVAL_LST.

El algoritmo de EVAL_LST es:

```
FUNCION EVAL_LST(s-expr)
                          clasificar s-expr
   SI NULL(s-expr)
      ENTONCES REGRESE(NIL)
   SI ATOM_LST(s-expr)    es un átomo
      ENTONCES SI EQUAL(s-expr,"TRUE)
                    ENTONCES REGRESE("TRUE)
               SI EQUAL(s-expr,"FALSE)
                    ENTONCES REGRESE("FALSE)
               SI NUMBERP(s-expr)
                    ENTONCES REGRESE(s-expr)
               SI QUOTED(s-expr)
                    ENTONCES REGRESE(s-expr)
```

```
            SI DOTED(s-expr)  buscar el valor asociado
                      ENTONCES REGRESE(GET_VALUE(s-expr))

    SI EQUAL(CAR_LST(s-expr),"QUOTE)
       ENTONCES REGRESE(CDR_LST(s-expr))
       DE OTRA MANERA REGRESE(APPLY_LST(CAR_LST(s-expr),
                             EVLIS(CDR_LST(s-expr)) )

    FIN-de-EVAL_LST.
```

EVAL_LST clasifica la s-expr en átomos y listas. Si s-expr es un átomo que comienza con dos puntos se da como resultado la última asociación del átomo. Para cualquier otro átomo se regresa como resultado la misma s-expr.

Algunas veces los procedimientos requieren que los operandos de una expresión no sean evaluados, para ello se usa el átomo "QUOTE. Cuando este átomo es el primero de una lista, el resultado de EVAL_LST son los demás elementos sin evaluar. Para cualquier otra lista el primer elemento es interpretado como el nombre del procedimiento a ejecutar y los demás elementos como los operandos del mismo, por lo que el resultado producido por EVAL_LST es la ejecución del procedimiento con los operandos evaluados por medio de APPLY_LST.

### 3.5.2 APPLY_LST.

Las instrucciones que forman APPLY_LST son:

```
FUNCION APPLY_LST(func,args)

            determinar si el procedimiento es primitivo del
            logo o definido por el usuario.

SI GET(func,"SUBR) <> NIL          es primitiva del logo
   ENTONCES REGRESE(FUNC_SUBR(func, args) )

                        es un procedimiento del usuario

def := GET(func, "EXPR)     obtener el procedimiento
parms := CAR_LST( CAR_LST( def))   parámetros formales
PUSH(PC)        guardar el puntero a la proxima expresión
PASAR_PARAMETROS(parms,args)
PUSH(parms)          guardar parámetros formales
PUSH(NIL)            marcar la pila

          estan las instrucciones Logo
          escritas en forma de expresiones?

s→expr := CDR_LST(CDR_LST(CAR_LST(def))) obtener expresión
SI s→expr <> NIL
   ENTONCES PC := s_expr           ya existen expresiones
   DE OTRA MANERA  s→expr := CDR_LST(def) obtener instruc.
                  PC := PARSE(s→expr)  formar expresiones
```

```
        evaluar cada expresión del procedimiento

  REPETIR HASTA QUE EQUAL(CAR_LST(IR),"STOP) O
                    EQUAL(CAR_LST(IR),"OUTPUT)
      IR := CAR_LST(PC)        expresión a evaluar
      PC := CDR_LST(PC)
      t := EVAL_LST(IR)
  FIN_DE_REPETICION.

  SI EQUAL(CAR_LST(IR),"OUTPUT)
     ENTONCES REGRESE(t)
     DE OTRA MANERA REGRESE(NIL)

FIN-DE-APPLY_LST.
```

Si la expresión que debe ejecutar contiene un procedimiento propio del Logo se utiliza FUNC_SUBR para que lo efectue.

En caso contrario es un procedimiento definido por el usuario. Se asocian parámetros formales con los reales y se guarda el puntero a la proxima expresión. Si las instrucciones ya estan en forma de expresiones, PC apunta a la S-Expr que las contiene. Sino debe convertirse las instrucciones a expresiones y PC apuntar al resultado. La S-Expr apuntada por PC es recorrida y evaluada cada expresión hasta encontrar las expresiones OUTPUT o STOP.

3.5.3 Repetición de nivel superior.

El proceso de ejecutar las instrucciones Logo proporcionadas por el usuario es conocida como la repetición de nivel superior. Su operación es la siguiente:

```
Leer las instrucciones Logo en forma de S-Expr.
Apuntar PC a la S-Expr leida.
Repetir MIENTRAS no EQUAL(CAR_LST(PC),"BYE)
   Apuntar PC al resultado de PARSE(PC) formar expresiones
   Repetir MIENTRAS no NULL(PC)
      Apuntar IR a CAR_LST(PC)     expresión a efectuar
      Apuntar PC a CDR_LST(PC)     proxima expresión
      EVAL_LST(IR)
   fin-de-repetición
   Leer instrucciones Logo y ponerlas en forma de S-expr
   Apuntar PC a la S-expr leida.
Fin-de-repetición.
```

3.5.4 Ejemplo del funcionamiento del intérprete.

Aqui se muestran los pasos necesarios paa la ejecución de la instrucción PRINT WORD "A :B. Se supone que el átomo B tiene asociado el valor "XYZ.

I. Las instrucciones son leidas en forma de S-Expr.

[PRINT WORD "A :B]

II. Las instrucciones son transformadas en expresiones. Estas expresiones son colocadas en una S-Expr1 apuntada por PC. IR apunta a la primera expresión a ejecutar:

[ [PRINT [WORD "A :B]] ]

III. Se ejecuta la expresión apuntada por IR: [PRINT [WORD "A :B]]. En la figura 3.5.1 se muestran las llamadas recursivas a EVAL_LST y APPLY_LST para efectuar la expresión. Si el procedimiento mostrado en una linea contiene como argumento a otro procedimiento o una expresión, se debe efectuar el procedimiento que aparece entre los argumentos o ejecutar la expresión en la linea inferior y regresar de nuevo a la linea original para desarrollar el procedimiento con los valores de los argumentos.

La figura esta dividida en tres franjas:

a) Argumentos sin evaluar: los procedimientos empleados contienen uno o mas argumentos reales que deben ser evaluados aun en la linea inferior.

b) Argumentos evaluados: se muestra el mismo procedimiento con los valores reales que deben tener los argumentos.

| Argumentos sin Evaluar | Argumentos Evaluados | Resultado del proceimiento |
|---|---|---|
| EVAL( [PRINT [WORD "A :B]]) | EVAL(NIL) | NIL |
| APPLY(PRINT,EVLIS([[WORD "A :B]]) | APPLY(PRINT,["AXYZ]) | NIL |
| LIST(EVAL([WORD "A :B]),NIL) | LIST("AXYZ,NIL) | ["AXYZ] |
| EVAL([WORD "A :B]) | EVAL("AXYZ) | "AXYZ |
| APPLY(WORD,EVLIS(["A :B])) | APPLY(WORD,["A "XYZ]) | "AXYZ |
| LIST(EVAL("A),EVAL(:B)) | LIST("A,"XYZ) | ["A "XYZ] |
| EVAL("A) | | "A |
| EVAL(:B) | | "XYZ |

3.5.1 Funcionamiento del intérprete

c) Resultados: muestra la respuesta obtenida al efectuar el procedimiento con los argumentos dados. Estos resultados son empleados como valor de argumentos de un nivel superior.

Las operaciones primitivas del Logo son efectuadas por el procedimiento FUNC_SUBR, en el ejemplo no es mostrada al realizar PRINT y WORD.

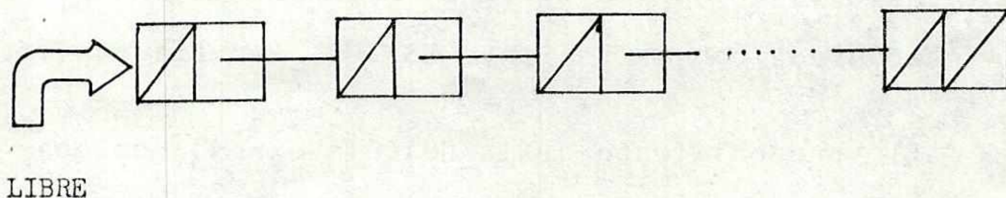
3.6 Manejo de memoria.

La memoria esta compuesta por nodos que son empleados para construir las S-Expr y las listas temporales que son necesarias para la ejecución de las expresiones.

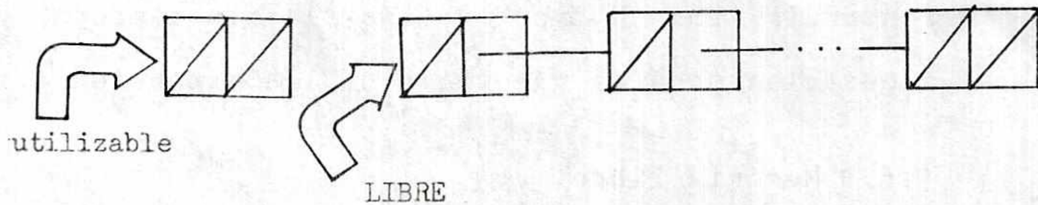
3.6.1 Memoria libre.

Los nodos que se encuentran disponibles forman una S-Expr que esta apuntada por el puntero LIBRE. El campo CAR de cada nodo contiene NIL y el CDR apunta al próximo nodo disponible. Ver figura 3.6.1.

El procedimiento para obtener un nodo es GETNODO. Este toma el nodo apuntado por LIBRE y lo regresa para que sea utilizado. Luego LIBRE debe apuntar al siguiente elemento en la S-Expr de nodos disponibles. Ver figura 3.6.2. Los




3.6.1 Lista de nodos disponibles




3.6.2 Obtener un nodo

pasos son:

Si existen nodos disponibles entonces:

a) apuntar UTILIZABLE al primer nodo apuntado por LIBRE.

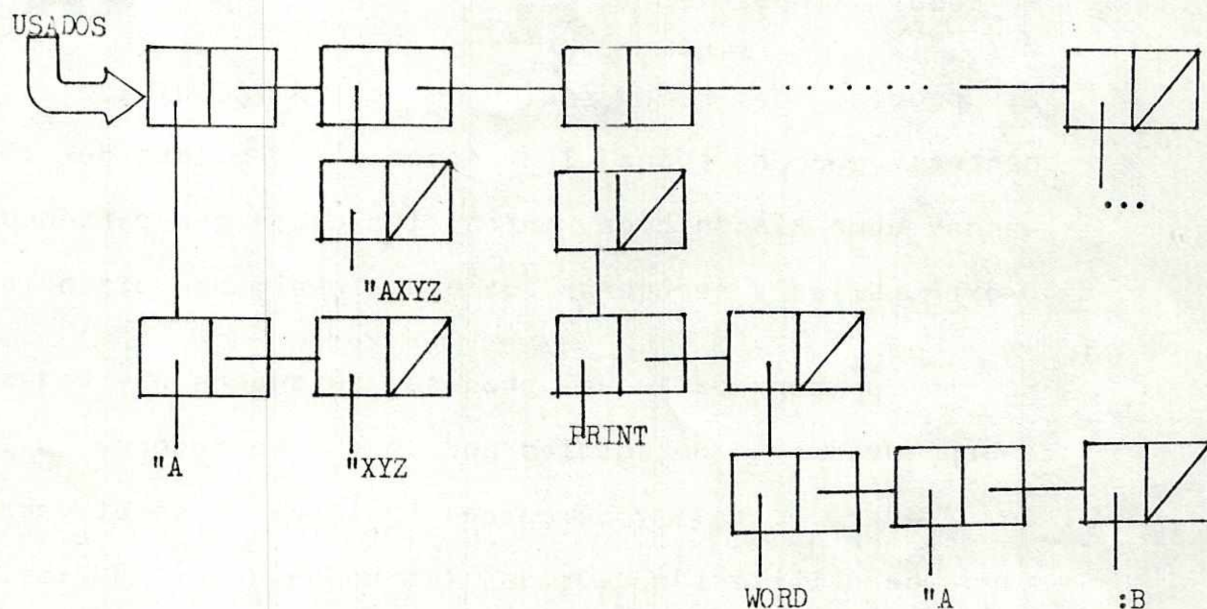
b) LIBRE debe apuntar a CDR_LST(LIBRE).

c) Regresar el nodo apuntado por UTILIZABLE para que sea empleado.

3.6.2 Memoria utilizada.

Todas las S-Expr que son construidas por cualquier operación requerida por el usuario o el intérprete son guardadas en una lista de S-expr utilizadas, la cual esta apuntada por la variable USADOS. Ver figura 3.6.3.

El procedimiento UNIR_USADO(s-expr1) coloca s-expr1 al principio de la s-expr de nodos usados.




3.6.3 S-Expr utilizadas

3.6.3 Recolector de basura.

La lista de nodos disponibles tiene un tamaño finito. Al ser requerido un nuevo nodo, GETNODO investiga si existen nodos en la S-expr de disponibles. Cuando se han agotado todos ellos se emplea el procedimiento de recuperación de basura.

Basura es el término empleado para denotar aquellos nodos que no forman parte de ninguna S-Expr útil. Por ejemplo las instrucciones Logo que son proporcionadas por el usuario en la repetición de nivel superior dejan de ser útiles cuando son convertidas en una lista de expresiones.

Todas las S-expr, ya sean utiles o no, estan en la lista de nodos usados.

El proceso de recuperación de basura se divide en tres partes: marcar todos los nodos en la lista de S-expr usadas como disponibles, marcar los nodos que pertenecen a S-expr utiles y recuperar los nodos realmente disponibles.

En la primera parte del proceso se supone que todas las S-expr usadas no son útiles por lo que se recorre la lista de S-expr usadas y se coloca la letra L en el campo de marca de utilización de todos los nodos de esa S-expr.

Luego es recorrido todo el diccionario central. Para cada descriptor se recorre la lista de valores y la lista de propiedades. En los nodos de cada una de ellas es colocada la letra U en su marca de utilización indicando que forman parte de S-expr útiles. La pila de ejecución es recorrida y se coloca una letra U en la marca de utilización de sus nodos. Asi quedan marcados todos los nodos que forman parte de expresiones útiles.

En la última parte del proceso se recorre de nuevo la lista de S-expr usadas y todos aquellos nodos que poseen aun la letra L en su marca de utilización son agregados al principio de la lista de nodos disponibles.

La recuperación de basura permite volver a emplear los nodos no útiles en la construcción de nuevas S-Expr. Consume tiempo de ejecución pero ocurre esporádicamente.

3.7 Ambiente de trabajo.

Las ayudas proporcionadas permiten:

a)Guardar y recuperar los valores y lista de propiedades asociadas con los átomos no numéricos.

b)Guardar y recuperar los procedimientos definidos por el usuario.

c)Impresión de los procedimientos definidos por el usuario, en la impresora del sistema.

3.7.1 Guardar y recuperar una sesión Logo.

El trabajo desarrollado en una sesión con el intérprete Logo esta formado por los procedimientos que escribe el usuario, y los valores y listas de propiedades que define para asociarlos con los átomos no numéricos. Este trabajo puede ser guardado y luego recuperado.

Existen dos archivos que permiten la comunicación con el mundo exterior al intérprete: ENTRADA y SALIDA. Para leer las instrucciones Logo se emplea ENTRADA. Para escribir la representación de alguna S-Expr se usa SALIDA.

Todas las instrucciones Logo proporcionadas por el usuario provienen del archivo ENTRADA. Al inicio de la sesión este archivo es asociado con el archivo INPUT, por lo que las instrucciones escritas en la pantalla son transmitidas al intérprete.

La orden LOAD "ARCH asocia el archivo llamado ARCH con ENTRADA. De esa manera el intérprete lee las instrucciones Logo contenidas en ARCH. Al encontrar el final de este archivo, ENTRADA es asociado de nuevo con INPUT.

La orden Logo SAVE "ARCH asocia el archivo llamado ARCH a SALIDA y luego efectua el procedimiento VAL_Y_PROP, el cual efectua la orden PRINT para cada valor y lista de propiedades existentes en el diccionario central. Tambien son guardados las definiciones de todos los procedimientos definidos por el usuario. Al finalizar se restaura la relación de SALIDA con el archivo OUTPUT.

La orden WRITE "PROC asocia SALIDA con la impresora y produce una copia del procedimiento llamado PROC. La orden PRINTER "ON asocia SALIDA con la impresora, por lo que todos los resultados que aparecerian en la pantalla son producidos en la impresora del sistema. Para restaurar la relación de SALIDA y OUTPUT se debe escribir la orden PRINTER "OFF.

3.7.2 Edición.

La definición de cada procedimiento del usuario es guardada en un archivo indexado cuya llave esta compuesta por el nombre del procedimiento. En este archivo se muestran las instrucciones del procedimiento tal como son escritas.

Para modificar estas instrucciones se utiliza la orden Logo EDIT "PROC. Al efectuarla se recuperan los registros cuya llave es PROC y se trasladan a un archivo secuencial. Este archivo es utilizado por medio del editor que proporciona el sistema operativo para efectuar los cambios deseados. Al terminar estos, el archivo secuencial es asociado con ENTRADA y se produce el mismo proceso de la orden LOAD.

# 4. Estructura del intérprete.

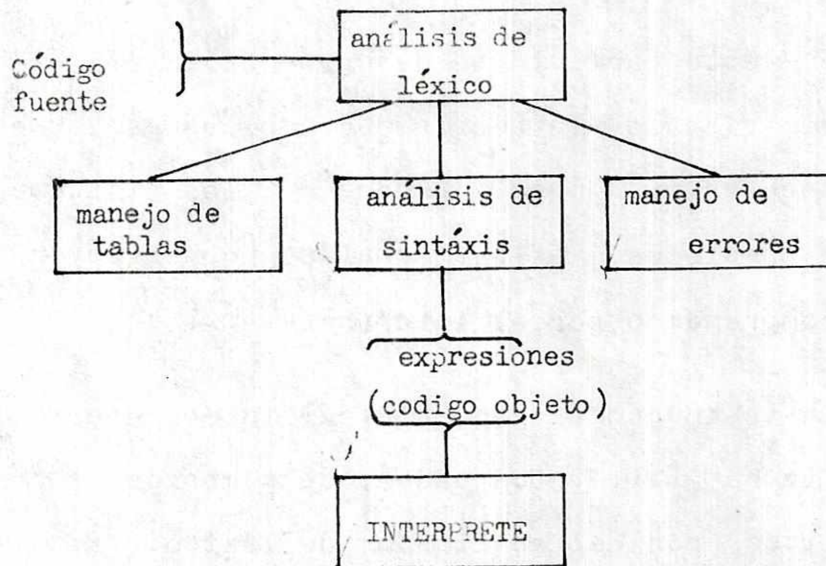
En este capítulo se describe al intérprete desde el punto de vista de la teoría de compiladores. Muestra las partes necesarias para traducir las instrucciones Logo a expresiones, las cuales constituyen el lenguaje comprendido por el intérprete.

La estructura general del intérprete es mostrada en la figura 4.0.1. Una cadena de símbolos es convertida en una lista por el analizador de léxico. Esta cadena sufre un análisis de sintaxis para verificar que corresponde al lenguaje Logo y al mismo tiempo es construido un árbol que representa la expresión equivalente.

En cada etapa se usa un módulo para manejar los errores y una tabla central que contiene los atributos de los átomos no numéricos que son utilizados.

## 4.1 Análisis de Léxico.

El analizador de léxico lee una cadena de símbolos y los agrupa en elementos de sintaxis. Estos elementos son identificadores, delimitadores, operadores de relación, operadores binarios y números.




4.0.1 Estructura general del intérprete

Las expresiones regulares que describen los elementos de sintaxis son:

identificador = caracter

comillas_ident = "caracter

dospuntos_ident = :caracter

caracter = digito | letra | caracter_especial

letra = A|B|...|Z|a|b|...|z

digito = 0|1|...|9

caracter_especial = "|^|ñ|&|'|,|.|\|:|;|@|~

número = decimal | exponencial

decimal = signo entero.entero | signo entero. | .entero

```
signo = - | nada
entero = digito
exponencial = (decimal | signo entero ) (E|N) entero
oper_relacion = < | > | =
oper_binario = + | - | * | /
inicio_lista = [
fin_lista = ]
delimitador = oper_relacion | oper_binario | blanco |
              inicio_lista | fin_lista | ( | )
```

Los elementos de sintaxis sirven para construir la s_expr que es el producto del analizador de léxico. En esta fase se eliminan blancos, se convierten las letras a mayúsculas. Los descriptores de los átomos no numéricos son ingresados al diccionario central. Los átomos numéricos son convertidos a números reales.

4.1.1 Estructura del analizador de léxico.

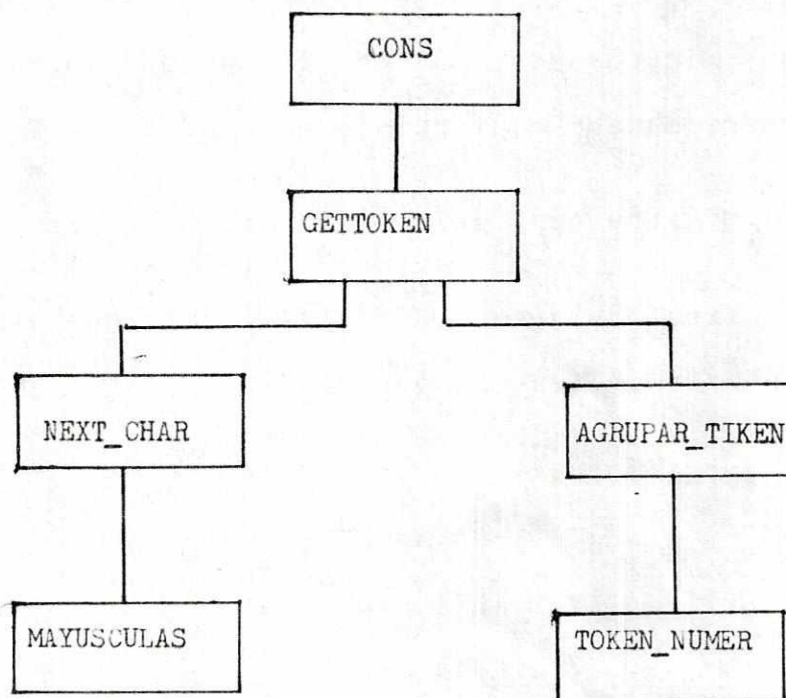
El tipo de dato TOKEN sirve para guardar un elemento de sintaxis. Tiene la forma:

```
TYPE  TOKEN = RECORD
          REP : SIMBOLOS;
          VAL_ALFA : STRING[15];
          VAL_NUM : REAL;
      END;
```

REP contiene la clase del elemento: identificador, delimitador, oper_relación, oper_binario, número, dospuntos_ident, comillas_ident. VAL_ALFA tiene la serie de símbolos alfanuméricos que forman al elemento y VAL_NUM es la representación numérica de un átomo numérico.

La variable NEXT_TOKEN contiene el elemento que ha obtenido el analizador de léxico.

La estructura del analizador de léxico es mostrada en la figura 4.1.1




4.1.1 Analizador de léxico

El procedimiento CONS construye una lista a partir de los elementos de sintaxis contenidos en la cadena de entrada. El puntero a esta lista es guardado en el parametro LISTA1. CONS funciona asi:

```
PROCEDURE CONS(LISTA1);
BEGIN
  GETTOKEN;  { OBTENER EL ELEMENTO DE SINTAXIS }
  CASE NEXT_TOKEN.REP OF
    INICIO_LISTA : BEGIN
              OBTENERNODO(NODOA);
              LISTA1 := NODOA;
              CONS(NODOA^.CAR);
              CONS(NODOA^.CDR);
           END;
    FIN_LISTA : LISTA1 := NIL;
    CUALQUIER_OTRO: BEGIN
              OBTENERNODO (NODOA); OBTENERNODO (NODOB);
              LISTA1 := NODOA; NODOA^.CAR := NODOB;
              NODOB^.VALOR := COLOCAR_HASH (
                                  NEXT_TOKEN.VAL_ALFA);
              END;
  END;
END;
```

El procedimiento GETTOKEN proporciona el siguiente elemento de sintaxis que se encuentra en la cadena de

símbolos de entrada. Este elemento es colocado en la variable NEXT_TOKEN. Para aislar el elemento se debe leer siempre un símbolo adelante, el cual ya no forma parte del elemento que se este procesando. La variable CH contiene el símbolo que se esta trabajando. El algoritmo de GETTOKEN es el siguiente:

```
PROCEDURE GETTOKEN;
BEGIN
  WHILE CH = ' ' DO NEXT_CHAR; { ELIMINAR BLANCOS }
  IF CH es delimitador
    THEN BEGIN
      NEXT_TOKEN.VAL_ALFA := CH;
      NEXT_TOKEN.VAL_NUM := 0;
      CASE CH OF        { QUE DELIMITADOR ES }
        '>','<','=' : NEXT_TOKEN.REP:=OPER_RELACION;
        '+','-','*','/' : NEXT_TOKEN.REP :=
                                  OPER_BINARIO;
        '[' : NEXT_TOKEN.REP := INICIO_LISTA;
        ']' : NEXT_TOKEN.REP := FIN_LISTA;
        ')','(' : NEXT_TOKEN.REP := PARENTESIS;
      END;
      NEXT_CHAR; { LEER UN SIMBOLO ADELANTE }
    END
    ELSE BEGIN
      AGRUPAR_TOKEN;
```

```
        END;
    END;
```

NEXT_CHAR es el procedimiento encargado de proporcionar un símbolo a la vez al analizador de léxico. Estos símbolos estan guardados en la variable CADENA_ENTRADA que es un arreglo de 80 posiciones de tipo CHAR.

ULT_ENTRADA es un puntero al último símbolo en CADENA_ENTRADA. PTR_ENTRADA apunta al símbolo que debe extraerse de la CADENA_ENTRADA. Cuando no existen mas símbolos que procesar debera leerse una nueva serie de ellos y colocarlos en CADENA_ENTRADA. El algoritmo de NEXT_CHAR es el siguiente:

```
PROCEDURE NEXT_CHAR;
BEGIN
  IF PTR_ENTRADA = ULT_ENTRADA
   THEN BEGIN { obtener una nueva linea }
     PTR_ENTRADA := 0;
     ULT_ENTRADA := 0;
     WHILE NOT EOLN(INPUT) AND(ULT_ENTRADA<80)
      DO BEGIN
         ULT_ENTRADA := ULT_ENTRADA + 1;
         READ(CADENA_ENTRADA[ULT_ENTRADA]);
      MAYUSCULAS(CADENA_ENTRADA[ULT_ENTRADA]);
      END;
```

```
            READLN;
         END;
         PTR_ENTRADA := PTR_ENTRADA + 1;
         CH := CADENA_ENTRADA[ULT_ENTRADA];
      END;
```

El intérprete sólo reconoce las mayúsculas por lo cual cada letra minúscula debe ser transformada en su respectiva mayúscula. Para ello se emplea el procedimiento MAYUSCULAS. Su algoritmo es:

```
PROCEDURE MAYUSCULAS(LETRA);
BEGIN
   IF (LETRA >= 'a') AND (LETRA <= 'z')
    THEN LETRA := CHR(ORD(LETRA) - 32);
END;
```

El procedimiento AGRUPAR_TOKEN aisla una serie de símbolos separados por delimitadores para formar un elemento de sintaxis. Si el elemento es un número lo convierte a su valor numérico. Su algoritmo es el siguiente:

```
PROCEDURE AGRUPAR_TOKEN;
BEGIN
   WHILE noESdelimitador CH DO {aislar el elemento}
    BEGIN
      NEXT_TOKEN.VAL_ALFA:=NEXT_TOKEN.VAL_ALFA + CH;
      NEXT_CHAR;
```

```
END;
IF TOKEN_NUMERICO
 THEN BEGIN
   NEXT_TOKEN.REP := NUMERO;
   NEXT_TOKEN.VAL_NUM := VALOR_NUMERICO(
                                NEXT_TOKEN.VAL_ALFA);
 END
 ELSE BEGIN
    CASE NEXT_TOKEN.VAL_ALFA[1] OF
       '"' : NEXT_TOKEN.REP := COMILLAS_IDENT;
       ':' : NEXT_TOKEN.REP := DOSPUNTOS_IDENT;
       CUALQUIER_OTRO : NEXT_TOKEN.REP :=
                                 IDENTIFICADOR;
     END;
   END;
END;
```

El procedimiento TOKEN_NUMERICO proceso cada símbolo colocado en NEXT_TOKEN.VAL_ALFA y determina si el elemento es un número.

4.1.2 Interacción con el manejo de tablas.

El analizador de léxico es el encargado de crear el descriptor para cada elemento de sintaxis que no sea: número, inicio_lista o fin_lista. El procedimiento CONS llama a la función COLOCAR_HASH para que crear una entrada en el diccionario central y colocar el valor

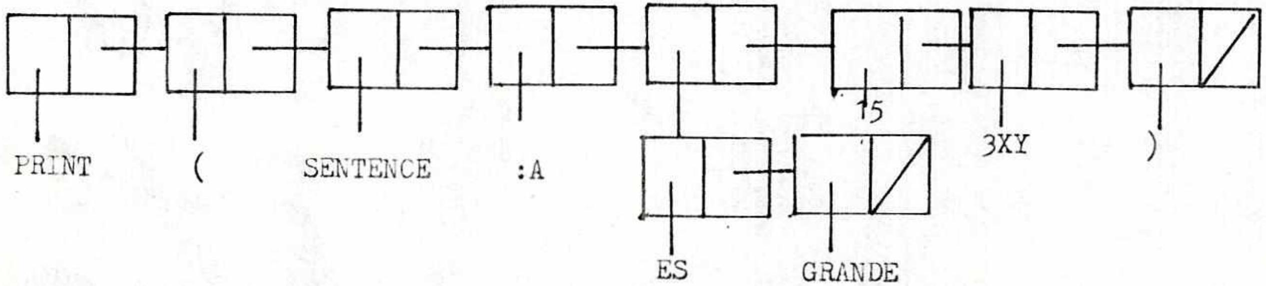
NEXT_TOKEN.VAL_ALFA en el campo de representación alfanumérica del descriptor.

### 4.1.3 Ejemplo de análisis de léxico.

En la figura 4.1.2 se muestran los elementos obtenidos al realizar el análisis de léxico y la S_expr producida al leer la cadena PRINT (SENTENCE :A [ES GRANDE] 15 3XY).

## 4.2 Análisis de sintaxis y generación de código.

El análisis de sintaxis determina si la S_expr producida por el analizador de léxico es una oración válida en Logo.




| | VAL_ALFA | VAL_NUM | REP |
|---|---|---|---|
| PRINT | PRINT | 0 | identificador |
| ( | ( | 0 | delimitador |
| SENTENCE | SENTENCE | 0 | identificador |
| :A | :A | 0 | dospunt-ident |
| [ | [ | 0 | inicio-lista |
| ES | ES | 0 | identificador |
| GRANDE | GRANDE | 0 | identificador |
| ] | ] | 0 | fin-lista |
| 15 | | 15 | número |
| 3XY | 3XY | 0 | identificador |
| ) | ) | 0 | delimitador |

4.1.2 Ejemplo de análisis de léxico

Esta S-expr debe ser convertida en un árbol de sintaxis que represente una expresión. La raíz es el operador y cada rama es un operando. Si se logra construir el árbol, la S-expr es una instrucción correcta del Logo, en caso contrario se usa la rutina de error para notificar que se trata de una oración incorrecta.

El árbol producido por el analizador de sintaxis es representado en forma de S-expr y constituye el código que ejecuta el intérprete.

4.2.1 Sintaxis del lenguaje Logo.

La sintaxis del Logo es muy sencilla. La mayoría de las instrucciones estan expresadas como procedimientos prefijos, pero también existen términos entrefijos.

Generalmente no existe ningún símbolo que agrupe la operación prefija y los operandos que le son necesarios, tal como el Lisp en donde los paréntesis siempre cumplen esta función. Por ello para determinar el número de operandos que corresponde a cada operador debe ser consultada la propiedad EXPR o SUBR del mismo. En el valor de esta propiedad se encuentra el número de operandos necesarios para la operación. Ver figura 3.1.5.

La sintaxis del Logo expresada en formato BNF (Backus-Naur Form) es la siguiente:

```
<expresión> ::=  <expr-prefix> ! <expr-if> ! <expr-infix>
<expr-prefix> ::= <operador> <op-list> ! <expr-opindef>
<op-list> ::= <nada> ! <operando> <op-list>     k elementos
<expr-opindef> ::= ( <operador> <indef-op-list> )
<indef-op-list> ::= <nada> ! <operando> <indef-op-list>
<operando> ::= <número>!<comillas-ident>! <dospuntos-ident>
               ! <lista> ! <expresión>
<lista> ::= [ <indef-op-list> ]
<operador>::=<identificador>!<oper-relacion>!<oper-binario>
<expr-infix> ::= <expr-simple> !
                 <expr-simple><oper-relacion><expr-simple>!
                 ( <expr-infix> )
<expr-simple> ::= <term> ! <expr-simple> + <term> !
                  <expr-simple> - <term>
<term> ::= <factor> ! <term> * <factor> ! <term> / <factor>
<factor> ::= <número>!<comillas-ident>! <dospuntos-ident> !
             <expr-opindef>
<expr-if> ::= IF <expresión> THEN [ <expr-list> ] !
              IF <expresión> THEN [ <expr-list> ]
                                  ELSE [ <expr-list> ]
<expr-list> ::= <nada> ! <expresión> <expr-list>
<nada> ::=
```

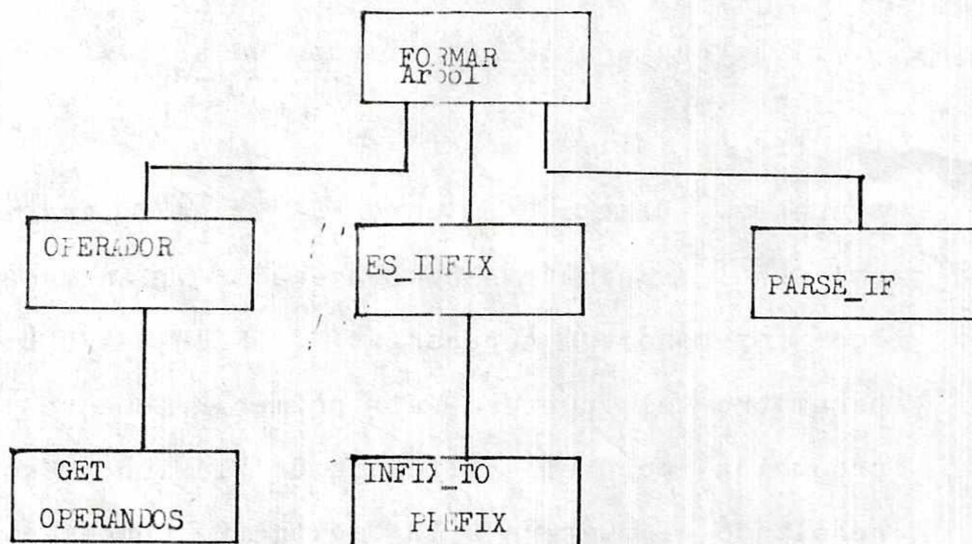
Los términos <identificador>, <comillas-ident>, <dospuntos-ident>, <número> y <oper-relación> fueron explicados en la seccion 4.1.

4.2.2 Estructura del analizador de sintaxis.

La figura 4.2.1 muestra la estructura del analizador de sintaxis. El procedimiento FORMAR_ARBOL trata de construir un árbol cuya raíz es un operador y las ramas representan los operandos necesarios. FORMAR_ARBOL recibe como parametro el puntero al primer elemento de la S-Expr producida por el analizador de léxico y proporciona como resultado el puntero a la S-expr que representa el árbol de sintaxis.

El algoritmo de FORMAR_ARBOL es el siguiente:

```
FUNCTION FORMAR_ARBOL(instrucción);
BEGIN
        { paréntesis: se trata de <expr-opindef> o
                                    ( <expr-infix> )  }
  IF EQUAL(CAR_LST(instrucción);'()
    THEN BEGIN
      instrucción := CDR_LST(instrucción);
      IF OPERADOR(CAR_LST(instrucción))
        THEN BEGIN   { es <expr-opindef>  }
          NEW(nodoa);
          nodoa^.CAR := CAR_LST(instrucción);
          nodoa^.CDR := GET_OPERANDOS(
                            MAXINT,instrucción);
          RETURN(nodoa);
        END
```




4.2.1 Analizador de sintaxis

```
      ELSE BEGIN    { es (<expr-infix>)  }
        RETURN(INFIX_TO_PREFIX(instrucción))
      END;
    END;

IF OPERADOR(CAR_LST(instrucción))
   THEN BEGIN
     IF EQUAL(CAR_LST(instrucción),IF)
       THEN BEGIN      {    <expr-if>   }
         RETURN(PARSE_IF(instrucción));
       END
       ELSE BEGIN       {    <expr-prefix>  }
         NEW(nodoa);
         nodoa^.CAR := CAR_LST(instrucción);
         i:=CUANTOS_OPERANDOS(CAR_LST(instrucción));
         nodoa^.CDR := GET_OPERANDOS(i,instrucción);
```

```
            RETURN(nodoa);
          END;
        END;

      IF EXPR_INFIX(instrucción)
        THEN BEGIN               {  <expr-infix>  }
          RETURN(INFIX_TO_PREFIX(instrucción));
        END
        ELSE BEGIN
          ERROR 'no corresponde a sintaxis'
        END;

  END;
```

La función OPERADOR produce un valor verdadero si el elemento de sintaxis representa <operador> y además posee la propiedad EXPR o SUBR.

La función CUANTOS_OPERANDOS investiga en la propiedad EXPR o SUBR del <operador> para obtener el número de operandos que necesita.

La función GET_OPERANDOS recibe como entrada el número de operandos que debe obtener de la S-expr apuntada por el parámetro INSTRUCCION. Los operandos son regresados en forma de lista. Su algoritmo es:

```
    FUNCTION GET_OPERANDOS(num_op, instrucción);
    BEGIN
```

```
obtenidos := 0;
lista_op := NIL;
WHILE (instrucción<>NIL) AND (OBTENIDOS < num_op) do
 BEGIN
   NEW(nodoa); nodoa^.CAR := NIL;
   IF ES_INFIX(instrucción)
      THEN BEGIN          {   <expr-infix>  }
        nodoa^.CAR := INFIX_TO_PREFIX(instrucción);
      END
      ELSE BEGIN
        IF OPERADOR(CAR_LST(instrucción))
          THEN BEGIN     {    <expr-prefix> }
            nodoa^.CAR := FORMAR_ARBOL(instrucción);
          END
          ELSE BEGIN     {  <operando>    }
            nodoa^.CAR := CAR_LST(INSTRUCCION);
            instrucción := CDR_LST(INSTRUCCION);
          END;
     END;
   obtenidos := obtenidos + 1;
   agregar nodoa al final de lista_op;
  END;
  RETURN(lista_op);
END;
```

La función ES_INFIX regresa un valor verdadero cuando encuentra una lista que presenta las instrucciones en formato entrefijo. Además aisla esta lista para que sea convertida en el árbol de sintaxis.

La función INFIX_TO_PREFIX crea el árbol de sintaxis para las expresiones entrefijas La lista infix IL es recorrida de izquierda a derecha y los operandos y operadores que aun no han sido agregados al árbol son guardados en dos listas separadas llamadas OPERADORES y OPERANDOS.

Se emplea una función que indica el peso de cada operador. Estos pesos son:

a) Para '=' : 0

b) para '>','<' : 2

c) para '+','-' : 4

d) para '*','/' : 6

Los objetos son agregados al principio de las dos listas cuando el operador al principio, de lo que aun existe en IL, tiene un peso mayor que el operador que se encuentra a la cabeza de OPERADORES. En caso contrario, el primer operador de la lista OPERADORES se combina con los primeros dos operandos guardados en OPERANDOS y esta combinación se guarda al principio de la lista OPERANDOS. El algoritmo es el siguiente:

```
FUNCTION INFIX_TO_PREFIX(il);
BEGIN
   operandos := NIL;      {    lista de operandos  }
   operadores := NIL;      {  lista de operadores }
5:
   IF OPERADOR(CAR_LST(il))
      THEN{agregar al principio de la lista operandos}
           operandos := CONS_LST(INFIX_TO_PREFIX(il),
                                              operandos);
       ELSE
       {agregar el elemento al principio de operandos}
           operandos := CONS_LST(CAR_LST(il),
                                              operandos);
   il := CDR_LST(il);

10:
   IF NULL (il) AND NULL (operadores)
       {el árbol de resultado }
     THEN RETURN(CAR_LST(operandos))

   IF NULL (il) OR (PESO(CAR_LST(il) <=
                    PESO(CAR_LST(operadores)) )
     THEN BEGIN   {  construir expresión prefix   }
       inst := CONS_LST(
                   CAR_LST(operadores),
                  CONS_LST(CAR_LST(CDR_LST(operandos)),
                             CAR_LST(OPERANDOS))
```

```
                );
          operandos := cdr_lst(cdr_lst(operandos);
          operandos := CONS_LST(inst,operandos);
          operadores := cdr_lst(operadores);
          GOTO 10;
        END;

        operadores := CONS_LST(CAR_LST(il),operadores);
        il := CDR_LST(il);
        GOTO 5;

   END;
```

La función PARSE_IF forma el árbol de sintaxis de la instrucción IF. El parámetro INSTRUCCION contiene el puntero a la lista producida por el analizador de léxico. Su algoritmo es el siguiente:

```
FUNCION PARSE_IF (instrucción);
BEGIN
  instrucción := CDR_LST(instrucción);
  NEW(nodoresp); nodoresp^.CDR := NIL;
  nodoresp^.CAR := FORMAR_ARBOL(instrucción);
        {formar [ then-lst ]}
  IF EQUAL(CAR_LST(instrucción),'THEN')
    THEN BEGIN
      NEW(nodot); nodot^.CDR := NIL;
      nodot^.CAR := FORMAR_ARBOL(instrucción);
```

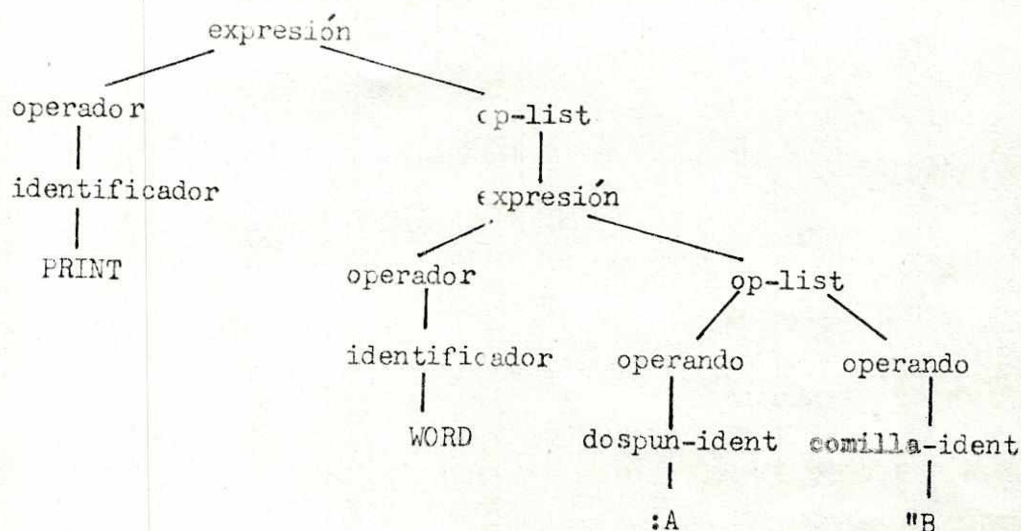
```
        END;
              { formar [else-lst] }
      IF EQUAL(CAR_LST(instrucción),'ELSE')
        THEN BEGIN
          NEW(nodot2); nodot2^.CDR := NIL;
          nodot2^.CAR := FORMAR_ARBOL(instrucción);
          nodot^.CAR := nodot2;
        END;
    END;
```

4.2.3 Ejemplo de análisis de sintaxis.

En esta sección se muestran ejemplos de los árboles que representan los pasos para verificar la sintaxis del lenguaje Logo para casos específicos. Además contiene las derivaciones de extrema izquierda para los mismos.

La figura 4.2.2 muestra el árbol de pasos de sintaxis para la expresión [PRINT WORD :A "B]. Su derivación de extrema izquierda es:

```
<expresión> ---> <expr-prefix>
            ---> <operador> <op-list>
            ---> <identificador> <op-list>
            ---> PRINT <expresión>
            ---> PRINT <operador> <op-list>
            ---> PRINT <identificador> <op-list>
            ---> PRINT WORD <operando> <operando>
```




4.2.2 [PRINT WORD :A B]

---> PRINT WORD <dospuntos-ident> <operando>

---> PRINT WORD :A <comillas-ident>

---> PRINT WORD :A "B

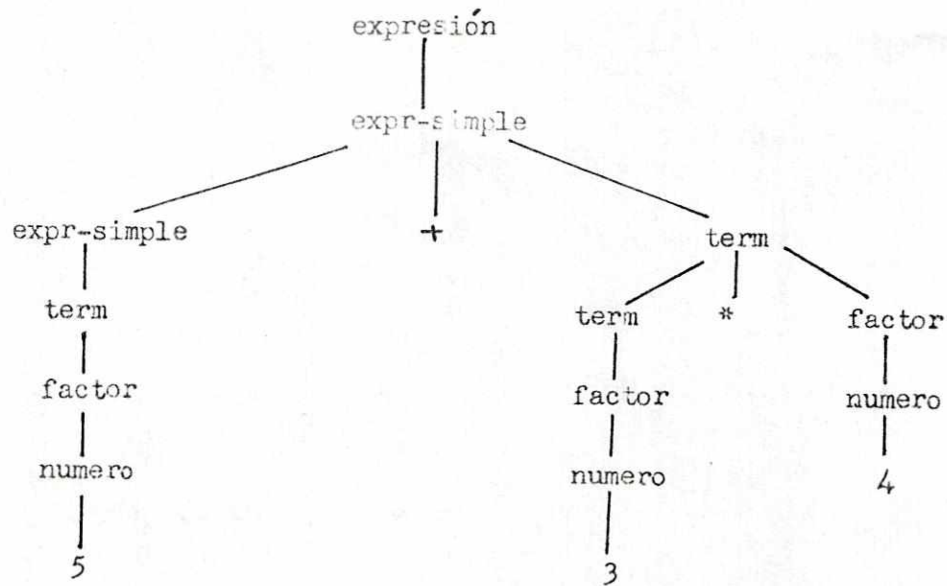
La figura 4.2.3 muestra el árbol de pasos de sintaxis para la expresión [5 + 3 * 4]. Su derivación de extrema izquierda es:

<expresión> ---> <expr-infix>

---> <expr-simple>

---> <expr-simple> + <term>

---> <term> + <term>




4.2.3 [5 + 3 * 4]

---> <factor> + <term>

---> <número> + <term>

---> 5 + <term>

---> 5 + <term> * <factor>

---> 5 + <factor> * <factor>

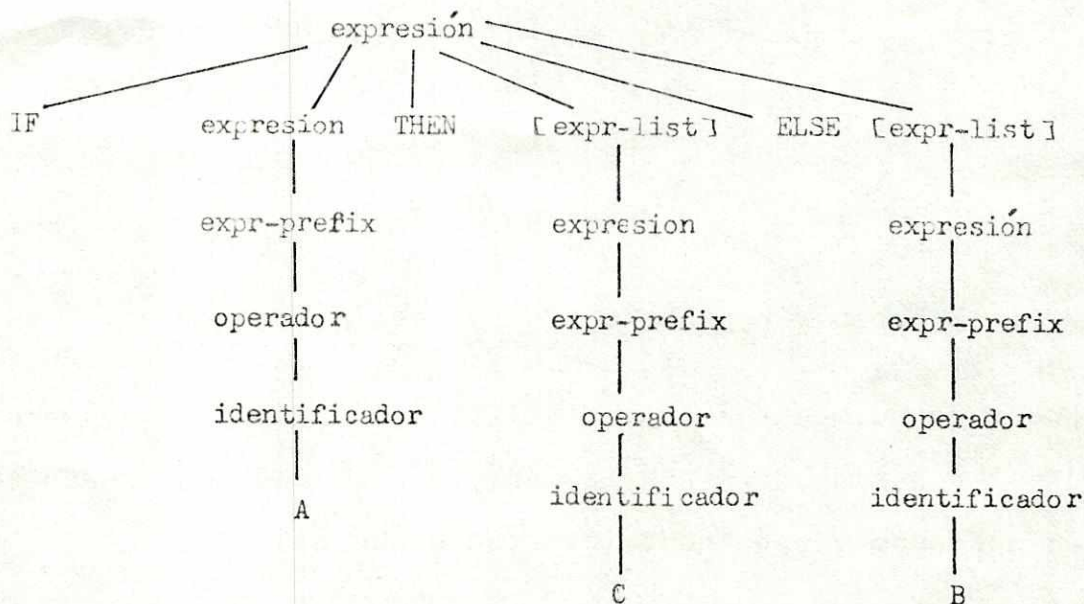
---> 5 + <número> * <factor>

---> 5 + 3 * <factor>

---> 5 + 3 * <número>

---> 5 + 3 * 4

La figura 4.2.4 muestra el árbol de pasos de sintaxis para la expresión [IF A THEN [C] ELSE [B]]. Su derivación de extrema izquierda es:




4.2.4 [IF A THEN[C] ELSE[B]]

<expr

---> IF <expr-prefi> THEN [<expr-list>]
ELSE [<expr-list>]

---> IF <operador> THEN [<expr-list>]
ELSE [<expr-list>]

---> IF <identificador> THEN [<expr-list>]
ELSE [<expr-list>]

---> IF A THEN [<expr-list>] ELSE [<expr-list>]

---> IF A THEN [<expresión>] ELSE [<expr-list>]

```
--->  IF A THEN[<expr-prefix>] ELSE[<expr-list>]
--->  IF A THEN [<operador>] ELSE [<expr-list>]
--->  IF A THEN [C] ELSE [<expresión>]
--->  IF A THEN [C] ELSE [<expr-prefix>]
--->  IF A THEN [C] ELSE [<operador>]
--->  IF A THEN [C] ELSE [B]
```

4.2.4 Ejemplo y representacion de código producido.

Los procedimientos del análisis de sintaxis producen un árbol resumido de sintaxis, en el cual cada hoja representa un operando y cada nodo interior es un operador.

El árbol resumido de sintaxis es representado por medio de listas encadenadas. Esta lista constituye la S-expr producida por el análisis de sintaxis y es el código entendido por el intérprete. En esta S-expr el primer elemento es el operador y los siguientes son sus operandos. En la figura 4.2.5 se presenta el árbol resumido de sintaxis y su representación en forma de S-expr, de las instrucciones PRINT WORD :A "B y 5+3*4.

4.2.5 Interacción con el manejo de tablas.

La función OPERADOR produce verdadero si el elemento de sintaxis investigado representa <operador>. Para ello consulta el descriptor del elemento y busca en su lista de propiedades la propiedad EXPR o SUBR.

PRINT WORD :A "B

PRINT
WORD
:A "B

+ 5 * 3 4

+
5 *
3 4

4.2.5 Arboles resumidos de sintaxis

Para formar <op-list>, lista de operandos que necesita un <operador>, es necesario conocer el número exacto de operandos. Para ello la funcion CUANTOS_OPERANDOS consulta el valor de la propiedad EXPR o SUBR, donde se encuentra indicado el número de parámetros que necesita el <operador> específico.

4.3 Manejo de tablas.

El diccionario central contiene los descriptores de los elementos de sintaxis que no sean número. Estos descriptores son creados por el analizador de léxico y consultado por el analizador de sintaxis. Además es empleado en tiempo de ejecución por el intérprete como se indicó en la seccion 3.1.2.1.

El diccionario central esta construido de acuerdo a la técnica de desmenuzamiento abierto. Esta formado por un vector, en el que cada entrada contiene el puntero a la lista de descriptores que tienen el mismo valor para la función de desmenuzamiento de su representación alfanumérica. En la lista de propiedades de los elementos <identificador> se encuentra la informacion del número de operandos que necesita el procedimiento.

4.3.1 Instalar un descriptor.

El descriptor contiene:

a)representación alfanumérica.

b)puntero a la lista de propiedades.

c)puntero a la lista de valores asociados.

d)puntero al próximo descriptor.

El encargado de crear estos descriptores en el momento de análisis de léxico es el procedimiento COLOCAR_HASH. Este toma la representación alfanumérica del elemento de sintaxis que se encuentra en NEXT_TOKEN.VAL_ALFA y por

medio de una función de desmenuzamiento determina en que entrada del diccionario central principia la lista en que debe colocarse al descriptor, lo instala y regresa como resultado el puntero al descriptor. El algoritmo de COLOCAR_HASH es el siguiente:

```
FUNCTION COLOCAR_HASH(rep_alfa);
BEGIN
  descriptor := BUSQUE_PTRL(rep_alfa);
  IF descriptor = NIL
    THEN BEGIN          { no existe el descriptor }
        NEW(descriptor);          {   crear descriptor }
        entrada := HASH(rep_alfa);
            {el descriptor es colocado al
             principio de la lista }
      descriptor^.NEXT_CABEZA:= diccionario[entrada];
           { representacion alfanumerica }
      descriptor^.NOMB := rep_alfa;
        descriptor^.VALOR := NIL;{  valores asociados }
        descriptor^.PPLIST:= NIL;{lista de propiedades}
    END;
    RETURN(descriptor);
 END;
```

4.3.2 Búsqueda en el diccionario central.

Para localizar un descriptor se emplea la funcion BUSQUE_PTRL. Ella toma como argumento la representación alfanumérica del elemento y por medio de la funcion hash obtiene la entrada en el Diccionario Central que contiene la lista en que se encuentra el descriptor deseado. Recorre esta lista hasta encontrar un descriptor con la representación alfanumérica igual a la que se busca. Regresa como resultado el puntero a este descriptor, si existe. El algoritmo de BUSQUE_PTRL es:

```
FUNCION BUSQUE_PTRL (rep_alfa);
BEGIN
   found := FALSE;
   descriptor := NIL;  {  suponer no existe  }
   lista := diccionario [HASH(rep_alfa)];
      { recorrer la lista de descriptores }
   WHILE (NOT found) AND ( lista <> NIL) DO
     IF lista^.NOMB = rep_alfa
       THEN BEGIN {se encontro el descriptor }
         descriptor := lista;
         found := TRUE;
       END
       ELSE  {  obtener siguiente descriptor }
         lista := lista^.NEXT_CABEZA;
  RETURN(descriptor);
END;
```

4.3.3 Instalar funciones propias del Logo.

Para que se pueda efectuar el análisis de sintaxis ya debe estar creado el descriptor de cada <identificador> que representa una función u operador. En el debe existir la propiedad EXPR o SUBR y en su valor contener el número de operandos necesarios para ejecutarla.

El procedimiento DEF_FUNCION crea el descriptor para cada función propia del Logo y coloca la propiedad SUBR en su lista de propiedades. El valor de la propiedad es el número de operandos y el identificador interno de la función. Su algoritmo es el siguiente:

```
PROCEDURE DEF_FUNCION ( nomb_externo,
                 nomb_interno, número_operandos);
BEGIN
  pdesc := COLOCAR_HASH(nomb_externo);
  DEFLIST(pdesc,'SUBR',
         LIST(nomb_interno,número_operandos) );
END;
```

## Apéndice A. Sintaxis del lenguaje Logo.

Las expresiones regulares que describen los elementos de sintaxis son:

```
identificador = caracter
comillas_ident = "caracter
dospuntos_ident = :caracter
caracter = digito ¦ letra ¦ caracter_especial
letra = A¦B¦...¦Z¦a¦b¦...¦z
digito = 0¦1¦...¦9
caracter_especial = "¦^¦$¦&¦'¦,¦.¦\¦:¦;¦@¦-
numero = decimal ¦ exponencial
decimal = signo entero.entero ¦ signo entero. ¦ .entero
signo = - ¦ nada
entero = digito
exponencial = (decimal ¦ signo entero ) (E¦N) entero
oper_relacion = < ¦ > ¦ =
oper_binario = + ¦ - ¦ * ¦ /
inicio_lista = [
fin_lista = ]
delimitador = oper_relacion ¦ oper_binario ¦ blanco ¦
              inicio_lista ¦ fin_lista ¦ ( ¦ )
```

La sintaxis del Logo expresada en formato BNF (Backus-Naur Form) es la siguiente:

```
<expresion> ::=  <expr-prefix> ! <expr-if> ! <expr-infix>

<expr-prefix> ::= <operador> <op-list> ! <expr-opindef>

<op-list> ::= <nada> ! <operando> <op-list>     k elementos

<expr-opindef> ::= ( <operador> <indef-op-list> )

<indef-op-list> ::= <nada> ! <operando> <indef-op-list>

<operando> ::= <numero>!<comillas-ident>! <dospuntos-ident>

            ! <lista> ! <expresion>

<lista> ::= [ <indef-op-list> ]

<operador>::=<identificador>!<oper-relacion>!<oper-binario>

<expr-infix> ::= <expr-simple> !

                 <expr-simple><oper-relacion><expr-simple>!

                 ( <expr-infix> )

<expr-simple> ::= <term> ! <expr-simple> + <term> !

                  <expr-simple> - <term>

<term> ::= <factor> ! <term> * <factor> ! <term> / <factor>

<factor> ::= <numero>!<comillas-ident>! <dospuntos-ident> !

            <expr-opindef>

<expr-if> ::= IF <expresion> THEN [ <expr-list> ] !

              IF <expresion> THEN [ <expr-list> ]

                                 ELSE [ <expr-list> ]

<expr-list> ::= <nada> ! <expresion> <expr-list>

<nada> ::=
```

Apéndice B. Un programa Logo.

El programa POET escribe un poema compuesto de tres líneas. Cada línea tiene una estructura especial definida por el usuario. Las palabras para formar estas líneas son tomadas al azar de una lista de sustantivos, verbos, preposiciones, artículos y adjetivos. Estas listas pueden ser cambiadas al gusto de la persona para formar poemas sobre un mismo tema.

Este programa fue copiado del libro "Learning with Logo", ver bibliografía. Consta de varios subprogramas. El procedmiento POEMS pide al usuario cuantos poemas desea. POET escribe un poema en tres líneas. LINE1 escribe una liea compuesta por un artículo, un adjetivo y un nombre. LINE2 forma una linea de artículo, nombre, verbo, preposición, artículo, adjetivo y nombre. LINE3 produce una línea con adjetivo, adjetivo y nombre.

Los procedimientos ARTICLE, ADJECTIVE, NOUN, VERB y PREPOSITION toma al azar una palabra de las listas de artículos, adjetivos, nombres, verbos y preposiciones respectivamente.

El procedimiento PICKRANDOM toma un elemento al azar de la lista que toma como parámetro. READNUMBER lee un número.

```
TO POEMS
CLEARTEXT
PRINT [CUANTOS POEMAS DESEA? ]
```

```
MAKE "N READNUMBER
CLEARTEXT PRINT [] PRINT []
PRINT SENTENCE :N [POEMAS POR EL POETA LOGO]
PRINT []
REPEAT :N [POET PRINT [] ]
END

TO POET
   PRINT LINE1
   PRINT LINE2
   PRINT LINE3
END

TO LINE1
  OUTPUT (SENTENCE ARTICLE ADJECTIVE NOUN)
END

TO LINE2
  OUTPUT (SENTENCE ARTICLE NOUN VERB PREPOSITION
          ARTICLE NOUN)
END

TO LINE3
   OUTPUT (SENTENCE ADJECTIVE ADJECTIVE NOUN)
END

TO ARTICLE
   OUTPUT PICKRANDOM :ARTICLELIST
END
```

```
TO ADJECTIVE
   OUTPUT PICKRANDOM :ADJECTIVELIST
END

TO NOUN
   OUTPUT PICKRANDOM :NOUNLIST
END

TO VERB
   OUTPUT PICKRANDOM :VERBLIST
END

TO PREPOSITION
   OUTPUT PICKRANDOM :PREPOSITIONLST
END

MAKE "ARTICLELIST [LA EL UN UNA]
MAKE "ADJECTIVELIST [GRANDE VELOZ EFICIENTE ]
MAKE "NOUNLIST [PROGRAMA PROGRAMADOR LISTADO
                COMPUTADOR ]
MAKE "VERBLIST [FUNCIONA IMPRIME FUNCIONA
                CALCULA PROGRAMA]
MAKE "PREPOSITIONLST [SOBRE EN ]

TO PICKRANDOM  :OBJECT
   MAKE "NUMBER 1 + RANDOM COUNT :OBJECT
END
```

```
TO PICK :NUMBER :OBJECT
  OUTPUT ITEM :NUMBER :OBJECT
END

TO READNUMBER
  MAKE "NUM1 READLIST
  OUTPUT FIRST :NUM1
  END
```

Apéndice C. Instrucciones Logo.

A continuación se encuentra una lista de las instrucciones Logo que reconoce el intérprete construido. El signo de interrogación al principio de una instrucción indica que se puede usar un número variable de parámetros.

Pred indica una palabra que sólo puede ser TRUE o FALSE. Obj indica que el parámetro puede ser cualquier objeto del lenguaje Logo. List especifica que la entrada al procedimiento debe ser una lista. N significa que el parámetro debe ser un número. Name indica que se trata de una palabra que comience con comillas.

| Instrucción | Descripción |
|---|---|
| ?AND pred1 pred2 | Produce TRUE si todos los parámetros son TRUE. |
| ARCTAN n | Produce la arcotangente de n. |
| ASCII char | Produce el codigo ASCII de char. |
| BACK, n<br>BK | Mueve la tortuga n pasos hacia atras. |
| BUTFIRST obj<br>BF | Produce obj sin su primer elemento. obj no debe ser número. |

| | |
|---|---|
| BUTLAST obj<br>BL | Produce obj sin su ultimo elemento. obj no debe ser un número. |
| CHAR n | Produce el simbolo cuyo codigo ASCII es n. |
| CLEAN | Borra la pantalla grafica sin afectar la tortuga. |
| CLEARSCREEN<br>CS | Borra la pantalla, mueve la tortuga a [0,0] y pone la cabeza a cero. |
| CLEARTEXT | Borra la pantalla de textos. |
| COS n | Produce el coseno de n grados. |
| COUNT list | Produce el número de elementos en list. |
| EMPTYP obj | Produce TRUE si obj es la lista vacia o la palabra nula. |
| EQUALP obj1 obj2 | Produce TRUE si obj1 es igual a obj2. |
| FIRST obj | Produce el primer elemento en obj. obj no debe ser número. |

| | |
|---|---|
| FORWARD n<br>FD | Mueve la tortuga n pasos hacia adelante. |
| FPUT obj list | Produce la lista que resulta al agregar obj al inicio de list. |
| FULLSCREEN | Dedica toda la pantalla a graficas. |
| GPROP name prop | Produce la propiedad prop de name. |
| HEADING | Produce el valor de la cabeza de la tortuga. |
| HIDETURTLE<br>HT | Hace invisible a la tortuga. |
| HOME | Mueve la tortuga hacia [0,0] y cero a la cabeza de la tortuga. |
| IF pred THEN list1<br>ELSE list2 | Si pred es TRUE, efectua las instrucciones de list1, en caso contrario las de list2. |
| INT n | Produce la parte entera de n. |
| ITEM n list | Produce el n-esimo elemento de list. |

| | |
|---|---|
| LAST obj | Produce el ultimo elemento de obj. obj no debe ser número. |
| LEFT n<br>LT | Gira a la tortuga n grados hacia la izquierda, es decir en sentido contrario a reloj. |
| ?LIST obj1 obj2 | Produce la lista formada por los parametros. |
| LISTP obj | Produce TRUE si obj es una lista. |
| LOAD name | Lee instrucciones Logo del archivo llamado name. |
| LPUT obj list | Produce la lista formada al agregar obj al final de list. |
| MAKE name obj | Crea una asociacion entre name y obj. |
| MEMBERP obj list | Produce TRUE si obj es un elemento de list. |
| NAME obj name | Crea asociacion entre el valor obj y name. |
| NAMEP word | Produce verdadero si word tiene relacionado algun valor. |

| | |
|---|---|
| NOT pred | Produce TRUE si pred es FALSE. |
| NUMBERP obj | Produce TRUE si obj es un número. |
| ?OR pred1 pred2 | Produce TRUE si alguno de sus parametros es verdadero. |
| OUTPUT obj | Produce obj como resultado de un procedimiento del usuario. |
| PENDOWN<br>PD | Hace que baje el lapiz. |
| PENERASE<br>PE | Coloca abajo al borrador. |
| PENREVERSE<br>PX | Coloca abajo al lapiz invertido. |
| PENUP<br>PU | Sube al lapiz. |
| PLIST name | Produce la lista de propiedades de name. |
| POS | Produce la posicion de la tortuga en forma de [x y]. |
| ?PRINT obj | Escribe obj seguido de un regreso de carro. |

| | |
|---|---|
| ?PRODUCT a b | Produce el producto de multiplicar los parametros. |
| QUOTIENT a b | Produce la parte entera de dividir a por b. |
| RANDOM n | Produce un número no negativo al azar menor que n. |
| READCHAR<br>RC | Produce el simbolo escrito por por el usuario. |
| READLIST<br>RL | Produce la linea escrita por el usuario. |
| REMAINDER a b | Produce el resto de la division de a por b. |
| REMPROP name prop | Elimina la propiedad prop de name. |
| REPEAT n list | Efectua las instrucciones en list n veces. |
| RERANDOM | Hace que se puedan reproducir los números generados por RANDOM. |
| RIGHT n<br>RT | Gira la tortuga n grados hacia la derecha, es decir en el sentido de las agujas del reloj. |

| | |
|---|---|
| ROUND n | Produce el valor de n redondeado al entero mas proximo. |
| RUN list | Efectua las instrucciones colocadas en list. |
| ?SENTENCE obj1 obj2<br>SE | Produce la lista producida con los parametros. |
| SETHEADING n<br>SETH | Apunta la cabeza de la tortuga a n grados. |
| SETPOS [x y] | Coloca la tortuga en las coordenadas [x y] |
| SETX pos | Mueve horizontalmente la tortuga de tal manera que su coordenada en x es pos. |
| SETY pos | Mueve verticalmente la tortuga de tal manera que su coordenada en y es pos. |
| SHOW obj | Escribe obj. Tambien escribe [] si obj es una lista. |
| SHOWNP | Produce TRUE si la tortuga esta visible. |

| | |
|---|---|
| SHOWTURTLE<br>ST | Hace que la tortuga sea visible. |
| SIN n | Produce el seno de n grados. |
| SPLITSCREEN | Divide la pantalla para que sea empleada por graficas y por texto. |
| SQRT n | Produce la raiz cuadrada de n. |
| STOP | Finaliza un procedimiento del usuario. |
| ?SUM a b | Produce la suma de los parametros. |
| TEST pred | Permite recordar si pred es TRUE o FALSE. |
| TEXTSCREEN | Dedica toda la pantalla para texto. |
| THING name | Produce el valor asociado con name. |
| TO name parms | Inicia la definicion de un procedimiento, los parametros deben ser escritos en la misma linea. |

| | |
|---|---|
| ?TYPE obj | Escribe obj. |
| ?WORD word1 word2 | Produce la palabra formada por la union de los parametros. |
| WORDP obj | Produce TRUE si obj es una palabra. |
| XCOR | Produce la coordenada x de la tortuga. |
| YCOR | Produce la coordenada y de la tortuga. |

BIBLIOGRAFIA

Aho, Alfred V. _Data Structures and Algorithms_. New York, Addison-
1979 Wesley Publishing Co.

__________; _Principles of Compler design_. New york, Addison-
1983 Wesley Publishing Co.

Davidson, Laurence J. _Logo Programmer´s Manual_. New York, Logo
1982 Computer Systems, Inc.

Gries, D. _Compiler Construction for digitsl computers_. New York,
1971 Wiley International.

Kernighan, Brian W.; Player, P. _Software Tools in Pascal_. Addison-
1981 Wesley Publishing Co.

Papert, Seymour _Mindstorms_. Basic Books.
1980

__________; _Desafío a la mente. Computadoras y educación_.
1981 Argentina, Ediciones Galápago.

Pratt, T. W. _Programming Languajes_. U.S.A. Prentice-Hall Interna-
1975 tional.

Reggini, Horacio C. _Alas para la mente. Logo: un lenguaje de com-_
1982 _putadoras y un estilo de pensar_. Argentina, Ediciones
Galápago.

Watt, Daniel. _Learning with Logo_. McGraw-Hill Book Company.
1982

Winston, P. H.; Horn, B.K. _Lisp_. Addison-Wesley.
1981