Universidad Nacional Autónoma de México

Colegio de Ciencias y Humanidades

Unidad Académica de los Ciclos Profesional y de Posgrado

DISEÑO E IMPLEMENTACION DE UN SISTEMA ORIENTADO A OBJETOS

T E S I S

Que para obtener el Grado de:

MAESTRIA EN CIENCIAS DE LA COMPUTACION

P r e s e n t a :

Fernando Fermín Jiménez Fraustro

México, D. F. 1986

A Dios ... gracias

A la memoria de mi padre, compañero y guía:

José Luis Jiménez Martínez

Con amor para mi madre:

Juana Ma. F. Vda. de Jiménez

Con cariño para mis hermanos:

Olga Imelda

Octavio Ismael

Guillermo Oswaldo

Miguel Angel

Luis Alejandro

Ana Isabel

Con gran amor a mi esposa, compañera y amiga:

Lidia

A la Universidad Nacional Autónoma de México.

Al Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas.

A los profesores de la Maestría en Ciencias de la Computación

A mis compañeros y amigos.

AGRADECIMIENTOS

Un profundo agradecimiento al director de la presente Tesis: Dr. Miguel Gerzso, por sus enseñanzas y puntos de vista. Se extiende el mismo, al Ing. Salvador Barra, por su colaboración en la realización del presente trabajo. Asimismo, al Dr. Renato Barrera, por la ética y profesionalismo en la revisión de este trabajo. De la misma manera, a las siguientes personas que fungieron como mis sinodales: Dr. Alejandro Buchmann, Dra. Hanna Oktaba y al Dr. Federico Martín.

Agradezco también, al Instituto de Investigaciones Eléctricas, por las facilidades prestadas para la elaboración final de esta Tesis.

Un agradecimiento muy especial, al Dr. Luis Ponce, por su desinteresada colaboración en la corrección de estilo y sintaxis del presente trabajo.

I N D I C E

CAPITULO 3 CLASES

CAPITULO 4 EL COMPILADOR

CAPITULO 5 MAQUINA VIRTUAL

CAPITULO 6 CONCLUSIONES

APENDICE A

APENDICE B

CAPITULO 0

INTRODUCCION

El notable avance en el diseño y producción de computadoras, ha propiciado un rápido incremento del número de usuarios, dando como resultado un desarrollo paralelo de sistemas de software cada vez más sofisticados y al mismo tiempo más accesibles a los usuarios.

Entre estos sistemas, se pueden mencionar los Sistemas Orientados a Objetos (SOO); cuya principal característica es la integración de datos y procedimientos en una sola entidad, llamada objeto.

El Grupo de Investigación de Xerox, desarrolló un sistema basado en los conceptos de SOO, conocido con el nombre de Smalltalk-80 (ST-80) [1], con el fín de realizar sistemas de software funcionales e interactivos para aplicaciones en computadoras personales.

Una de las principales características del sistema ST-80 es la flexibilidad que tiene para desarrollar grandes aplicaciones en comparación de los sistemas tradicionales, ya que el usuario tiene siempre a su alcance todo el sistema a la vez y, cada nueva aplicación forma parte del sistema automáticamente. El usuario puede, además, crear o modificar aplicaciones basadas en las ya existentes.

El sistema ST-80 consta de un sofisticado medio ambiente de interacción con el usuario, a través del cual se establece la comunicación usuario-sistema. En él se pueden definir nuevas aplicaciones, modificar las ya existentes e interactuar con los objetos del sistema, todo ello de una forma totalmente interactiva. Para realizar lo anterior existe un lenguaje al cual el usuario se debe sujetar para establecer la comunicación. El sistema ST-80 fué realizado sobre una máquina Virtual, lo cual hace al sistema transportable de una configuración de hardware a otra.

En éste trabajo se intenta construir un Sistema Smalltalk Simplificado (SS). La simplificación consiste en que la definición de una aplicación no se hace de forma interactiva, si no que es hecha en base a una sintaxis definida a la cual el usuario se sujeta para la definición de la aplicación; este hecho hace al SS un sistema menos flexible comparado con ST-80. Fuera de la forma de definir o modificar una aplicación, el Sistema SS abarca las características principales del sistema ST-80.

La sintaxis para la definición de una aplicación se basó del lenguaje TM, el cual ha sido diseñado [2] en el Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) de la Universidad Nacional Autónoma de México (UNAM).

0.1 RESUMEN.

En el capítulo 1 se describen los conceptos básicos de un Sistema Orientado a Objetos, en particular del sistema ST-80. El desarrollo del trabajo se basa en esos conceptos. Fundamentalmente se explica el concepto de objeto como una entidad independiente; de mensajes, que es la forma de interactuar con esas entidades; de clase, que es la forma de definir nuevos objetos en el sistema y los mensajes a que pueden responder. En la parte final del capítulo, se resaltan las diferencias básicas entre estos tipos de sistemas y los sistemas tradicionales.

En el capítulo 2 se analizan las expresiones válidas, desde el punto de vista sintáctico, para escribir mensajes en el Sistema SS. La sintaxis de las expresiones varía un poco respecto a las que son válidas para el sistema ST-80, a lo largo del capítulo se discuten esas diferencias. Sin embargo, los tipos de expresiones son las misma que para ST-80.

En el capítulo 3 se discute la estructura de una clase para el sistema SS. Se muestra la forma en que un usuario debe de escribir una aplicación. A lo largo de éste capítulo se desarrolla un ejemplo de aplicación, que va creciendo en complejidad de acuerdo a como se van explicando nuevos conceptos sobre clases, para que al final del capítulo alcance su forma final. Se discute también los diferentes tipos de objetos a los que el usuario tiene acceso al escribir una aplicación.

En los capítulos 4 y 5 se muestra la implementación del Sistema SS. En ellos se ve el diseño general del sistema, las herramientas utilizadas para su desarrollo y los algoritmos empleados en los programas. El sistema consiste de tres partes: un medio ambiente de interacción, a través del cual el usuario escribe expresiones para interactuar con los objetos del sistema; un compilador cuya función es traducir esas expresiones para su posterior evaluación; y una Máquina Virtual que tiene como función evaluar esas expresiones y administrar los objetos dentro del sistema.

En el capítulo 4 se trata lo referente al diseño e implementación del compilador. En un principio, se presentan las bases teóricas sobre gramáticas libres de contexto y las diferentes fases de una compilación, para posteriormente, explicar las estructuras de datos y rutinas principales utilizadas para su implementación.

El capítulo 5 trata sobre la implementación de la Máquina Virtual. Se discute el concepto de Máquina Virtual y se listan las ventajas y desventajas que acarrea su uso para el desarrollo de nuevos sistemas. Posteriormente, se muestran los diferentes componentes de la máquina y para cada uno de ellos se muestran los algoritmos y estructuras de datos necesarios para su implementación. Debe mencionarse que el diseño de la Máquina Virtual esta basado en gran parte en la máquina del Sistema ST-80.

En el capítulo 6, se muestran las conclusiones de éste trabajo. Se discuten los resultados logrados en la implementación del Sistema SS y se dan algunas sugerencias para futuros trabajos.

En el apéndice A, se muestran los diagramas sintácticos del lenguaje SS y en el apéndice B, se presenta el uso del generador de tablas de sintaxis utilizado para la implementación del compilador. El generador fué implementado por Salvador Barra alumno de la Maestría en Ciencias de la Computación del IIMAS de la UNAM.

0.2 REFERENCIAS.

[1].-A.Goldberg,D.Robson;Smalltalk-80: The Language and its Implementation; E.U.,Addison-Wesley series in Computer Science (1983).

[2].-M.Gerzso;Report on the Language TM: its design and Definition; México, IIMAS, UNAM (1983).

CAPITULO 1

CONCEPTOS BASICOS

El primer objetivo del presente capítulo es describir los conceptos básicos de un Sistema Orientado a Objetos, tomando como base los del Sistema Smalltalk-80 (ST-80). El segundo, es comparar los sistemas de software tradicionales con los SOO.

El ST-80 fué desarrollado por el grupo de investigación de Xerox, para realizar sistemas de software funcionales e interactivos para aplicaciones en computadoras personales. Asimismo, es un intento serio para llevar a la práctica las ideas de un Sistema Orientado a Objetos.

En el presente trabajo se intenta desarrollar un Sistema Smalltalk Simplificado (SS). El punto de partida para su desarrollo, fué el adoptar los conceptos básicos en los cuales esta basado ST-80 [1]. Por lo que es conveniente resumirlos en este capítulo.

1.1 CONCEPTOS BASICOS

El sistema ST-80 esta basado en un número mínimo de conceptos y el lector deberá de familiarizarse con esos conceptos y explorar las diversas formas en las cuales ellos son aplicados en el sistema. Hay cinco conceptos básicos que deben conocerse en principio: objeto, mensaje, clase, instancia y método. Esas cinco palabras son definidas una en función de la otra. A continuación se describen brevemente estos conceptos.

1.1.1 OBJETOS Y MENSAJES.

Un objeto consiste de una memoria privada y de un conjunto de operaciones. La naturaleza de éstas depende del tipo de

componente que represente. Cuando los objetos son números ejecutan funciones aritméticas. Si los objetos representan estructuras de datos almacenan y accesan información. Cuando los objetos son posiciones y áreas responden a requerimentos acerca de sus relación con otras posiciones y áreas.

Un mensaje es un requerimento a un objeto para llevar a cabo alguna función. Un mensaje indica que operación es deseada, pero no como esa operación va a ser llevada a cabo. El receptor -objeto al cual el mensaje fué enviado- determina como llevar a cabo la operación requerida. Por ejemplo, una suma se realiza enviando un mensaje a un objeto que representa un número. El mensaje especifica que la operación deseada es una suma y también que número será sumado al receptor. El mensaje no especifica como la suma ha de llevarse a cabo. La computación es vista como una capacidad intrínseca de los objetos que pueden ser uniformemente invocados a través de mensajes.

Al conjunto de mensajes a los que un objeto puede responder se le llama su interfase con el resto del sistema. La única forma de interactuar con un objeto es a través de su interfase. Una propiedad crucial de los objetos es que su memoria privada puede ser manipulada sólo por sus propias operaciones. Una propiedad crucial de los mensajes es que son la única forma para invocar una operación de un objeto. Estas propiedades aseguran que la implementación de un objeto no puede depender de los detalles internos de otros objetos, sino únicamente de los mensajes a los cuales ellos responden.

Los mensajes aseguran la modularidad del sistema ya que especifican el tipo de operación deseada, pero no como esa operación será llevada a cabo. Por ejemplo, hay dos representaciones de valores números en el sistema: los enteros y los reales; entienden el mismo mensaje requiriendo el cálculo de la suma con otro número, pero cada representación implica una forma diferente de calcular la suma. Para interactuar con números, o con cualquier otro objeto, sólo se necesita conocer a que mensajes responden y no como se representan.

Una parte importante para diseñar programas en este lenguaje es determinar la representación del objeto y cuales mensajes proveen un vocabulario útil de interacción con otros objetos. Por ejemplo, si se quiere realizar un programa para manejar puntos, primero se elige su representación. Una podría ser sus coordenadas X,Y en el plano. El conjunto de mensajes útiles podría ser: sumar las coordenadas de un punto con las de otro, escalar un punto, crear nuevos punto con ciertas coordenadas, etc..

1.1.2 CLASES E INSTANCIAS.

Una clase describe la implementación de un conjunto de objetos. Al objeto descrito por una clase se le llama Instancia

de esa clase. Una clase describe la estructura de la memoria privada y las operaciones que pueden realizar sus instancias. A cada componente de la memoria privada del objeto se llama Variable de Instancia (VI). Cada instancia de una clase tiene su propio conjunto de variables de instancia, pero generalmente todas tienen el mismo número (excepto las instancias de la clase arreglo). Por ejemplo, la clase que implementa la representación de rectángulos define dos variables de instancia para representar una instancia y cada una de ellas se refiere a otro objeto, llamado el valor de la VI. Uno de los valores de las VI de una instancia de rectángulo, es un objeto punto que representa la esquina inferior izquierda y el otro valor de la VI es un objeto punto que representa la esquina superior derecha. Cada uno de los puntos es una instancia de la clase punto. El hecho de que un rectángulo se represente de ésta forma es información estrictamente interna, no disponible fuera de la definición de la clase rectángulo.

La elección de las VI en el ejemplo anterior no es única, ya que se podría representar con otras VI, por decir algo, una que definiera el centro del rectángulo, otra que indicara su anchura y una más que representara su altura. Esto es válido y no influye en los mensajes a que puede responder ese objeto. En caso que la representación interna de un objeto cambiara, sólo las manipulaciones de esa información dentro de la misma clase cambiaría, sin embargo, los mensajes que se envien a objetos de este tipo en otras clase serían los mismos.

1.1.3 METODOS.

Al conjunto de mensajes a que un objeto puede responder son implementados en la clase del objeto por medio de Métodos. El método ejecuta las operaciones requeridas para un mensaje en particular. Cuando ese mensaje se envia a cualquier instancia de la clase, el método se ejecuta. El método describe como ejecutar la operación en términos de manipular la memoria privada de la instancia (receptor del mensaje). El método puede hacer cambios en su memoria privada, regresar un valor como respuesta al mensaje enviado o enviar una serie de mensajes a otros objetos. Una clase incluye un método para cada tipo de operación que sus instancias pueden ejecutar. Los métodos de un objeto pueden accesar sus propias variables de instancia, pero no los de otros objetos. Por ejemplo, el método que la clase rectángulo utiliza para calcular su centro, los dos puntos referidos por sus VI. Sin embargo, el método no puede accesar los campos de esos puntos. El método especifica a través de mensajes enviados a esos puntos los cálculos requeridos. Los mensajes a esos puntos podrían ser la suma de ellos y después escalarlos por el factor 0.5. El objeto resultante de esas operaciones sería el valor regresado por el método.

1.2 COMPARACION CON LOS SISTEMAS TRADICIONALES.

El interés por implementar un sistema de este tipo surge por los nuevos conceptos que aporta a las Ciencias de la Computación. El principal concepto de este tipo de lenguajes es la integración de datos y su manipulación en una sola entidad. Para explicar esto, se comparará con los sistemas desarrolados con lenguajes tradicionales, tales como Fortran y Pascal, llamados lenguajes orientados a procedimientos.

Un sistema de software realizado con un lenguaje tradicional está compuesto de un conjunto de datos que representan información y un conjunto de procedimientos para manipularlos.

El problema con este punto de vista, es que los datos y procedimientos son tratados como si ellos fueran independientes cuando, en realidad, no lo son. Los procedimientos asumen la forma de los datos que manipulan. Si alguno de estos datos le fuera proporcionado incorrectamente el sistema se comportará de manera extraña.

En un sistema propiamente probado, la selección correcta de procedimientos y datos es siempre hecha a priori. Sin embargo, en un sistema en desarrollo, los datos esperados por un procedimiento pueden ser enteramente diferentes a los previstos. Aún en estos sistemas la selección correcta del procedimiento y datos son siempre hechos por el programador.

Esos problemas han sido atacados en estos sistemas agregando ciertas características al lenguaje de programación. Una de ellas son los tipos de datos, cuya utilidad es que el programador seleccione los datos correctos para un procedimiento en particular. También el programador es notificado cuando utiliza datos erroneos en la llamada al procedimiento. Pero aún con todo esto, estos sistemas pueden caer en comportamientos extraños cuando los datos no son debidamente inicializados por el programador.

En general, los sistemas tradicionales tienen dos tipos de entidades, que representan la información y su manipulación independientemente. En particular, un Sistema Orientado a Objetos tiene sólo un tipo de entidad; esto es, el objeto que representa a ambos. Así, los objetos pueden ser tratados como piezas de datos; sin embargo, como procedimientos, el objeto describe su propia manipulación.

La información se manipula al enviar un mensaje al objeto que la representa; cuando un objeto recibe el mensaje, determina implícitamente como manejarse. Un mensaje incluye un nombre simbólico que indica el tipo de manipulación que se desea, el cual se llama selector. La característica importante de los mensajes es que el selector es sólo un nombre para la operación deseada; describe que es lo que el programador quiere que pase, no como pase. El receptor del mensaje contiene la descripción de como la manipulación actual se lleva a cabo. El programador de

un Sistema Orientado a Objetos envía un mensaje para invocar una manipulación, en lugar de llamar a un procedimiento. Un mensaje nombra la manipulación; un procedimiento describe los detalles de ella.

Los procedimientos tienen nombres también, los cuales se utilizan para invocarlos, sin embargo, hay sólo un procedimiento por cada nombre; esto es, un nombre especifica el procedimiento exacto a ser llamado y que pasará. Por el contrario, un mensaje puede ser interpretado en diferentes formas por diferentes receptores; un mensaje no determina exactamente que pasará; el receptor es el que lo hace.

Los objetos son tratados desde dos puntos de vista: externa e interna. Externa es como los demás objetos interactuan con un objeto en particular a través de mensajes. Interna es como lo ve el programador que implementa su comportamiento a través de métodos.

Al conjunto de mensajes a que un objeto puede responder se le llama protocolo. La vista externa del objeto no es otra cosa más que su protocolo; la vista interna de un objeto es como en los sistemas tradicionales.

Un objeto tiene un conjunto de variables que se refieren a otros objetos, llamadas variables privadas. También tiene un conjunto de métodos que describen que hacer cuando se recibe un mensaje. Los valores de las variables privadas juegan el papel de los datos y los métodos juegan el papel de los procedimientos. Esta distinción entre datos y procedimientos es localizada estrictamente en la vista interna del objeto.

En el capítulo 2, se desarrollarán los conceptos de objeto y mensaje aplicados a las diversas expresiones de lenguaje; posteriormente, en el capítulo 3, se desarrollarán los de clase, instancia y métodos en la construcción de programas de aplicación.

1.3 REFERENCIAS.

[1] A. Goldbert, D. Robson; Smalltalk-80 The Language and its Implementation;E.E.U.U.. Addison-Wesley Pub. (1983), Cap. 1 pp 6-16.

CAPITULO 2

EXPRESIONES

En este capítulo se analizarán los diversos tipos de expresiones válidas para escribir mensajes. También se describen las formas de referirse a objetos y los tipos de mensajes, desde el punto de vista sintáctico, que pueden ser mandados a éstos.

2.1 EXPRESIONES.

Una expresión es una secuencia de caracteres que describe un objeto llamado el valor de la expresión. La sintaxis presentada en éste capítulo explica la secuencia de caracteres que forman expresiones válidas. Hay cuatro tipo de expresiones en el lenguaje; esta clasificación es similar a la del sistema ST-80 [1] y las diferencias serán remarcadas a lo largo de este capítulo. A continuación se presenta esta clasificación:

1. **Las literales** describen ciertos objetos constantes, tales como números, caracteres, cadenas de caracteres, arreglos, etc..

2. **Los nombres de variables** describen las variables accesibles. El valor de un nombre de variable es el valor actual de la variable con ese nombre.

3. **La expresión de mensaje** describe mensajes a receptores. El valor de una expresión de mensaje se determina por el método que el mensaje invoca. Ese método se encuentra en la clase del receptor.

4. **Los bloques** describen objetos que representan actividades diferidas, y son usados principalmente para implementar estructuras de control.

De las cuatro expresiones listadas arriba sólo los nombres de variables son dependientes del contexto en donde se usen; cada clase posee sus propios nombres de variables que sólo por ella son accesibles.

2.2 LITERALES.

Existen cuatro tipos de objetos que pueden ser referidos por literales. Como el valor de ésta es siempre el mismo objeto, es llamada literal constante. Los tipos son:

1. Números
2. Caracteres
3. Cadena de caracteres
4. Arreglos constantes.

Los números son objetos que representan valores numéricos. La representación literal de un número es una secuencia de dígitos que pueden ser precedidos por un signo menos y/o seguidos por un punto decimal y otra secuencia de dígitos. Ejemplos de estos números válidos son

```
3
55.7
-0.44
7.0
1921
```

En el sistema St-80, los números se pueden representar en diversas bases; a saber, en base octal, hexadecimal y notación científica.

Los caracteres son objetos que representan un símbolo del alfabeto. Una literal caracter consiste de un signo de dolar seguido del caracter, por ejemplo:

```
$A
$9
$(
$$
```

Las cadenas son objetos que representan una secuencia de caracteres. Esta secuencia esta delimitada por apóstrofes, por ejemplo

```
'hola'
'Esta es una cadena'
```

Cualquier caracter puede ser incluido en la secuencia. Si un apóstrofe es incluido en la secuencia, debe ser duplicado para evitar confusión con los delimitadores. Por ejemplo

```
'Restaurante Manolo''s'
```

La secuencia de caracteres debe terminar en el mismo renglón en donde se abrió el primer apóstrofe, y si no se cumple con esto, se notifica un error al usuario.

Un arreglo es un objeto que representa una estructura de datos simple, cuyo contenido puede ser accesado a través de un índice que va de uno a la longitud del arreglo. Los arreglos responden a mensajes que invocan acceso o modificación a uno de sus componentes. La representación literal de un arreglo es una secuencia de otras literales -números, caracteres, cadenas, arreglos- delimitadas por paréntesis y precedido por el signo "#". Las demás literales son separadas por espacios. Los arreglos anidados no son precedidos por el signo "#". Por ejemplo, un arreglo de tres números es descrito por la expresión

```
#( 1 2 3 )
```

Un arreglo de cuatro cadenas es descrito por la expresión

```
#('alimentos' 'renta' 'transporte' 'impuestos')
```

Un arreglo de dos arreglos, una cadena y un número por

```
#( ('uno' 1) ('dos' 2) 'hola' 12.3 )
```

2.3 VARIABLES.

La memoria disponible a un objeto es a través de variables, en donde cada una recuerda un objeto y puede ser usada en las expresiones para referirse a ese objeto. El acceso a esos objetos representados por la variable es determinado por el lugar en donde se use el nombre de la variable. Por ejemplo, las variables que representan los campos de una instancia no pueden ser accesadas por otros objetos, porque esas variables sólo pueden ser accesadas por los métodos de la clase de ese objeto. Por tal razón, existen variables privadas y variables globales; las primeras sólo son accesibles a un objeto y las últimas pueden ser a varios o a todos los objetos. Las diferentes clases de

variables disponibles en el lenguaje serán vistas posteriormente cuando se explique la estructura de una clase. Por ahora se supone que una variable es un identificador.

2.3.1 ASIGNACIONES A VARIABLES.

El objeto referido por una variable es cambiado usando una expresión de asignación. Esta se forma anteponiendo a una expresión el nombre de una variable y el operador de asignación (:=). Algunos ejemplos de expresiones de asignación son:

```
coordx := 100
nombre := 'OCTAVIO JIMENEZ MEDINA'
```

También se puede cambiar el valor de una serie de variables al mismo tiempo. Por ejemplo

```
temp1 := temp2 := temp3 := 0
```

En el sistema ST-80, es una flecha orientada a la izquierda, caracter que no existe en la mayoría de las terminales, por lo que se adopto el operador (:=).

2.3.2 PSEUDO-VARIABLES.

Una pseudo-variable es un identificador que se refiere a un objeto (al igual que una variable), y lo que la diferencia de una variable, es que su valor no puede ser cambiado con una expresión de asignación. Algunas de las pseudo-variables en el sistema son constantes; siempre se refieren al mismo objeto. Esta pseudo-variables son: nil, true y false.

1. nil - Se refiere a un objeto usado como el valor de una variable cuando otro objeto no es apropiado. Las variables que no han sido inicializadas se refieren a nil.

2. true - Se refiere a un objeto que representa la verdad lógica. Es usada como una respuesta afirmativa a un mensaje que realiza una proposición lógica de verdadero o falso.

3. false - Se refiere a un objeto que representa la falsedad lógica. Es usado para respuestas negativas a un mensaje que realiza una proposición de verdadero o falso.

Los objetos llamados true y false son llamados objetos booleanos.

Otra importante pseudo-variable en el sistema es Self cuyos valores son diferentes dependiendo del lugar en donde sea usada. Se usa en los métodos de una clase y se refiere al objeto que invocó un método (receptor del mensaje).

2.4 EXPRESION DE MENSAJE.

Los mensajes representan la interacción entre los componentes del sistema. Un mensaje invoca una operación del receptor. Una expresión de este tipo tiene la estructura siguiente

```
RECEPTOR <= MENSAJE
         ^
Operador de mensaje
```

El receptor puede ser alguna de las cuatro expresiones mostradas al principio de éste capítulo, además para claridad en los programas se permite que esas expresiones puedan ir entre paréntesis. El mensaje describe al selector y posiblemente algunos argumentos. Los argumentos son descritos también por otras expresiones, sin embargo, si la expresión es de mensaje debe de ir forzosamente entre paréntesis. El selector se especifica literalmente, y determina que operación del receptor será invocada. Los argumentos son otros objetos que son tomados en cuenta en la operación de selección. Hay tres patrones de mensajes establecidos:

1. Mensaje unitario. Un mensaje sin argumentos es llamado mensaje unitario. Por ejemplo, para requerir la coordenada x de un punto cualquiera, se realiza el siguiente mensaje:

```
PUNTO1 <= X
```

   Estos mensajes son llamados unitarios porque sólo un objeto, el receptor es requerido. El selector de este tipo de mensaje es un identificador y se le llama "selector unitario". En el ejemplo, el receptor es el objeto punto representado por la variable PUNTO1 y el selector unitario es "X". Le corresponde al receptor decidir como responder al mensaje (en este caso regresa el valor de la coordenada X del objeto PUNTO1).

2. Mensajes Binarios. Este tipo de expresión de mensaje tiene sólo un argumento. El selector de este mensaje es un caracter dentro de un conjunto de caracteres especiales

definidos en el sistema, ya que los mensajes binarios se utilizan en general para operaciones aritméticas; algunos ejemplos son:

```
3 <= + 4
I <= + 1
I <= * J
```

En el primer ejemplo, el mensaje binario "+ 4" se envía al objeto que representa la literal "3". El selector es "+" y le indica al receptor que tiene que calcular la suma y regresar como valor del mensaje el resultado. El argumento indica la cantidad a ser sumada. Se le llama binario porque se involucran dos objetos en el mensaje: el receptor y un argumento.

3. Mensajes múltiples. Un mensaje con uno o más argumentos, es llamado mensaje múltiple. El selector de un mensaje múltiple esta compuesto de uno o más subselectores, uno precediendo a cada argumento. Un subselector es un identificador concatenado con el caracter ':'. Un ejemplo de una expresión de mensaje múltiple es:

```
PUNTO1 <= X: VALORX Y: VALORY
```

El mensaje X: VALORX Y: VALORY se envía al objeto que representa la variable PUNTO1. El selector es X:Y: y los argumentos VALORX y VALORY. Este mensaje invoca la inicialización del PUNTO1 con la coordenada X igual a VALORX y la coordenada Y igual a VALORY.

En el sistema St-80, no existe el operador de mensaje, este fué adoptado del lenguaje TM; el operador de mensaje permite escribir una serie de mensajes en cadena a la vez en una misma expresión, cosa que en el sistema ST-80 no se puede hacer, ya que existe una serie de precedencias entre los diversos tipos de mensajes, haciendo la escritura de una expresión más complicada para el usuario. Es decir, en el sistema SS el usuario puede escribir en una misma expresión mensajes unario, binarios y múltiples en el orden que se desee; y en el sistema ST-80 deben de guardar un orden establecido para que la evaluación sea correcta.

Los mensajes proveen dos formas de comunicación. En la primera forma están involucrados el selector y los argumentos, que transmiten información al receptor acerca de que tipo de respuesta hacer. La segunda forma corresponde al receptor, que transmite información regresando un objeto que llega a ser el valor de la expresión de mensaje. Si una expresión de mensaje es incluida en una expresión de asignación, el objeto regresado por

el receptor llega a ser el nuevo objeto referido por la variable del lado izquierdo. Por ejemplo, la expresión

```
SUMA := 3 <= + 4
```

hará que 7 sea el nuevo valor de la variable SUMA.

El número referido por INDICE será incrementado por la expresión

```
INDICE := INDICE <= + 1
```

Aun cuando la información que regrese el receptor no sea necesaria, éste siempre regresa un valor de la expresión del mensaje. El regreso de un valor indica que la respuesta al mensaje se llevó a cabo correctamente. No necesariamente el valor del mensaje debe guardarse, en muchos casos el valor regresado es el receptor de otro mensaje. Por ejemplo, en el mensaje

```
VENTANA <= MARCO <= CENTRO
```

el selector unitario MARCO se envía primero al objeto representado por VENTANA. Después, el mensaje unitario CENTRO se envía al resultado de la expresión VENTANA <= MARCO (es decir, el objeto regresado de la respuesta del objeto VENTANA al mensaje MARCO). Otro ejemplo es el siguiente

```
INDICE <= + OFFSET <= * 2
```

en esta expresión el resultado de enviar el mensaje binario "+ OFFSET" al objeto llamado INDICE es el receptor del mensaje binario "* 2".

Puede haber tantos mensajes en cadena como se desee. El rastreo de un mensaje es hecho de izquierda a derecha y no se toma en cuenta ninguna precedencia, sólo el orden en el cual los mensajes aparecen, ya sean unitarios, binarios o múltiples.

Los paréntesis pueden ser utilizados para cambiar el orden de evaluación. Un mensaje entre paréntesis se envía antes que cualquier otro mensaje fuera de estos. Si el ejemplo anterior fuera escrito como se expresa a continuación:

```
INDICE <= + ( OFFSET <= * 2 )
```

el resultado del mensaje binario "* 2" al objeto OFFSET es usado como el argumento del mensaje binario con receptor INDICE y selector "+". Pueden ir tantos mensajes anidados entre paréntesis como se desee.

El programador es libre de formar expresiones en varias formas usando espacios, tabs y return. Por ejemplo, los mensajes múltiples se escriben con frecuencia con cada argumento en una línea diferente:

```
.PUNTO <= creaX: (P1 <= X)
          Y: (P2 <= Y)
```

(el objeto .PUNTO se refiere a la clase punto, en el capitulo 3 se explicará).

2.4.1 MENSAJES EN CASCADA.

Una expresión de mensajes en cascada describe una secuencia de mensajes a ser enviados al mismo objeto. Los mensajes van separados por comas, por ejemplo:

```
.ARRAY <= crea: 3,EN: 1 PON: 'FERNANDO',
                  EN: 2 PON: 'EUGENIA',
                  EN: 3 PON: 'LIDIA'
```

Tres mensajes EN:PON: se envían al resultado de .ARRAY<=crea:3 o sea, un arreglo de tres elementos. Sin cascada, esto requeriría de cuatro expresiones y una variable. Por ejemplo, las cuatro expresiones siguientes, separadas por punto y coma, tienen el mismo efecto que la expresión de cascada, esto es:

```
TEMP := .ARRAY <= crea: 3;
TEMP <= EN: 1 PON: 'FERNANDO';
TEMP <= EN: 2 PON: 'EUGENIA';
TEMP <= EN: 3 PON: 'LIDIA'
```

2.5 BLOQUES.

Un bloque representa una secuencia de acciones diferidas. Se compone de una secuencia de expresiones separadas por punto y coma y delimitadas por paréntesis angulares; Por ejemplo:

```
[ indice := indice <= + 1 ]
```

o

```
[ indice := indice <= + 1 ;
  arreglo <= en: indice pon: 0 ]
```

son bloques válidos.

La última expresión no debe terminar con punto y coma.

Cuando un bloque se encuentra, las expresiones entre los paréntesis angulares no se ejecutan inmediatamente, ya que el valor de un bloque es un objeto que puede ejecutar las expresiones más tarde, cuando sea requerido para realizarlo. Ese objeto es tratado como literal por el sistema (en otros sistemas al bloque se le llama método literal) y se usa como tal. Es decir, un bloque puede usarse como argumento de un mensaje. Por ejemplo

```
4 <= repite: [ i := i <= + 1 ]
```

El receptor del mensaje, en este caso el número 4 toma una acción sobre el bloque que se le pasa como argumento. Esto es, la de evaluar cuatro veces el bloque.

A una variable puede asignarsele un bloque, como se muestra en la siguiente expresión:

```
incrementai := [ i := i <= + 1 ]
```

es válida.

La secuencia de acciones descritas en el bloque toman lugar cuando al objeto que representa el bloque se le manda el mensaje unitario "ejecuta". Por ejemplo, las siguientes acciones tienen el mismo efecto

```
i := i <= + 1
```

y

```
[ i := i <= + 1 ] <= ejecuta
```

y

```
incrementai <= ejecuta
```

La última expresión asume que la asignación mencionada arriba se realizó primero.

El objeto que se regresa como resultado de mensaje "ejecuta", es el valor de la última expresión en la secuencia; a éste valor se le llama valor del bloque. Así, si la expresión

```
sumbloque := [ i <= + 1 ]
```

se ejecuta, otra forma de incrementar i es evaluar

```
i := sumbloque <= ejecuta
```

2.5.1 ESTRUCTURAS DE CONTROL.

Las estructuras de control no secuenciales fueron implementadas en el sistema usando bloques. Las dos estructuras más comunes son la selección condicional y repetición condicional. Estas se forman usando mensajes.

La selección condicional de una actividad es provista por un mensaje a un objeto booleano con el selector iftrue:iffalse: y dos bloques como argumentos. Los únicos objetos que entienden este mensaje son TRUE y FALSE. Estos, tienen respuestas contrarias: TRUE manda el mensaje "ejecuta" al primer argumento e ignora el segundo; FALSE manda el mensaje "ejecuta" al segundo argumento e ignora el primero. Por ejemplo, la siguiente expresión asigna a la variable paridad. un 0 si el valor del número es divisible entre 2 y 1 en caso contrario.

```
( numero <= mod: 2 <= eq: 0 ) <=
   iftrue: [ paridad := 0 ]
    iffalse: [ paridad := 1 ]
```

El resultado del mensaje es el valor del bloque ejecutado. El ejemplo anterior pudo escribirse de la siguiente manera

```
paridad := (numero<=mod: 2<=eq: 0) <=
             iftrue: [0]
            iffalse: [1]
```

Hay otro mensaje que sólo especifica una acción condicional. El selector de ese mensaje es iftrue: con un bloque como argumento. Por ejemplo:

```
indice <= le: (arreglo <= longitud) <=
     iftrue: [ arreglo <= en: indice pon: 0 ]
```

Si el bloque no se ejecuta el valor del mensaje es NIL.

La repetición condicional de una actividad es provista por un mensaje a un bloque con el selector whiletrue: y otro bloque como argumento. El bloque receptor se envía a si mismo el mensaje "ejecuta" y si su valor es TRUE, se envía el mensaje "ejecuta" al bloque argumento. Estas acciones se repiten hasta que el valor del bloque receptor sea FALSE. Por ejemplo, una repetición condicional podría ser usada para inicializar los elementos de un arreglo

```
indice := 1
[ indice <= le: (arreglo<=longitud) ] <=
   whiletrue: [ arreglo <= en: indice pon: 0;
                indice := indice <= + 1 ]
```

El programador es libre de implementar sus propias estructuras de control, las mencionadas anteriormente son las que forman parte del sistema.

2.5.2 ARGUMENTOS DE BLOQUES.

Para realizar estructuras de control no secuenciales más faciles de expresar, los bloques pueden recibir uno o más argumentos. Los argumentos de bloques se especifican incluyendo identificadores precedidos por dos puntos al principio del bloque. Los argumentos son separados de las expresiones por una barra vertical "|". El siguiente ejemplo describe un bloque con un argumento

```
[:arreglo | total:=total<=+(arreglo <= longitud)]
```

Un uso común de bloques con argumentos, es implementar funciones a ser aplicadas a todos los elementos de una estructura de datos. Por ejemplo, los diferentes tipos de objetos que representan estructuras de datos responden al mensaje DO:, el cual toma un bloque como argumento. El objeto que recibe el mensaje DO: evalua el bloque una vez por cada elemento contenido en la estructura de datos. El argumento toma el valor de cada elemento de la estructura de datos en cada evaluación del bloque. El siguiente ejemplo, calcula la sumas de los cuadrados de los primeros cinco números primos. El resultado es el valor de suma.

```
suma := 0;
#(2 3 5 7 11)<=
do: [:primo| suma := suma <=+ (primo<=* primo)]
```

El objeto que implementa esta estructura de control transmite los valores de los argumentos del bloque al enviar a éste el mensaje ejecuta:. Un bloque con un argumento responde a ejecuta: tomando el argumento un valor igual al enviado en el mensaje ejecuta: y posteriormente ejecuta las expresiones del bloque. Por ejemplo, al evaluar las expresiones siguientes resulta en que la variable TOTAL tiene el valor 7.

```
sumaLong := [:arreglo| total:=total<=+
                          (arreglo<=longitud)];
total := 0;
sumaLong <= ejecuta:  (1 2 3);
sumaLong <= ejecuta:  ($a $b);
sumaLong <= ejecuta:  ('hola' 'que')
```

bloques pueden tener más de un argumento

```
[ :x :y | (x<=*x)<=+(y<=*y) ]
```

Este bloque se le tiene que enviar el mensaje ejecuta:ejecuta:.

Si un bloque recibe un número menor de argumentos que los declarados, un mensaje de error se notifica.

Una vez analizadas las diferentes expresiones, se está en condiciones de utilizarlas para formar métodos, cuya estructura y aplicación se desarrolla en el capítulo 3.

2.6 REFERENCIAS.

[1].- A. Goldberg, D. Robson; Smalltalk-80 The language and its Implementation; E.E.U.U.. Addison-Wesley Pub. (1983), Cap. 2 pp 18-38.

CAPITULO 3

CLASES

En el presente capítulo se define la manera de construir una nueva clase en el sistema. En el sistema ST-80 la definición de una clase se realiza de una manera totalmente interactiva, por lo que no hay necesidad de tener una estructura sintáctica de una clase; la principal estructura es la definición de los métodos; así, la definición de la clase se simplifica, ya que no se tiene que pensar en la estructura final de una clase para definirla, si no que, se van agregando métodos para contestar mensajes de acuerdo a las necesidades del programador. En el sistema SS no existe un sistema interactivo para la definición de la clase, principalmente por la complejidad de implementarlo y además por no ser el objetivo del presente trabajo. Para sustituir ésta deficiencia, el sistema SS tuvó que adoptar una estructura sintáctica para la definición de la clase; de esta manera, el programador se sujeta a esa sintáxis para contruir su clase, y no existe un modo para agregar métodos una vez que la clase este instalada en el sistema.

La estructura sintáctica que se adoptó se basa en la definición del lenguaje TM[1], con algunas adaptaciónes para el sistema SS; la principal es la definición de un método, que tuvo que adaptarse a la sintaxis de las expresiones vistas en el capítulo 2. Algunas otras modificaciones serán mencionadas explícitamente en el texto. Los diagramas de sintaxis del sistema SS, se presentan en el apéndice A.

Una vez que la clase se define en el sistema SS, conceptualmente es similar a una clase del sistema ST-80 [2], es decir, los mecanismos son los mismos para accesar esa clase, crear instancias, enviar mensajes y una serie de conceptos que se tratan a lo largo de éste capítulo.

En principio, se tratará la declaración y referencia de los diferentes objetos del sistema; más tarde, se verá la estructura de una clase, en donde se sientan las bases de la escritura de un

programa para la definición de la clase; a continuación, se analiza la estructura y definición de los métodos haciendo énfasis en como los mensajes se utilizan para accesar funcionalmente objetos y además, se verán las variables a que tiene acceso un método (su medio ambiente). En la penúltima sección, se trata el concepto de subclase, el cual es importante, puesto que de una forma u otra, todas las clases son subclase de otra llamada superclase, también se analiza el concepto de herencia entre clases y los mecanismos para selección del método dado un mensaje. Finalmente, en la última sección, se analiza el concepto de metaclase, las cuales son las administradoras de clases.

A lo largo del capítulo se desarrolla una aplicación que va creciendo en complejidad de acuerdo a como se definan nuevos conceptos y estructuras. La aplicación consiste de un sistema para el control de ingresos y egresos de una empresa, teniendo en cuenta los conceptos por lo que sale o entra dinero, las cantidades que son deducibles de impuestos, etc..

3.1 REFERENCIA Y DECLARACION DE VARIABLES.

Cada clase tiene un nombre que describe el tipo de componente que sus instancias representan, además de proveer una forma para referirla en las expresiónes. Las clases son un componente dentro del mismo sistema, esto es, también se representan por objetos asociados al nombre de la clase. El nombre de la clase llega a ser automáticamente el nombre de una variable pública, es decir, que es accesible por todos los objetos. Para establecer la diferencia sintáctica entre clases y sus instancias cuando sean referidas en expresiónes, se estableció la siguiente regla sintáctica

```
<nom-clase> ::= . <IDENTIFICADOR>
<nom-inst>  ::= <IDENTIFICADOR>
```

O sea, una clase se referencia anteponiendo un punto al nombre de la clase y, por otra parte, a una instancia sólo por su nombre.

En la presente implementación, a toda instancia se le puede declarar asociándole una clase que lo administra. Para esto, se sigue la siguiente regla sintáctica

```
<inst-decl> ::= <nom-inst> . <nom-clase>
```

Esta regla se aplica al declarar variables, tales como las variables de instancia, variables temporales, etc., como se verá en este capítulo. Si se declaran de esta manera, el sistema verifica los tipos de datos, por una parte en las asignaciones y

por la otra, en el paso de argumentos en los mensajes. La asociación a un administrador es opcional; si se omite, a esa variable se le puede asignar cualquier objeto (variable genérica).

Una expresión que se usa en todo el capítulo es la creación de nuevos objetos al enviar el mensaje CREA a la clase, por ejemplo

```
.punto <= crea
```

regresa una instancia de la clase Punto con dos campos cuyos valores no estan inicializados y apuntan a NIL. Cuando una instancia es creada, automáticamente comparte los métodos de la clase que recibió el mensaje de creación de una instancia.

3.2 ESTRUCTURA DE UNA CLASE.

En el código fuente de una clase, el usuario define como se llamará esta clase; su relacion con otras; la estructura de los objetos a los cuales administrará y el conjunto de mensajes a que es capaz de responder (la interfase). Para tal fín, el código fuente de la clase posee una estructura y ciertas reglas sintácticas que el usuario debe seguir para construir la clase en el sistema. En términos generales una clase tiene la siguiente estructura (con mayúscula se presentan las palabras reservadas):

```
CLASS_MOD
    + Identificación de la clase.

    + [ superclase ]

    + [ Declaración de variables de instancia ]

    + [ Declaración de variables de clase ]

    + Métodos (interfase)
END_CLASS
```

La estructura anterior, es la estructura general de una clase en el lenguaje TM [1].

Una aplicación de la estructura de una clase es dada en el ejemplo 3.1. Hay que hacer notar que esta implementación aún no es completa; a lo largo del capítulo se verá como toma su forma final. Esto es con el fín de no mezclar conceptos que pudieran confundir al lector.

3.2.1 IDENTIFICACION DE LA CLASE.

En esta sección se define el nombre de la clase de la siguiente forma:

NAME <nom-clase>

El nombre de la clase no debe de existir en el sistema.

3.2.2 SUPERCLASE.

Opcionalmente, una superclase puede ser asociada a la clase, de la siguiente manera:

SUPERCLASS <nom-clase>

El concepto de superclase es visto en detalle en la sección 3.4.

3.2.3 DECLARACION DE VARIABLES DE INSTANCIA.

En esta parte se declaran las variables de instancia que forman la estructura de las instancias de esta clase. Los nombres no deben repetirse y son separados por espacios. La sintaxis es la siguiente:

VAR_INS <decl-var> [<decl-var> ...]

En el ejemplo 3.1 la declaración de variables de instancia son tres, esto es:

```
VAR_INS     dinero
            ingresos
            egresos
```

EJEMPLO 3.1

```
CLASS_MOD
   NAME          HistoriaFin
   VAR_INS       dinero
                 ingresos
                 egresos

TO_OBJECT

    recibir: cantidad.fix de: fuente.str =>
      ingresos <= en: fuente
                    pon: (self <= recibidoTot: fuente <= +
cantidad);
      dinero := dinero <= + cantidad
    END

    gastar: cantidad.fix por: concepto.str =>
      egresos <= en: concepto
                 pon: (self <= gastoTot: concepto <= + cantidad);
      dinero := dinero <= - cantidad
    END

    dinero => <- dinero END

    recibidoTot: fuente.str =>
      ingresos <= existe: fuente <=
            iftrue: [ <- ingresos <= en: fuente ]
            iffalse: [ <- 0 ]
    END

    gastoTot: concepto.str =>
      egresos <= existe: concepto <=
        iftrue: [ <- egresos <= en: concepto ]
      iffalse: [ <- 0 ]
    END

    balanceini: cantidad.fix =>
        dinero := cantidad;
        ingresos := .diccionario <= crea;
      egresos := .diccionario <= crea
   END
END_OBJ
END_CLASS
```

Una instancia de HistoriaFin utiliza dos diccionarios, uno para almacenar la razón y otro para la cantidad total de lo gastado y recibido, y una variable más para recordar el dinero en efectivo.

- EGRESOS se refiere a un diccionario que asocia la razón con la cantidad del gasto.
- INGRESOS se refiere a un diccionario que asocia la razón con la cantidad de lo recibido.
- DINERO se refiere a un número que representa la cantidad de dinero en efectivo.

Cuando alguna expresión dentro de un método de esta clase usa una variable de instancia, al evaluarse esa expresión se refieren al valor de esa variable de la instancia que recibe el mensaje.

Cuando una nueva instancia se crea, al enviar un mensaje a la clase, tiene un nuevo conjunto de variables de instancia. Esas variables son inicializadas como se especifique en el método asociado con el mensaje de creación de instancias.

Por ejemplo, para poder utilizar los mensajes a HistoriaFin, una expresión como la siguiente debe haber sido previamente evaluada

```
finCasa := .HistoriaFin <= crea <= balanceIni: 50000
```

El primer mensaje crea un objeto cuyas tres variables de instancia se refieren a NIL. El segundo mensaje hace que la nueva instancia tenga las variables de instancia con un valor mas apropiado.

3.2.4 DECLARACION DE VARIABLES DE CLASE.

Aqui se declaran las variables que van a ser compartidas por todas las instancias de la clase. No deben repetirse y son separadas por espacios. La sintaxis es como sigue

```
DCL <decl-var> [ <decl-var> ... ]
```

3.2.5 DECLARACION DE METODOS.

Finalmente, en esta sección se definen el conjunto de

mensajes a los que la clase e instancias de ésta pueden responder. A la descripción de una respuesta a un mensaje se le llama método. Existen dos clases de métodos: el primero que responde a mensajes enviados a la clase y el segundo que responde a los enviados a las instancias de la clase. Para establecer la diferencia sintáctica los primeros están delimitados por las palabras reservadas TO_ITSELF...END_IT y los últimos por las palabras reservadas TO_OBJECT...END_OBJ, debiendo conservar ese orden. Así, los métodos de una clase se definirán como sigue:

```
TO_ITSELF
  método
    .
    .
  método
END_IT
TO_OBJECT
  método
    .
    .
  método
END_OBJ
```

No necesariamente ambos conjuntos de métodos deben declararse, pero al menos uno sí. Los métodos pertenecientes a la clase, sintácticamente son iguales que los de las intancias, sus diferencias semánticas se presentan en la sección 3.5.

3.3 METODOS.

Un método describe una secuencia de acciones llevadas a cabo cuando un mensaje con un particular selector es recibido por una instancia de una clase. Esas acciones consisten en enviar otros mensajes, asignar variables y regresar algun valor al mensaje original.

La estructura de un método se muestra a continuación:

```
Patrón del mensaje =>
   [ Declaración temporales ]
   expresión [ ; expresión ... ]
END
```

La estructura del método es similar a la del lenguaje TM, sólo el patrón de mensaje tiene diferencias ya que se tuvo que adaptar a la sintaxis de las expresiones de mensaje.

Como ejemplo de un método, se repite uno de los métodos del ejemplo 3.1:

```
        gastar: cantidad por: razon =>
           egresos <= en: razon
                     pon: self <= gastoTot: razon <= +
cantidad ;
           dinero := dinero <= - cantidad
      END
```

A continuación se describen los componentes de la estructura de un método.

3.3.1 PATRON DEL MENSAJE.

En esta sección se describe la estructura del mensaje. Como se presentó en el capítulo 2, hay tres tipos de mensajes:

1. Mensajes Unitarios.- Son mensajes sin argumentos. Consiste de un identificador llamado selector unario.

2. Mensajes binarios.- Son patrones que responden a mensajes con sólo un argumento y cuyo selector pertenece a un conjunto de caracteres especiales y se le llama "selector binario". Un ejemplo común de selector binario son los símbolos aritméticos (+, -, *, /).

3. Mensajes múltiples.- Consisten de uno o varios argumentos y un selector que es hecho de una serie de identificadores, uno precediendo a cada argumento.

El patrón del mensaje gastar:cantidad por:razon, indica que éste método será usado en respuesta a todos los mensajes con el selector gastar:por:.

La primera expresión en el cuerpo del método suma la nueva cantidad a la cantidad ya gastada por la razón indicada. La segunda expresión es una asignación que decrementa el dinero en efectivo por la nueva cantidad.

Como se vió en el ejemplo anterior, el patrón del mensaje contiene un selector y los nombres de los argumentos. Un patrón de mensaje entiende cualquier mensaje que tenga el mismo selector. Una clase tiene sólo un método con un selector dado en

su patrón de mensaje. Cuando un mensaje se envía, el método con igual patrón de mensaje se selecciona de la clase del receptor. Es entonces cuando las expresiones en el método seleccionado se evaluan una tras otra, y por último, se regresa un valor a quien envió el mensaje.

3.3.1.1 ARGUMENTOS DEL PATRON DE MENSAJE. -

Los nombres de los argumentos declarados en el patrón de mensaje del método son pseudo-variables refiriendose a los argumentos del mensaje actual. Si el método que se muestra arriba se invocara por la expresión

```
finCasa <= gastar: (3000 <= + impuesto) por: 'alimentos'
```

la pseudo-variable cantidad se refiere al valor regresado al evaluar la expresión encerrada entre paréntesis y la pseudo-variable razon, se refiere a la cadena 'ALIMENTOS'.

Puesto que los argumentos son pseudo variables, sus valores no pueden alterarse por medio de una expresión de asignación.

Los argumentos pueden declararse en el patrón del mensaje asociándoles un administrador; si éste es el caso, antes de ejecutar el método se revisa el tipo del objeto pasado como argumento y si no corresponde al declarado, se notifica un mensaje de error y la ejecución del método se suspende.

3.3.2 DECLARACION TEMPORALES.

Otra componente importante del método son las variables temporales. Estas variables se utilizan durante la ejecución del método y "desaparecen" cuando éste termina su ejecución. No deben repetirse y son separados por espacios. La sintaxis es:

```
DCL <decl-var> [  <decl-var> ... ]
```

Como ejemplo, se escribe el método para gastar:por: usando una variable temporal para recordar el gasto anterior.

```
gastar: cantidad.fix por: razon.str =>
   dcl gastoAnt.fix ;
   gastoAnt := self <= gastoTot: razon ;
   egresos <= en: razon
             pon: gastoAnt <= + cantidad ;
   dinero := dinero <= - cantidad
END
```

Los valores de las variables temporales son accesibles sólo a expresiones en el método donde son declaradas y son olvidadas cuando el método termina su ejecución.

3.3.3 EXPRESIONES.

Finalmente, una secuencia de expresiones complementan la estructura del método. En las expresiones se involucran referencias a clases y variables. Cuando se referencia a cualquiera de éstas, ya deben de haber sido previamente declaradas. A continuación se presenta un resumen de las variables referenciables por expresiones en un método.

- Las variables de instancia. Son aquellas variables que sirven para referenciar un campo de un objeto. Sólo pueden ser usadas en la clase de ese objeto. Esta es la única manera de referenciar los campos de un objeto.

- La Pseudo-variable SELF. Es una variable muy importante que se puede utilizar en cualquier método de cualquier clase. SELF se refiere al receptor del mensaje que invocó al método. Es llamada Pseudo-variable porque su valor puede ser accesado como una variable cualquiera, pero su valor no se puede cambiar usando una expresión de asignación.

- Argumentos de mensaje. Son pseudo-variables declaradas en el patrón del mensaje del método y sólo existen durante la ejecución del método.

- Variables temporales. Son declaradas dentro del cuerpo de un método y son usadas durante la ejecución del mismo. Cuando éste termina dejan de existir.

- Variables de clase. Son compartidas por todas las instancias de una clase y por la clase misma. Son declaradas al principio de la clase.

- Variables públicas. Son compartidas por todas las instancias de todas las clases del sistema.

3.3.3.1 RETORNO DE VALORES. -

El método para GASTAR:POR: no especifica cual será el valor del mensaje, a falta de éste, el receptor se regresa como el valor del mensaje. Cuando otro valor diferente a ese quiera regresarse, deben usarse una o más expresiones de retorno en el método. Cualquier expresión puede regresarse en una expresión de retorno precediéndola con el operador (<-). Como ejemplos, para

regresar el valor de una variable se escribe

```
<- dinero
```

Para regresar el valor de otro mensaje

```
<- egresos <= en: razon
```

Un objeto literal puede regresarse como en

```
<- 0
```

Aún, una expresión de asignación puede regresarse en una expresión de retorno, como en

```
<- indice := 0
```

En este caso, la asignación se ejecuta primero y el nuevo valor de la variable se regresa.

Un ejemplo del uso de una expresión de retorno es la siguiente implementación del método gastoTot: correspondiente a la clase HistoriaFin:

```
gastoTot: razon.str =>
   ( egresos <= existe: razon )
     iftrue: [ <- egresos <= en: razon ]
    iffalse: [ <- 0 ]
END
```

Este método consiste de sólo un mensaje condicional. Si dentro de los egresos existe la razón (razon), el valor asociado a ésta se regresará; en caso contrario, el valor del mensaje gastoTot: será de cero.

3.3.3.2 LA PSEUDO-VARIABLE SELF. -

Además de las pseudo-variables referidas por los argumentos de un mensaje, los métodos tienen acceso a una pseudo-variable llamada SELF que se refiere al receptor del mensaje. Por ejemplo, en el método para gastar:por:, el mensaje gastoTot: es enviado al receptor del mensaje gastar:por:. Cuando se ejecuta este método, el mensaje gastoTot: se envía al mismo objeto

(SELF) que recibió gastar:por:. Al resultado de ese mensaje se le envía el mensaje +cantidad y, el resultado de éste se utiliza como segundo argumento del mensaje en:pon:.

La pseudo-variable SELF puede ser utilizada para implementar funciones recursivas. Por ejemplo, el mensaje FACTORIAL es entendido por los números enteros para calcular esa función. El método asociado con factorial es

```
factorial
  self <= eq: 0 iftrue: [ <- 1 ];
  self <= lt: 0 iftrue: [ self <= error: 'factorial
invalido']
          iffalse: [<- self <= * (self <= - 1 <=
factorial)]
 END
```

El receptor es un número entero. La primera expresión prueba si el receptor es 0 y si lo es, regresa 1 como valor del mensaje. La segunda expresión prueba el signo de receptor, si éste es menor que cero un error se notifica (todo objeto responde al mensaje ERROR: con una notificación de que un error ha sido encontrado). Si el receptor es mayor que 0, entonces, el valor que regresa resulta de la multiplicación del receptor por el factorial del valor del receptor menos uno.

3.3.4 METODOS PRIMITIVOS.

No todos los mensajes se responden ejecutando un método como el que se ha descrito anteriormente, hay algunos que se responden por operaciones propias de la Máquina Virtual. Por ejemplo, operaciones de entrada/salida, operaciones aritméticas, etc.. Para invocar ha estas operaciones desde un método se escribe la expresión

PRIMITIVE #

En donde # es un número entero indicando cual método primitivo será invocado. Los métodos primitivos se encuentran descritos en la Máquina Virtual.

La utilidad de éstas expresiones es la reimplementación de clases primitivas en el sistema.

Para mayor información de estos métodos ver el capítulo 5, que trata de la Máquina Virtual.

3.4 SUBCLASES.

En la descripción hecha hasta aquí de la estructura de una clase, no se ha contemplado la intersección entre miembros de diferentes clases, o sea, cada objeto ha sido visto como una instancia de sólo una clase. Esta estructura se ilustra en la fig. 3.1. En ella los pequeños círculos representan instancias y las cajas clases. Si un círculo esta dentro de una caja, entonces se trata de una instancia de la clase representada por la caja.

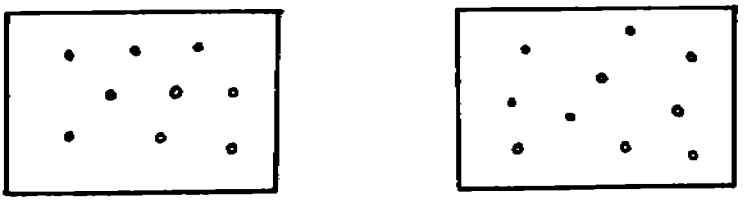


FIG. 3.1

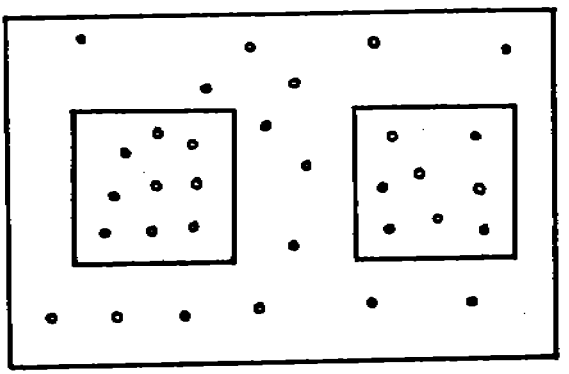


FIG. 3.2

A veces se presenta el caso que dos objetos son similares y sólo difieren en ciertos aspectos. Por ejemplo, un número real y un entero son similares en su capacidad de responder a mensajes aritméticos, pero son diferentes en su forma de representar los valores numéricos. Es deseable por tanto que estas dos clases compartan un mismo conjunto de mensajes tales como +, -, *, / y además respondan a mensajes propios de su representación, por ejemplo, los números reales responden al mensaje TRUNC que lo trunca y regresa como resultado un número de punto fijo; en cambio, los números enteros pueden responder a mensajes tal como FACTORIAL que calcula el factorial de un número.

Para esto se creó el concepto de subclase, cuyo propósito es el de que una o varias clases compartan las similitudes de otra clase llamada superclase. Esta estructura se muestra en la fig. 3.2, y es estrictamente jerárquica, es decir, una clase puede tener tan sólo una superclase, pero ésta puede tener varias subclases.

Una subclase especifica que sus instancias serán iguales a las instancias de otra clase, llamada superclase, excepto por las diferencias que sean explícitamente establecidas. Todas las clases son subclase de una clase ya existente. Una clase del sistema llamada OBJETO describe las similitudes de todos los objetos en el sistema; así, todas las clases al menos serán una subclase de OBJETO.

Cada clase tiene sólo una superclase y muchas clases pueden tener la misma superclase; de esta manera las clases forman una estructura de árbol, en donde la raíz de éste es la clase OBJETO.

En base a la estructura de una clase, una subclase debe cumplir las siguientes reglas:

1. Debe tener un nombre propio de clase.

2. Se pueden declarar nuevas variables de instancia.

3. Se pueden declarar nuevas variables de clase.

4. Se pueden adicionar o sobreponer nuevos métodos.

Esto es, si se declaran nuevas variables de instancia, las instancias de la subclase tendrán más campos que las instancias de la superclase. Si se declaran nuevas variables de clase, serán accesibles a las instancias de la subclase, pero no por las de su superclase. Todas éstas nuevas declaraciones, deben de ser diferentes de cualquiera de las declaradas en la superclase.

Si una subclase adiciona un método cuyo patrón de mensaje tiene el mismo selector que un método en la superclase, sus instancias responden a mensajes con ese selector ejecutando el nuevo método. Esto se llama sobreponer un método. Si la subclase adiciona un método con un selector no identificado en

los métodos de la superclase, las instancias de la subclase responderán a mensajes no entendidos por las instancias de la superclase.

El ejemplo 3.2 muestra una subclase de la clase HistoriaFin (ejemplo 3.1) llamada DedHisFin. En ella, se incluye una nueva entrada en el encabezado de la clase en donde se especifica la superclase. Con lo cual, las instancias de la subclases comparten las funciones de HistoriaFin de manejar información acerca de los ingresos y egresos. Además, agrega la función de recordar los gastos que son deducibles de impuesto. Para lo cual, adiciona una variable de instancia y cuatro métodos. Uno de esos métodos (BalanceIni:) se sobrepone al método de la superclase.

Una instancia de la clase DedHisFin es capaz de responder a los mensajes contenidos en la propia clase DedHisFin, y en las clases HistoriaFin y Objeto. De esta manera, esta instancia tiene cuatro variables de instancia, de las cuales tres hereda de la superclase HistoriaFin, y una se especifica en la misma clase. Por lo que se refiere a la clase Objeto, ésta no tiene variables de instancia. La fig. 3.3 muestra la representación de esas tres clases.

Las instancias de DedHisFin puede ser utilizadas para recordar la historia de gente que paga impuestos (negocios,caseros). Las instancias de HistoriaFin pueden ser usadas para recordar la historia de gente que no paga impuestos (organizaciones religiosas por ejemplo).

En adición a los mensajes y métodos heredados de la clase HistoriaFin, una instancia de DedHisFin puede responder a los mensajes que indican que todo o parte de los gastos son deducibles de impuesto.

Los nuevos mensajes disponibles son: GastoDed:Por:, el cual se usa si la cantidad total es deducible; Gasto:Por:Deduccion:, si sólo una parte de lo gastado es deducible; y TotalDed, para conocer la cantidad total de deducible.

3.4.1 DETERMINACION DEL METODO.

Cuando un mensaje se envía, el selector de éste se busca en los métodos de la clase del receptor hasta encontrar uno que sea igual. Si ningún selector se encuentra, entonces la busqueda sigue en los métodos de la superclase. La busqueda continuará en la cadena de superclases hasta que el método se encuentre. La busqueda termina en la clase objeto y si aqui no es hayado el selector, se reporta un error al usuario ("Mensaje no entendido").

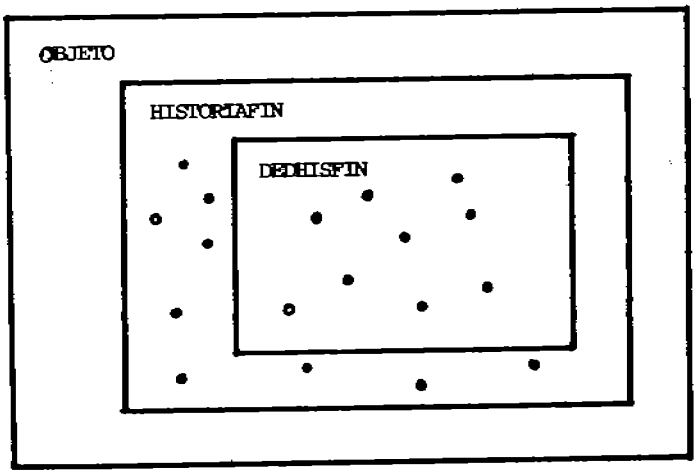


FIG. 3.3

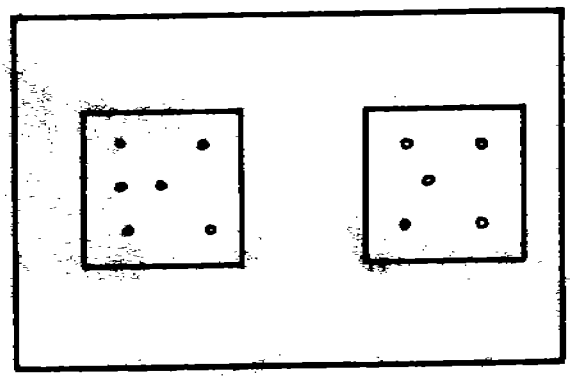


FIG. 3.4

EJEMPLO 3.2

```
CLASS_MOD
    NAME          DedHisFin
    SUPERCLASS    HistoriaFin
    VAR_INS       egresosded

TO_OBJECT

    gastoded: cantidad.fix por: concepto.str =>
      self <= gastar: cantidad por: concepto;
      egresosded := egresosded <= + cantidad
    END

    gastar: cantidad.fix por: concepto deduccion: cantded.fix =>
      self <= gastar: cantidad por: concepto;
      egresosded := egresosded <= + cantded
    END

    totalded => <- egresosded END

    balanceini: cantidad.fix =>
        super <= balanceini: cantidad;
        egresoded := 0
    END
END_OBJ
END_CLASS
```

Si se supone que se envía el mensaje Gasto:Por: a una instancia de DedHisFin. Este método se encuentra definido en la superclase HistoriaFin y se repite una vez más aquí

```
Gastar: cantidad por: razon =>
   egresos <= en: razon
              pon: (self<=TotGasPor:razon<=+cantidad);
   dinero := dinero <= - cantidad
end
```

Los valores de las variables de instancia egresos y dinero son accesadas en el receptor del mensaje, esto es, en la instancia de DedHisFin. La Pseudo-variable SELF se referencia también en éste método y representa la instancia de DedHisFin que fué el receptor del mensaje.

3.4.2 MENSAJES A SELF.

Cuando un método contiene un mensaje cuyo receptor es self, la busqueda del método para ese mensaje empieza en la clase de la instancia, sin considerar cual clase tiene el método que contiene self.

Los mensajes a self serán explicados usando dos clases de ejemplo llamadas Una y Dos. Dos es una subclase de Una y Una es una subclase de Objeto. Ambas clases incluyen métodos para los mensajes prueba y resulta, que regresa el resultado de la expresión

SELF <= PRUEBA

```
CLASS_MOD
   NAME Una
   SUPERCLASE Objeto
TO_OBJECT
   prueba => <- 1 END

   resulta => <- self <= prueba END
END_OBJ
END_CLASS

CLASS_MOD
   NAME Dos
   SUPERCLASS Una
TO_OBJECT
   prueba => <- 2 END
END_OBJ
END_CLASS
```

Una instancia de cada clase será utilizada para demostrar la determinación del método en los mensajes a self. Así, ejem1 es una instancia de la clase Una y ejem2 es una instancia de clase Dos.

```
ejem1 := .una <= crea ;
ejem2 := .dos <= crea
```

La siguiente tabla muestra el resultado de evaluar varias expresiones.

expresion	resultado
ejem1 <= prueba	1
ejem1 <= resulta	1
ejem2 <= prueba	2
ejem2 <= resulta	2

Los dos mensajes resulta invocan al mismo método, el cual se encuentra en la clase Una. Como podrá notarse, producen resultados diferentes debido al mensaje a self contenido en ese método. Cuando el mensaje resulta se envía a ejem2, la busqueda del método empieza en Dos. Como el método no es encontrado en Dos, la busqueda continua en la superclase, Una. Un método para resulta es encontrado en Una, el cual consiste de la expresión, <-self<=prueba. La pseudo-variable self se refiere al receptor, ejem2. La busqueda para la respuesta a prueba, por lo tanto, empieza en la clase Dos. Un método para prueba es encontrado en Dos, el cual regresa el valor 2.

3.4.3 MENSAJES A SUPER.

Existe otra pseudo-variable llamada super para utilizarse en las expresiones del método. Super se refiere al receptor del mensaje al igual que self. Sin embargo, cuando un mensaje se envía a super, la busqueda del método no comienza en la clase del receptor; en su lugar, la busqueda comienza en la superclase de la clase conteniendo el método. El uso de super permite a un método accesar métodos definidos en la superclase aún cuando el método haya sido sobrepuesto. Otro uso de super diferente a receptor (por ejemplo, un argumento), tiene un efecto igual que self.

Esta pseudo-variable no se contempla en el lenguaje TM.

3.4.4 SUPERCLASES ABSTRACTAS.

Las superclases abstractas son creadas cuando dos clases comparten una parte de su descripción y ninguna es propiamente una subclase de la otra. Entonces, una superclase se crea para las dos clases y que además contiene sus similitudes. Este tipo de superclases se llaman abstractas porque no pueden tener instancias. Su representación se muestra en la fig. 3.4.

La superclase abstracta sólo es un concepto derivado del de subclase, por lo cual, no existe una estructura sintáctica especial para definirlas en los lenguajes TM y SS, por lo que se deja el cuidado de su implementación al usuario.

Como un ejemplo, se consideran dos clases cuyas instancias representan diccionarios. Se entiende por diccionario un par de listas paralelas en donde se asocia a un nombre un valor. Una clase, llamada DiccMin, minimiza el espacio necesario para almacenar el contenido: el inconveniente de esta estructura es que el acceso a la información es lento puesto que se realiza una busqueda lineal; la otra clase, llamada DiccRap, almacena la información esparcidamente, pero el acceso a la información es rápido ya que se utiliza una técnica de hash para localizar los nombres.

Con excepción de como los nombres son localizados en DiccMin y DiccRap, ambas clases son muy similares: responden al mismo conjunto de mensajes y las dos manejan la misma estructura del diccionario. Estas similitudes son representadas en una superclase abstracta llamada Diccionario y el ejemplo 3.3 muestra su implementación.

Los tres mensajes a self utilizados en sus métodos no son implementados en esta clase: longitud, indiceDe: y creaIndiceDe:. Esta es la razón por la que es llamada clase abstracta. Si una instancia fuera creada, no sería capaz de responder a todos los mensajes. Las subclases DiccMin y DiccRap serán creadas para contestar esos mensajes ya que dependen de la forma de acceso al diccionario.

La implementación de la clase DiccMin se presenta en el ejemplo 3.4. Este ejemplo, no adiciona variables de instancia y además tiene un método para crecer en caso de ser necesario. Puesto que los nombres se almacenan contiguamente, la longitud del DiccMin es la longitud del arreglo de nombres. El índice de un nombre, en particular, se determina por una busqueda lineal en el arreglo de nombres. Si no se encuentra, se regresa el índice 0 y un mensaje de error es notificado. Siempre que una nueva asociación se agrega en el diccionario, el método para creaIndiceDe: se usa para encontrar el índice apropiado. En este caso, asume que los tamaños de los nombres y valores son exactamente los tamaños necesarios para almacenar los elementos actuales. Esto significa que no hay espacio disponible para accesar nuevos elementos. El mensaje crece crea dos nuevos arreglos que son copias de los anteriores, con un elemento más al

final. En el método para creaIndiceDe:, primero los tamaños de nombres y valores son incrementados y luego el nuevo nombre es puesto al final del arreglo. El método que llama a creaIndiceDe: tiene la responsabilidad de almacenar el valor.

La implementación de la clase DiccRap se presenta en el ejemplo 3.5. En general, todos los objetos responden al mensaje hash y se regresa como valor del mensaje un número. Los números responden al mensaje mod: regresando su valor en el módulo de su argumento.

EJEMPLO 3.3

```
CLASS_MOD
   NAME          diccionario
   VAR_INS       nombres.array
                 valores.array

TO_OBJECT

    en: nombre.str =>
      dcl indice.fix ;
      indice := self <= indiceDe: nombre;
      (indice <= eq: 0) <=
          iftrue: [ self <= error: 'Nombre no encontrado' ]
            iffalse: [ <- valores <= en: indice ]
    END

    en: nombre.str pon: valor =>
      dcl indice.fix ;
      indice := self <= indiceDe: nombre;
      (indice <= eq: 0) <=
          iftrue: [ indice := self <= creaIndiceDe: nombre ];
      <- valores <= en: indice pon: valor
    END

    existe: nombre.str =>
      <- self <= indiceDe: nombre <= ne: 0
    END

    esvacio => <- self <= longitud <= eq: 0 END

    inicializa =>
        nombres := .array <= crea: 0;
        valores := .array <= crea: 0
    END

END_OBJ
END_CLASS
```

EJEMPLO 3.4

```
CLASS_MOD
    NAME         DiccMin
    SUPERCLASS   Diccionario

TO_OBJECT

    longitud => <- nombres <= longitud END

    indiceDe: nombre =>
      1 <= to: (nombres <= longitud) do:
          [ :indice | nombres <= en: indice <= eq: nombre <=
              iftrue: [ <- indice]];
      <- 0
    END

    creaIndiceDe: nombre =>
      self <= crece;
      nombres <= en:(nombres<=longitud) pon: nombre;
      <- nombres <= longitud
    END

    crece =>
      DCL nombresAnt valoresAnt;
      nombresAnt := nombres;
      valoresAnt := valores;
      nombres := .Array <= crea: (nombres<=longitud<=+1);
      valores := .Array <= crea: (valores<=longitud<=+1);
      nombres <= remplazaDe: 1 a: (nombresAnt<=longitud) con:
nombresAnt;
      valores <= remplazaDe: 1 a: (valoresAnt<=longitud) con:
valoresAnt
    END

END_OBJ
END_CLASS
```

EJEMPLO 3.5

```
CLASS_MOD
    NAME          DiccRap
    SUPERCLASS    Diccionario
TO_OBJECT

    longitud =>
      DCL longitud;
      longitud := 0;
      nombres <= do: [ :nombre | nombre <= noNil <=
          iftrue: [ longitud := longitud <= + 1 ] ];
      <- longitud
    END

    inicializa =>
      nombres := .array <= crea: 4;
      valores := .array <= crea: 4
    END

    indiceDe: nombre =>
      DCL indice;
      indice := nombre <= hash <= mod: (nombres<=longitud) <=
+ 1;
      [ nombres <= en: indice <= ne: nombre ] <=
          whiletrue: [ nombres<=en: indice <= esNil <=
              iftrue: [ <-0]
              iffalse:
[indice:=indice<=mod:(nombres<=longitud)<=+1]];
      <- indice
    END

    creaIndiceDe: nombre =>
      DCL indice;
      nombres <= longitud <= - (self <= longitud) <=
      le: (nombres <= longitud <= / 4) <=
          iftrue: [self<=crece];
      indice := nombre<=hash <=mod: (nombres<=longitud)
<=+1;
      [ nombres <= en: indice <= noNil] <=
          whiletrue: [indice:=indice<=mod:(nombres<=longitud)
<= + 1];
      nombres <= en: indice pon: nombre;
      <- indice
    END
```

```
    crece =>
      DCL nombresAnt valoresAnt;
      nombresAnt := nombres;
      valoresAnt := valores;
      nombres := .array<=crea: (nombres<=longitud <=2);     2);
      valores := .array<=crea: (valores<=longitud <=2);
      1 <= to: (nombresAnt<=longitud) do:
          [:indice |
              nombresAnt <= en: indice <= noNil <=
                  iftrue: [self<=en: (nombresAnt<=en: indice)
                                  pon: (valoresAnt<=en: indice)]]

    END
END_OBJ
END_CLASS
```

La clase DiccRap sobrepone la implentación del método inicializa para crear arreglos con algún espacio para realizar el hash. La longitud de un diccionario DiccRap, no es simplemente la longitud del arreglo, ya que estos tienen siempre elementos vacios. La longitud se determina examinando todos los elementos del arreglo y contando el número de elementos que nos están vacios.

La implementación de creaIndiceDe: sigue básicamente la idea usada por DiccMin, excepto cuando el tamaño de un arreglo se cambia (duplicado en éste caso por el método crece), cada elemento es explícitamente copiado de los arreglos anteriores a los nuevos de tal manera que, a cada uno de los elementos le es aplicado nuevamente el hash. La longitud no siempre tiene que cambiar, como sucede en DiccMin, sino que, se cambia sólo cuando el número de localidades vacias en nombres cae por debajo de un mínimo. El mínimo es igual a 25% de los elementos.

```
nombres<=longitud
        <=-(self<=longitud)
        <= le:(nombres<=longitud<=/4)
```

3.5 METACLASES.

En el sistema las clases también se representan por objetos y, por lo tanto son instancias de una clase. Una clase cuyas instancias son en si mismas una clase, se le llama metaclases. La siguiente expresión

```
.HistoriaFin <= crea
```

muestra como un mensaje se envía al objeto que representa la clase HistoriaFin, esto hace pensar que es necesaria una metaclase que administre esos objetos y responda a ese mensaje. Existe una metaclase que responde a mensajes comunes a todas las clases, en particular al mensaje crea, para la creación de una nueva instancia de la clase, a la cual se le envió el mensaje. El resultado de este mensaje es una instancia de esa clase, con las variables de instancia no inicializadas. Esto obliga al usuario a mandar un mensaje a ese objeto para su inicialización. Por ejemplo

```
.HistoriaFin <= crea <= balanceIni: 3000
```

el primer mensaje crea una instancia no inicializada de la clase HistoriaFin y el segundo mensaje va dirigido al objeto creado y

sirve para inicializar sus variables de instancia. Esto implica que el programador siempre que mande el mensaje crea tenga que mandar otro mensaje para inicializarlo. Ejemplos de este tipo de inicialización, fueron mostrados en la clase HistoriaFin del ejemplo 3.1. Por esta razón se pensó en que cada clase tuviera asociada una metaclase, que sirviera para inicializar sus variables de instancia y así el usuario con un solo mensaje creara e inicializara una instancia. La metaclase se especifica opcionalmente en la clase y puede tener varios métodos para inicializar sus objetos. Así, sea un ejemplo con la clase rectángulo. Podría inicializar sus instancias dados dos puntos (el inferior izquierdo y el superior derecho) o bien dado un punto (el inferior izquierdo), su ancho y su altura. La metaclase tendría un método de inicialización para cada uno de esos mensajes.

Puesto que hay una correspondencia uno a uno entre una clase y su metaclase, su descripción se presenta junta. Sintácticamente los métodos que responden a mensajes enviados a la clase misma, o sea, a la metaclase, estan limitados por las palabras TO_ITSELF...END_IT. La sintáxis de un método es la misma. Un ejemplo de metaclases se muestra en la nueva implementación de la clase HistoriaFin dado en el ejemplo 3.6.

Se han hecho dos cambios en esta implementación:

1. Dos mensajes a la metaclase han sido agregados: BalanceIni: y crea.

2. El mensaje a las instancia BalanceIni a sido remplazado por iniBalance.

Este ejemplo ilustra como las metaclases crean instancias inicializadas. Los métodos para creación de instancias BalanceIni: y crea no tienen acceso directo a las variables de instancia de la nueva instancia (dinero, ingresos, egresos). Esto es, porque esos métodos no son parte de la clase de la nueva instancia. Por lo tanto, los métodos para la creación de instancias primero la crean e inicializan y después envían el mensaje de inicialización iniBalance:, a la nueva instancia. El método para éste mensaje se encuentra en la parte de los métodos de las instancias, estos si pueden asignar valores apropiados a las variables de instancia.

El método anterior para inicialización de instancias fue remplazado, porque la manera propia para crear una instancia de HistoriaFin es

```
.HistoriaFin <= balanceIni: 3000
```

y no

```
.HistoriaFin <= crea <= balanceIni: 3000
```

EJEMPLO 3.6

```
CLASS_MOD
   NAME       HistoriaFin
   VAR_INS    dinero.fix
              ingresos.diccionario
              egresos.diccionario

TO_ITSELF

    balanceIni: cantidad =>
      <- super <= crea <= iniBalance: cantidad
    END

    crea =>
      <- super <= crea <= iniBalance: cantidad
    END

END_IT
TO_OBJECT

    recibir: cantidad.fix de: fuente.str =>
      ingresos <= en: fuente
          pon: (self <= recibidoTot: fuente <= + cantidad);
      dinero := dinero <= + cantidad
    END

    gastar: cantidad.fix por: concepto.str =>
      egresos <= en: concepto
          pon: (self <= gastoTot: concepto <= + cantidad);
      dinero := dinero <= - cantidad
    END

    dinero => <- dinero end

    recibidoTot: fuente.str =>
      ingresos <= existe: fuente <=
            iftrue: [ <- ingresos <= en: fuente ]
              iffalse: [ <- 0 ]
    END

    gastoTot: concepto.str =>
      egresos <= existe: concepto <=
          iftrue: [ <- egresos <= en: concepto ]
          iffalse: [ <- 0 ]
    END
```

```
    iniBalance: cantidad.fix =>
        dinero := cantidad;
        ingresos := .diccionario <= crea;
      egresos := .diccionario <= crea
    END
END_OBJ
END_CLASS
```

3.5.1 HERENCIA ENTRE METACLASES.

Los mecanismos para herencia entre metaclases son exactamente los mismos que para herencia entre instancias de clases. La jerarquía de subclases de metaclase es paralela a la jerarquía de sus instancias. Sólo hay que aclarar los siguientes puntos:

1. La metaclase de la clase Objeto describe las similitudes de todas las metaclases.

2. Cuando una metaclase se refiere a self, se refiere a la clase que recibió el mensaje.

3. Cuando una metaclase se refiere a super se refiere a la metaclase de la superclase, representada por la clase que recibe el mensaje.

3.5.2 INICIALIZACION DE VARIABLES DE CLASE.

Otro uso de los mensajes a las clases es para la inicialización de variables de clase. En la definición de la clase se declaran las variables de clase no inicializadas. Por lo tanto, debe de existir un método para inicializarlas. Una manera apropiada de hacerlo es enviar un mensaje a la clase para que lo haga.

Las variables de clase son accesibles tanto a la metaclase como a la clase. En el ejemplo 3.7 se describe la implementación final de la clase DedHisFin, presentada en el ejemplo 3.2. Esta vez con sus mensajes de inicialización y una variable de clase que necesita ser inicializada. La variable de clase declarada en este ejemplo es deduccionMin.

Esta versión de DedHisFin, tiene cinco métodos para las instancias, uno de los cuales es esValido. La respuesta a este mensaje es true ó false, dependiendo si se han acumulado varias deducciones como para ser tomadas en cuenta en una declaración de impuestos. Por poner un ejemplo, si se considera que la mínima cantidad acumulada sea 20000, esta constante es referida por la variable de clase deduccionMin. Esa asignación se hace con el mensaje inicializa antes de que cualquier instancia sea creada.

EJEMPLO 3.7

```
CLASS_MOD
    NAME                DedHisFin
    SUPERCLASS          HistoriaFin
    VAR_INS             egresosded
    DCL                 deduccionMin

TO_ITSELF

    balanceIni: cantidad => DCL hisNueva;
      hisNueva := super <= balanceIni: cantidad;
      hisNueva <= iniDeduccion;
      <- hisNueva
    END

    crea => DCL hisNueva;
      hisNueva := super <= balanceIni: 0;
      hisNueva <= iniDeduccion;
      <- hisNueva
    END

    inicializa => DeduccionMin := 20000 END

END_IT
TO_OBJECT

    gastoded: cantidad.fix por: concepto.str =>
      self <= gastar: cantidad por: concepto;
      egresosded := egresosded <= + cantidad
    END

    gastar: cantidad.fix por: concepto deduccion: cantded.fix =>
      self <= gastar: cantidad por: concepto;
      egresosded := egresosded <= + cantded
    END

    esValida =>
      <- egresosDed <= ge: deduccionMin
    END

    totalded => <- egresosded end

    iniDeduccion => egresosDed := 0 END

END_OBJ
END_CLASS
```

Finalmente, una vez que se han definido todos los elementos del lenguaje SS, el siguiente paso será el diseño e implementación del compilador, que traduce los elementos que forman una clase para que sean incorporados al sistema.

3.6 REFERENCIAS.

[1].- M. Gerzso; Report on the Language TM: its design and Definition; México, IIMAS-UNAM (1983).

[2].- A. Goldberd, D. Robson; Smalltalk-80 The Language and its Implementation; E.E.U.U.. Addison-Wesley Pub. (1983), Cap. 3 pp 40-89.

CAPITULO 4

EL COMPILADOR

En este capítulo se describe el diseño y la implementación del compilador. Primero se presentan las bases teóricas para la implementación sin pretender entrar en detalle en ellas, ni demostrar que las técnicas usadas para la implementación son las mejores. Estas técnicas, fueron escogidas porque su efectividad en la implementación de otros compiladores ha sido comprobada. Después se presenta la implementación del programa, mencionando las estructuras de datos principales, así como los módulos que componen al compilador y aquellos con los cuales interactua.

El compilador es un programa que dado el código fuente de una clase, produce la representación interna de esa clase en el sistema, según la gramática presentada en los capítulos anteriores.

4.1 ESPECIFICACION DE LA GRAMATICA.

Para la especificación sintáctica del lenguaje se utilizó la notación llamada gramática libre de contexto, la cual se llama también BNF(Backus-Naur Form). Esta notación ofrece algunas ventajas [1]:

1. Una forma fácil y precisa de especificar un lenguaje.
2. Imparte una estructura al lenguaje que es muy útil para su translación a código objeto y para la detección de errores.
3. Esta notación sirve como entrada a la mayoría de los generadores de Tablas de Análisis Sintáctico.

Las gramáticas de este tipo comprenden cuatro elementos: terminales, no terminales, un símbolo inicial y un conjunto de producciones.

Los terminales son los símbolos que forman el alfabeto del lenguaje y los reconoce el Analizador Léxico. Los **no terminales** definen una categoría sintáctica. Un no terminal se selecciona como **símbolo inicial** y denota el lenguaje en el cual se esta interesado. Las **producciones** definen las formas en las cuales las categorias sintácticas pueden ser construidas en función de otras y de los terminales.

Esta gramática se especifica de tal manera que no presente ambiguedades y además que su estructura sea tal que facilite las acciones semánticas. En el apendice B se muestra la gramática utilizada para construir el compilador especificada en esta notación.

4.2 FASES DE COMPILACION.

La compilación se lleva a cabo en tres fases, las cuales se muestran en la fig. 4.1. El analizador léxico tiene la función de reconocer los símbolos terminales (alfabeto) del lenguaje. El analizador sintáctico (analizador), revisa que los símbolos terminales se encuentren dispuestos de tal manera que cumplan con la sintaxis definida. La semántica verifica que la sintaxis tenga sentido dentro del contexto del programa, además de generar la representación interna en el sistema del código fuente. La rutina de errores notifica al usuario de algún error ocurrido en alguna de las tres fases anteriores y será capaz de recuperarse de ese error para retomar la compilación. Si ocurre algún error no es generada la representación interna.

4.2.1 EL ANALIZADOR SINTACTICO.

El analizador que se utiliza en el compilador se llama analizador LR porque rastrea la cadena de entrada de izquierda a derecha y construye una derivación a la derecha en reversa [2]. Para obtener el analizador se utilizó un generador (apéndice B) tal que dada una gramática libre de contexto nos produce automáticamente un analizador para la gramática. Si la gramática tiene ambiguedades o alguna otra construcción que sea difícil hacer el análisis sintáctico en un rastreo de izquierda a derecha de la entrada, entonces el generador puede localizar esa construcción e informar al diseñador del compilador de su presencia.

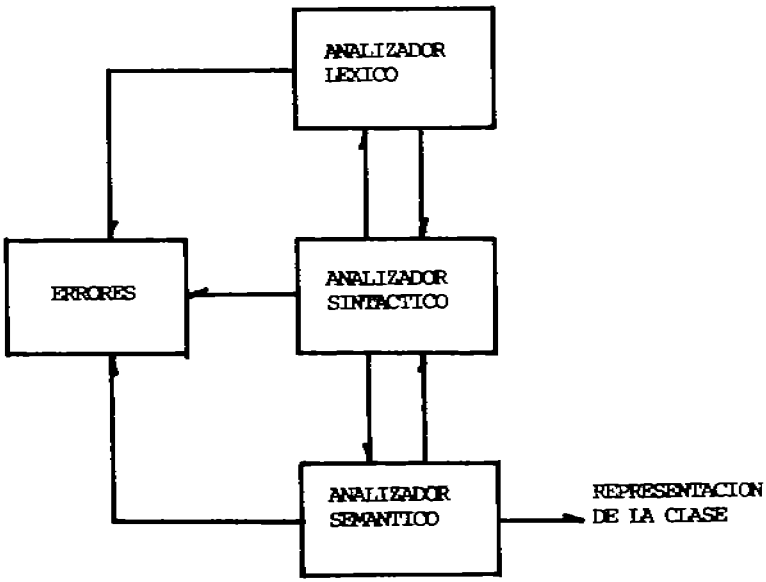
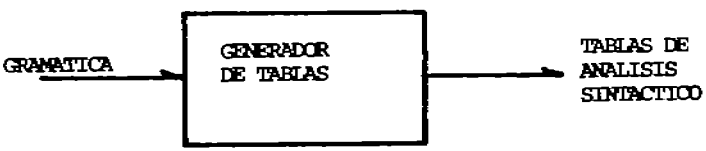
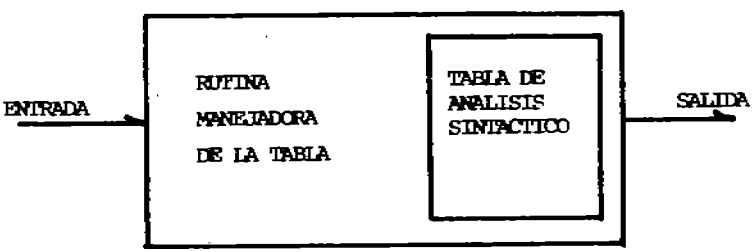


FIG. 4.1



(a) GENERANDO EL ANALIZADOR



(b) OPERACION DEL ANALIZADOR

FIG. 4.2

Lógicamente el analizador LR consiste de dos partes: una tabla de análisis sintáctico y un rutina que maneja esa tabla. Esta rutina es la misma para cualquier analizador LR (sólo con ajuste de algunos parámetros); y sólo la tabla cambia de un analizador a otro. El hecho anterior es útil durante el diseño del compilador, ya que la sintáxis cambia frecuentemente de acuerdo a las necesidades del diseño, produciendo diferentes tablas de análisis sintáctico, pero sólo una rutina se utiliza para manejarla. El esquema del analizador LR se muestra en la fig. 4.2.

La fig. 4.3 muestra el funcionamiento del analizador. Tiene una cadena de entrada, una pila y una tabla de análisis sintáctico. La cadena de entrada se lee de izquierda a derecha, un símbolo (token) a la vez. La pila contiene una cadena de la forma s0X1s1X2s2 ... Xmsm, donde sm esta en el tope. Cada Xi es un símbolo gramatical y cada si es un símbolo llamado estado. Cada símbolo de estado resume la información contenida debajo de la pila y se usa para guiar la decisión del shift-reduce. En esta implementación, los símbolos gramaticales no necesitan aparecer en la pila. Se incluyen sólo para ayudar a explicar el comportamiento del analizador. La tabla de análisis sintáctico consiste de dos partes, una tabla de ACCION y una de GOTO.

El analizador funciona de la siguiente manera: determina sm, el estado en el tope de la pila, y aj, el símbolo actual de entrada. Entonces consulta ACCION[sm,aj]. Esta entrada puede tener uno de los cuatro valores siguientes:

1. shift s

2. reduce A -> B

3. accepta

4. error

La función GOTO toma un estado y un símbolo gramatical como entrada y produce un estado. Es esencialmente la tabla de transición de un Autómata Finito Determinístico cuyos símbolos de entrada son los terminales y no terminales de la gramática.

Una configuración de un analizador LR es un par de componentes, el primero es el contenido de la pila y el segundo es la cadena que falta por ver

(s0 X1 s1 X2 s2 ... Xmsm , ai ai+1 ... an $)

El próximo movimiento del analizador se determina al leer ai, el símbolo actual de entrada, y sm, el estado en el tope de la pila, y entonces se consulta la Tabla de Acción con entrada ACCION[sm,ai]. La configuración resultante después de cada uno de los cuatro tipos de movimientos son los siguientes:

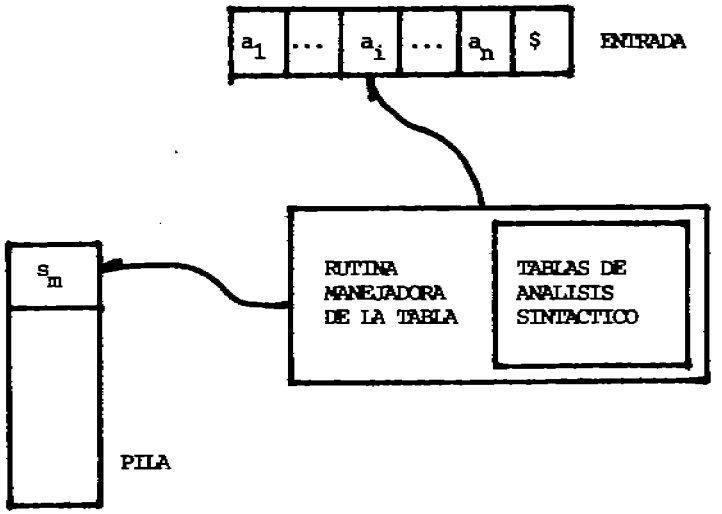

a_1 ... a_i ... a_n $
ENTRADA
s_m
RUTINA MANEJADORA DE LA TABLA
TABLAS DE ANALISIS SINTACTICO
PILA

FIG. 4.3

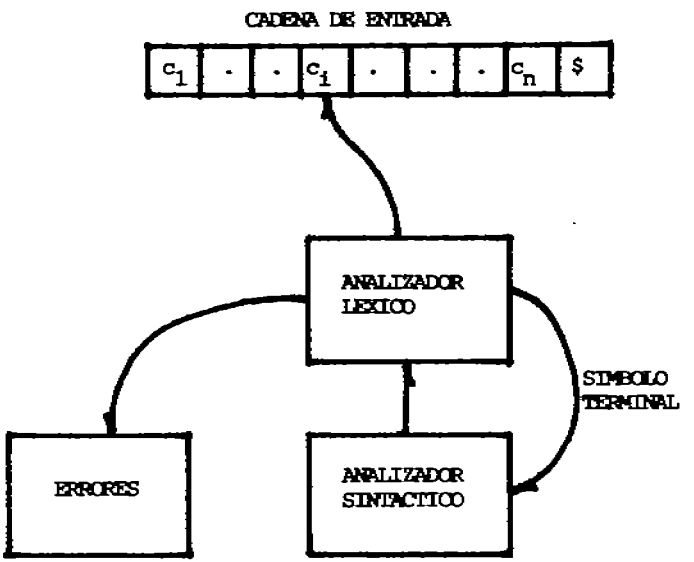

CADENA DE ENTRADA
c_1 . . c_i . . . c_n $
ANALIZADOR LEXICO
SIMBOLO TERMINAL
ERRORES
ANALIZADOR SINTACTICO

FIG. 4.4

1. Si ACCION[sm,aj]=shift s, entonces el analizador ejecuta un movimiento de introducir el nuevo estado en la pila y su nueva configuración es la siguiente:

   (s0 X1 s1 X2 s2 ... Xm sm ai s , ai+1 ... an $)

   dondeaii+1 es el actual símbolo de entrada.

2. Si ACCION[sm,ai] = reduce A -> B, entonces el analizador ejecuta un movimiento de reducción, entrando en la configuración

   (s0 X1 s1 X2 s2 ... Xm-r sm-r A s , ai ai+1 ... an $)

   donde s=GOTO[sm-r,A] y r es la longitud de B, el lado derecho de la producción. Aqui el analizador primero retira 2r símbolos de la pila (r símbolos de entrada y r símbolos gramaticales), quedando el estado sm-r en el tope de la pila. Entonces introduce tanto la A, el lado izquiedo de la producción, como s a la pila. El símbolo actual de entrada no es cambiado en un movimiento de reducción. La secuencia de símbolos gramaticales retirados de la pila siempre es lo mismo que B, el lado derecho de la producción reducida.

3. Si ACCION[sm,ai]=acepta, el análisis sintáctico se completa.

4. Si ACCION[sm,ai]=error, el analizador ha descubierto un error y llama a la rutina de manejo de errores.

   Inicialmente el analizador esta en la configuración

   (s0 , a1 a2 ... an)

   donde s0 es el estado inicial designado y a1a2..an es la cadena de entrada. Entonces el analizador ejecuta movimientos hasta una acción de acepta o error se encuentra.

4.2.2 ANALIZADOR LEXICO.

La función principal del analizador léxico es la de reconocer al conjunto de símbolos términales definidos para el lenguaje. Otras funciones secundarias son:

1. Salto de caracteres de edición, tales como TAB, CR, LF, FF, BLANCO. Utilizados como caracteres de "break", o sea, separadores de símbolos terminales.

2. Llevar un control de la línea y de la posición del texto, para el caso que ocurra un error, reportar la posición exacta de éste dentro del texto.

3. Salto de comentarios. Estos van entre los símbolos (* *).

El esquema de funcionamiento del analizador léxico se muestra en la fig. 4.4. El analizador léxico tiene como entrada la cadena de caracteres escrita por el programador y es invocado por el analizador sintáctico solicitandole el próximo símbolo terminal. Entonces, el analizador léxico lee uno o más caracteres de la cadena de entrada, hasta que un símbolo términal se reconoce ó un error ocurra (reconocer un símbolo que no este dentro del conjunto de símbolos terminales). Si el símbolo es válido su valor se regresa al analizador sintáctico junto con un conjunto de valores semánticos si el símbolo lo requiere.

Para la creación del analizador léxico es necesario primero especificar los símbolos terminales que son válidos para el lenguaje. Los símbolos terminales se dividen en tres grupos:

1. Símbolos de puntuación.

2. Símbolos compuestos.

3. Palabras reservadas.

A continuación describiremos los símbolos terminales. La notación que se utiliza es la siguiente: el índice al lado izquierdo indica el valor del símbolo al ser reconocido por el Analizador léxico y a continuación se presenta la descripción del símbolo.

Los símbolos de puntuacion reconocidos son 10:

1. (- Paréntesis izquierdo.
2.) - Paréntesis derecho.
3. , - Coma.
4. ; - Punto y coma.
5. [- Paréntesis cuadrado izquierdo.
6.] - Paréntesis cuadrado derecho.
7. := - Operador de asignación.
8. <= - Operador de mensaje.
9. => - Operador de método.
10. <- - Operador de retorno.

Los símbolos compuestos son símbolos primitivos y algunos son definidos en función de otros. Los símbolos primitivos son los siguientes conjuntos:

```
letras := ['A'..'Z']
dígitos:= ['0'..'9']
```

Los símbolos compuestos son:

11. IDENTIFICADOR
12. CARACTER-ESPECIAL
13. KEYWORD
14. CLASE-INST
15. OBJECT-DECL
16. LITERAL-PRIMITIVA

IDENTIFICADOR. Su primer caracter es una letra y le pueden seguir una o más letras o dígitos o el caracter '_' hasta un límite máximo de 12 caracteres. Si se pasa de este número el identificador es truncado a 12 caracters y se le notifica al usuario de tal ocurrencia.

CARACTER-ESPECIAL. Es un caracter que pertenece al conjunto de cualquiera de los siguientes 20 caracteres:

{ + - * / = < > ! " # $ % & ? @ \ ^ ` ~ : }

KEYWORD. Es un identificador seguido por el caracter ':'.

CLASE-INST. Es un punto seguido de un identificador

OBJECT-DECL. Es un identificador seguido por un punto y otro identificador.

LITERAL-PRIMITIVA. Es definida por las literales del sistema:

- número
- caracter
- cadena
- arreglo
- TRUE
- FALSE
- NIL

El analizador léxico regresa el valor de esas literales. En el caso de cadenas y arreglos genera el objeto que representa esas literales.

El tercer conjunto lo forman las palabras reservadas, las cuales son:

17. CLASS_MOD
18. END_CLASS
19. NAME
20. SUPERCLASS

21. VAR_INS
22. DCL
23. TO_ITSELF
24. END_IT
25. TO_OBJECT
26. END_OBJ
27. END

4.2.3 ANALIZADOR SEMANTICO.

El analizador semántico tiene como función de que el programa tenga sentido dentro del contexto en el cual se escribe, entre otras cosas, revisa que las variables utilizadas en las expresiones esten declaradas y si pertenecen a ese contexto, genera el código de las expresiones contenidas en los métodos, ordena los métodos de la clase actual que se este compilando y por último formar la representación interna de esa clase en el sistema. El código de un método, el método mismo y la representación interna de la clase son objetos que genera ésta función.

Las acciones semánticas son dirigidas por sintáxis, es decir, la función es llamada cada vez que el manejador de la tabla de análisis sintáctico reconoce la parte derecha de una producción y realiza una reducción. Dado el número de esta producción (número de regla), su función es realizar las acciones semánticas necesarias para la revisión de la estructura sintáctica hasta aquí construida.

El esquema de translación dirigido por sintaxis es útil, porque permite al diseñador del compilador expresar la generación de código en términos de la estructura sintáctica del lenguaje. La translación se realiza en conjunción con el analizador utilizado.

Por cada regla se realizan las siguientes acciones:

- Cargar valores semánticos.
- Realizar acción semántica.
- Guardar nuevos valores semánticos.

A continuación se describen estas acciones: carga valores semánticos, los cuales se obtienen de la pila de análisis sintáctico. Estos valores se requieren para la ejecución de la regla semántica actual. Representan el recuerdo de todas las acciones semánticas realizadas hasta aqui. Se requieren de tres conjuntos de valores semánticos como máximo. Esto quiere decir, que la parte derecha de una regla de producción puede contener como máximo tres estructuras sintácticas que hayan dejado cada

una sus valores semánticos. Este número se determinó empíricamente analizando las reglas del lenguaje. Esto no quiere decir que una regla de producción este limitada a tres elementos a la derecha, ya que no todos los elementos producen valores semánticos. Como un ejemplo, se muestra la pila de análisis sintáctico (fig.4.5(a)) al realizar la siguiente regla:

<iden-modulo> ::= NAME <class-inst> SUPERCLASS <class-inst>

Sólo la regla gramatical <class-inst> produce valores semánticos (nombre de la clase y superclase), estos serán cargados y se realizara la acción semántica correspondiente (revisar que el nombre de la clase no exista y el de la superclase si) por último se establecen los valores semánticos correspondientes a ésta regla (posición de la nueva clase y de la superclase en el diccionario global del sistema).

Al terminar de ejecutar esta regla termina semántica. Es el manejador de la tabla de análisis sintáctico el encargado de remplazar el lado derecho de la producción por el lado izquierdo de esta regla con sus correspondientes valores semánticos generados (fig.4.5(b)). Así, se remplazaron cuatro elementos gramaticales por uno solo que se podría tomar como el resumen de los otros.

A continuación se describen las principales acciones semánticas tomadas a nivel de estructura sintáctica para el lenguaje SS.

4.2.3.1 DECLARACION DE LA CLASE. -

NAME <class-inst> [SUPERCLASS <class-inst>]

Revisa que el nombre de la nueva clase que se está definiendo no exista y que la superclase que se le asocia exista. Si la superclase es omitida, le asocia la clase Objeto por default.

VAR_INS <decl-var> [<decl-var> ...]

Revisa que la lista de objetos declarados no se repitan entre sí, además de verificar que no hayan sido declarados en la superclase. Para esto forma una lista de variables de instancia (VI). Esta es contruida en orden inverso de como han sido declaradas. Esto es con el fin de poder ligar las nuevas VI declaradas en esta clase con las de su superclase (en caso de especificarse) para formar sólo una lista para esta clase. Para visualizar esto, un ejemplo con tres clases se presenta:

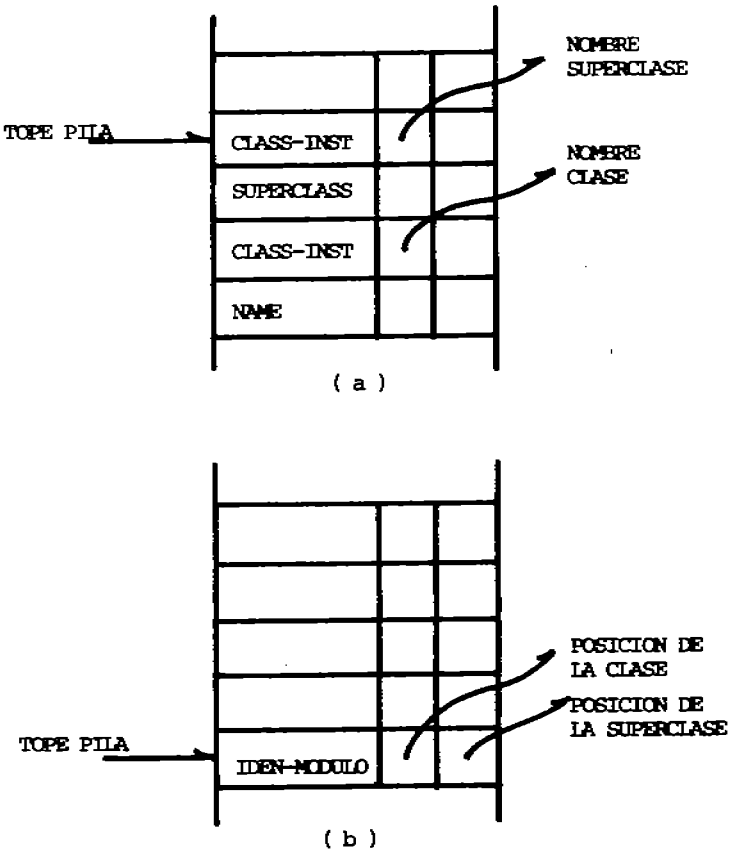

TOPE PILA
CLASS-INST
SUPERCLASS
CLASS-INST
NAME
NOMBRE SUPERCLASE
NOMBRE CLASE
(a)
TOPE PILA
IDEN-MODULO
POSICION DE LA CLASE
POSICION DE LA SUPERCLASE
(b)

FIG. 4.5

```
CLASS_MOD
   NAME Persona
   VAR_INS
      nombre.str
      edad.fix
      sexo.str
   -
END_CLASS
```

```
CLASS_MOD
   NAME Empleado
   SUPERCLASS Persona
   FIELDS
      n_empl.fix
      profesion.str
   -
END_CLASS
```

```
CLASS_MOD
  NAME Estudiante
  SUPERCLASS Persona
   FIELDS
      n_cuenta.fix
         grado.fix
      -
END_CLASS
```

Las clases son Persona, Empleado y Estudiante. Las últimas dos como subclases de persona. Una vez procesadas las tres clases, las listas se muestran en la fig.4.6 De esta manera, en una sola lista se tienen todos los campos, tanto los heredados como los declarados en la clase misma. La lista es invertida por el hecho que las nuevas VI declaradas en la clase se suman a los ya existentes para mantener la misma estructura del objeto. Esto sirve para implementar la herencia de clases, ya que el código generado en los métodos de la superclase esta en función de la posición de los campos en el objeto. Así la estructura de las instancias de Persona, Empleado y Persona quedan como se muestra en la fig 4.7.

4.2.3.2 METODOS. -

Un método esta contituido por las siguientes secciones

```
patrón del mensaje =>
   [Declaraciones temporales]
   lista de expresiones
END
```

Del patrón del mensaje se determinan dos cosas: el selector y los argumentos. Semántica determina la posición del selector en la tabla de selectores e instala los argumentos en la tabla de variables locales para resolver posteriores referencias. En caso de existir variables temporales son puestas en la misma tabla. Semántica establece la diferencia entre argumentos y variables para saber cuales variables de esa tabla pueden ser asignadas y cuales no en las expresiones del método y bloques.

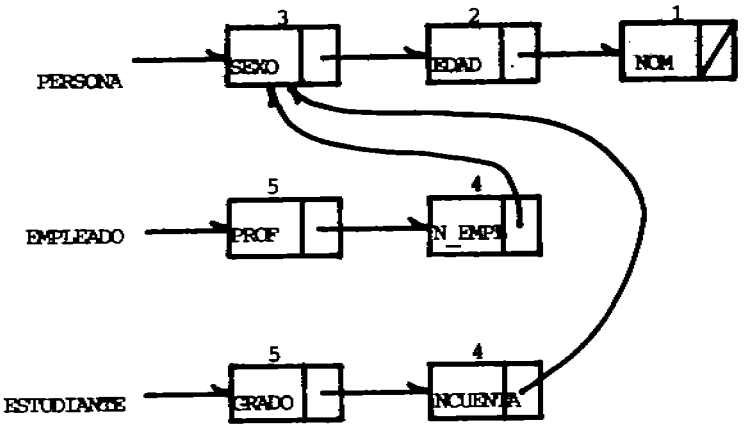

3
2
1
PERSONA
SEXO
EDAD
NOM
5
4
EMPLEADO
PROF
N_EMPL
5
4
ESTUDIANTE
GRADO
NCUENTA

FIG. 4.6

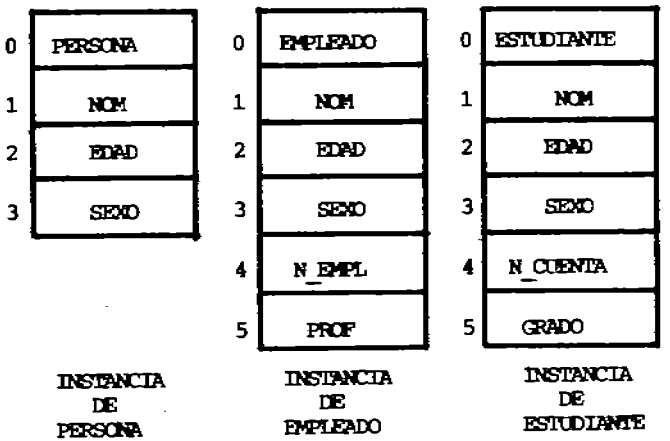

0
PERSONA
1
NOM
2
EDAD
3
SEXO
INSTANCIA DE PERSONA
0
EMPLEADO
1
NOM
2
EDAD
3
SEXO
4
N_EMPL
5
PROF
INSTANCIA DE EMPLEADO
0
ESTUDIANTE
1
NOM
2
EDAD
3
SEXO
4
N_CUENTA
5
GRADO
INSTANCIA DE ESTUDIANTE

FIG. 4.7

El cuerpo del método esta formado por una serie de expresiones separadas por punto y coma. Semántica realiza dos acciones por cada expresión: resolver las referencias a variables y literales usadas en esa expresión de acuerdo al contexto de ésta; y la de generar el código apropiado para la evaluación de la expresión. Estas acciones se presentan en las siguientes dos secciones. Como producto final por cada método compilado se produce el objeto método. Este esta formado por tres campos

- El selector. Con el cuál se invoca al método por medio de un mensaje. Es el identificador del método.

- El objeto código. Este es el resultado de la generación de código de las expresiones del método.

- Liga. Es un apuntador que sirve para insertar el objeto método en la lista final de métodos de la clase. A continuación se verá como se forma esa lista.

Entre las palabras reservadas TO_ITSELF...END_IT se definen los métodos que responden a mensajes enviados a la clase misma. Cada método forma un objeto método. El primer campo de éste objeto contiene el selector del método, que es un índice a la tabla de selectores. De acuerdo a éste índice, se forma una lista de métodos en forma ascendente. Esto con el fín de hacer la busqueda del método más eficiente a la hora de ejecución.

Entre las palabras reservadas TO_OBJECT...END_OBJ se definen los métodos que responden a mensajes enviados a las instancias de la clase. Se forma una lista de estos métodos con la misma lógica que la anterior.

Una vez que las acciones semánticas anteriores sé han realizado y no ha ocurrido algún error, semántica contruye la representación interna de la clase en el sistema. Esta representación la tiene un objeto llamado descriptor de clase. Este objeto tiene los siguientes campos:

1. Lista de los nombres de las variables de instancia que forman la estructura de las instancias de ésta clase en particular. Esa lista se encuentra formada en la tabla de campos.

2. Número de variables de instancia. Este valor se usa para la creación de nuevas instancias.

3. Variables de Clase. Es una referencia a las variables que pueden accesar los métodos de esta clase. Esas variables se localizan en un diccionario global del sistema.

4. Métodos a la clase misma. Es un apuntador a la lista de métodos construida para responder a mensajes dirigidos a la clase misma.

5. Métodos de las instancias. Es una lista de métodos construida para responder a mensajes dirigidos a las instancias de ésta clase.

Un esquema del Descriptor de Clase es mostrado en la fig.4.8

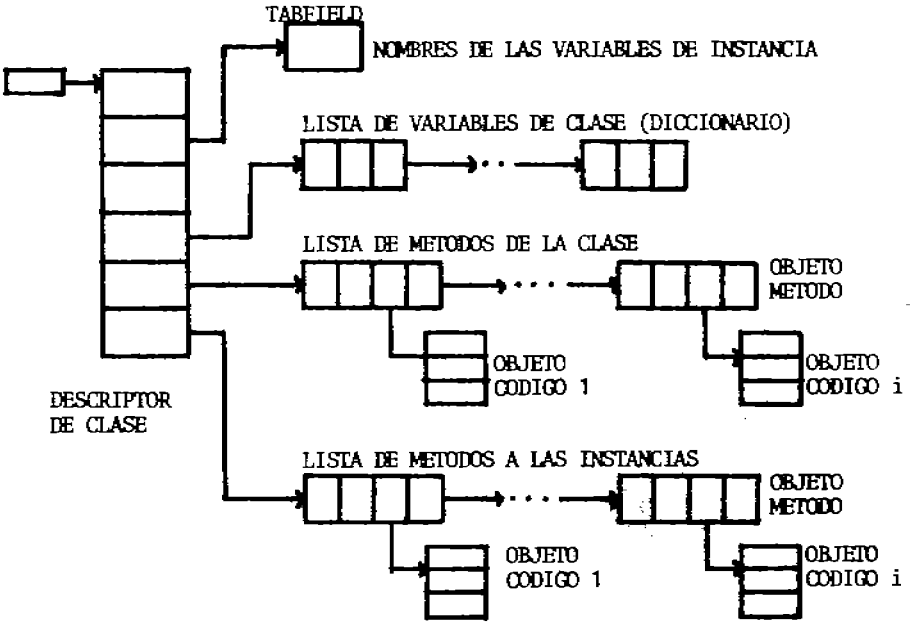


FIG. 4.8 REPRESENTACION INTERNA DE LA CLASE.

4.2.3.3 RESOLUCION DE REFERENCIAS. -

Los objetos involucrados en una expresión pueden ser directa o indirectamente referenciados por una instrucción. En los primeros caen aquellos objetos que por los mecanismos propios de la evaluación de la expresión su referencia esta contenida en la instrucción. En ésta categoria caen

- El receptor y argumentos del mensaje.
- Las variables temporales.
- Las variables de instancia del receptor.

El receptor, los argumentos y las variables temporales se localizan en la misma área de evaluación del método. A esta área se le llama Area Temporal porque desaparece una vez que el método termina. Semántica mantiene los nombres de esas variables en la tabla de variables locales para que posteriormente resuelva las referencias según su posición relativa respecto a ésta tabla.

Las variables de instancia del receptor se localizan en una lista referenciada por el descriptor de clase (campo 1). Semántica resuelve referencias a éstas variables por la posición relativa de éstas en la lista. A esa lista se llama Area del Receptor.

En la categoria de objetos que no pueden ser directamente referenciados por una instrucción caen las variables globales y las literales. Cuando semántica detecta una referencia a uno de estos objetos pone su referencia en un área propia del método y localizada en el mismo objeto código producido para el método. A esta área se le llama Area de Literales. Para una literal, su referencia es puesta directamente en esa área, sin embargo, una variable global es una referencia indirecta a través de un diccionario, el cual mantiene todos los objetos globales del sistema. Debido a lo anterior, el objeto código toma la estructura que se muestra en la fig 4.9. El primer campo es un índice a la primera instrucción ejecutable del método (inicio del área de código). Los siguientes campos corresponden al Area de Literales y por último el Area de Código producido para éste método.

A continuación se presenta un método y las áreas por medio de las cuales semántica resuelve las referencias.

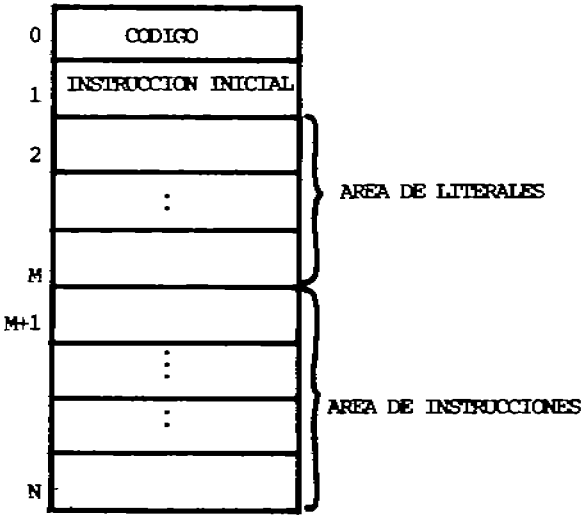

0
CODIGO
1
INSTRUCCION INICIAL
2
M
M+1
N
AREA DE LITERALES
AREA DE INSTRUCCIONES

FIG. 4.9

```
+ puntol =>
    DCL sumX sumY;
    sumX := x <= + ( puntol <= x );
    sumY := y <= + ( puntol <= y );
    <- .punto <= creaX: sumX y: sumY
END
```

Es un método correspondiente a la clase Punto. Esta clase define sus instancias por medio de dos variables de instancia "x" y "y". Calcula la suma de los campos del punto receptor a los del punto pasado como un argumento. Tiene dos variables temporales sumX y sumY.

El Area de Temporales (A.T.) se forma por las siguientes variables; a la derecha se muestra su posición relativa

```
receptor (self)     - 0
puntol (argumento)  - 1
sumX (local)        - 2
sumY (local)        - 3
```

El Area del Receptor (A.R.) se forma por sus variables de instancia, en este caso

```
x       - 1
y       - 2
```

El Area de Literales (A.L.) sólo contiene una referencia a una variable global (a la clase Punto)

```
.punto  - 1
```

Las variables que se referencian en el método son resueltas de la siguiente manera (de acuerdo a como se encuentran en el texto):

```
VARIABLE  AREA POSICION RELATIVA

sumX    - A.T.      (2)
x       - A.R.      (1)
puntol  - A.T.      (1)
sumY    - A.T.      (3)
y       - A.R.      (2)
.punto  - A.L.      (1)
```

Dentro de un método pueden existir uno o varios bloques, el conjunto de variables que se pueden referenciar en estos es el mismo que para un método. Adicionalmente, un bloque tiene acceso a los argumentos que le fueron declarados. Esos argumentos no son accesibles fuera de ese bloque. Para la resolución de referencias semántica coloca esos nombres de argumentos en el Area de Temporales mientras se esta compilando el bloque. Al terminar de compilarse éste, esas variables son retiradas de esa área. Los bloques anidados en otros tienen acceso tanto a las variables del método como a los argumentos de los bloques en los cuales estan anidados. Las literales usadas en los bloques comparten la misma Area de Literales del método.

4.2.3.4 GENERACION DE CODIGO. -

Una vez resueltas las referencias a objetos usados en las expresiones semántica produce las instrucciones para evaluarlas. Las instrucciones son para que el intérprete de la Máquina Virtual las ejecute (Capítulo 5). A continuación se describen las instrucciones generadas por semántica para que posteriormente se presenten algunos ejemplos de su uso para evaluar expresiones.

Las intrucciones generadas son: push, pop, store, send, check, return, blockcopy, jump.

La instrucción PUSH especifica un objeto que va ha ser introducido en la pila de ejecución del intérprete. Los objetos que pueden introducirse en la pila son:

- El receptor del mensaje que invocó al método (Self).
- Las variables de instancia del receptor (Area del Receptor).
- Los argumentos y variables locales (Area de Temporales).
- Literales y variables globales (Area de Literales).

La instrucción POP retira un objeto de la pila.

La instrucción STORE es la última instrucción generada de una expresión de asignación. Las instrucciones antes al store calculan el nuevo valor de la variable y lo dejan en el tope de la pila. La instrucción store indica que variable será cambiada. Las variables que pueden ser cambiadas son

- Las variables del instancia del receptor.
- Las variables locales.
- Las variables globales.

Si la variable que se asigna tiene asociado un administrador, ésta instrucción revisa que el objeto asignado sea del mismo tipo.

La instrucción SEND especifica el selector de un mensaje a ser enviado y cuantos argumentos tiene. El receptor y argumentos del mensaje se toman de la pila del intérprete, el receptor debajo de los argumentos. Una vez terminada de ejecutar ésta instrucción, el resultado del mensaje remplaza al receptor y los argumentos.

La instrucción CHECK revisa que los argumentos pasados en un mensaje correspondan a los declarados en el método que esta por ejecutarse. Si el tipo de objeto recibido es diferente al esperado, se reporta un error y se interrumpe la ejecución de las siguientes instrucciones. Sólo se genera cuando a un argumento de un método le es asociado un adminisrador.

Cuando una instrucción RETURN se ejecuta, termina la ejecución del método. Antes la instrucción remplaza el receptor y argumentos del método por el valor del método. Ese valor es encontrado en el tope de la pila.

La instrucción BLOCKCOPY es generada al detectarse el inicio de un bloque y sirve para crear otro contexto de ejecución. Esta instrucción se explica en detalle en el Capítulo 5 (5.3.5).

La instrucción JUMP es generada inmediatamente después de la instrucción BLOCKCOPY y sirve para saltarse las instrucciones correspondientes a un bloque. Es un salto incondicional.

A continuación se presentan algunos ejemplos que muestran la generación de estas instrucciones para evaluar expresiones contenidas en un método. Para simplificar la explicación se utilizan en la mayoría de los ejemplos números. Se utiliza una pila a la derecha para indicar como se evaluan esas expresiones.

En una expresión de mensaje como la siguiente,

```
2 <= + 3
```

El receptor del mensaje (2) se introduce primero en la pila, después el argumento del mensaje (3) y por último el mensaje "+" se envía. Las instrucciones generadas para ésta expresión son

```
PUSH 2          (2)
PUSH 3          (2 3)
SEND +,1        (5)
```

En una expresión de mensaje en cadena como

```
3 <= + 4 <= * 5 <= - 1
```

se genera

```
PUSH 3          (3)
PUSH 4          (3 4)
SEND +,1        (7)
PUSH 5          (7 5)
SEND *,1        (35)
PUSH 1          (35 1)
SEND -,1        (34)
```

Las expresiones entre paréntesis se evaluan primero, por ejemplo

```
3 <= + (4 <= * 5) <= - 1
```

genera

```
PUSH 3          (3)
PUSH 4          (3 4)
PUSH 5          (3 4 5)
SEND *,1        (3 20)
SEND +,1        (23)
PUSH 1          (23 1)
SEND -,1        (22)
```

En los mensajes en cascada el tope de la pila se duplica y luego un objeto se retira y duplicado de nuevo el tope de la pila. Por ejemplo

```
3 <= + 4, * 5, - 1
```

genera

```
PUSH 3          (3)
PUSH 4          (3 4)
SEND +,1        (7)
PUSH TOP_PIL    (7 7)
PUSH 5          (7 7 5)
SEND *,1        (7 35)
POP             (7)
PUSH TOP_PIL    (7 7)
PUSH 1          (7 7 1)
SEND -,1        (7 6)
POP             (7)
```

En una expresión de asignación el lado derecho se evalua primero y su valor se asigna a la variable del lado izquierdo. Por ejemplo

```
a := 3 <= + 4
```

genera

```
PUSH 3          (3)
PUSH 4          (3 4)
SEND +,1        (7)
STORE en a      (7)
```

El compilador genera también la instrucción POP cuando hay valores en la pila que ya no se utilizan ni referencian. La siguiente lista de expresiones muestra un ejemplo

```
5 <= * 4 ;
a := 3 ;
b := 4
```

genera

```
PUSH 5          (5)
PUSH 4          (5 4)
SEND *,1        (20)
POP             ()
PUSH 3          (3)
STORE en a      (3)
POP             ()
PUSH 4          (4)
STORE en b      (4)
```

En las expresiones de retorno, el tope de la pila se regresa como el valor del método. En el siguiente ejemplo se supone que la expresión de retorno esta en un método que fué invocado por un receptor (R) y algunos argumentos (A)

```
<- 3 <= + 4
```

genera

```
                        (R A)
PUSH 3                  (R A 3)
PUSH 4                  (R A 3 4)
SEND +,1                (R A 7)
RET                     (7)
```

La única diferencia entre expresiones contenidas en un método y las contenidas en un bloque es la última (si no es de retorno). Si la última expresión de un método no es de retorno, se regresa el receptor del mensaje como valor del método. En un bloque si la última expresión no es de retorno, se regresa el valor de la última expresión como el valor del bloque. Por ejemplo, si la expresión

```
3 <= * 4
```

es la última de un método, se genera

```
PUSH 3          (R A 3)
PUSH 4          (R A 3 4)
SEND *,1        (R A 12)
RET self        (self)
```

y si fuera la de un bloque

```
PUSH 3          (3)
PUSH 4          (3 4)
SEND *,1        (12)
RET             (12)
```

Ejemplos de las instrucciones BLOCKCOPY y JUMP son vistos en el capítulo 5.

4.3 LA IMPLEMENTACION.

En esta sección se describe la implementación del compilador. Primero se presenta las estructuras globales del sistema usadas, las cuales el compilador mantiene. Posteriormente se describen las interacciones del compilador con otros módulos, para finalizar con la implementación de éste.

La fig. 4.10 muestra los diversos módulos del sistema SS que tienen que ver de alguna manera con el compilador.

Los módulos DICC, TABCAM y TABSELE, tienen la función de mantener las estructuras globales del sistema, las cuales son actualizadas por el compilador cada vez que una nueva clase se instala en el sistema; su descripción se muestra en la sección 4.3.1.

El módulo ADMI es un componente de la Máquina Virtual cuya función es administrar el espacio de memoria; el compilador interactua con este módulo para crear y accesar diversos objetos durante la compilación de una clase.; su descripción se presenta en el capítulo 5 y en la sección 4.3.2 sólo se muestran las rutinas utilizadas por el compilador.

El módulo ANALEX (analizador léxico) tiene la función de proporsionar al compilador el siguiente símbolo terminal del texto que se esta procesando; su descripción e implementación toma lugar en la sección 4.3.3.

Finalmente, el módulo COMPI tiene la función de realizar el análisis sintáctico y semántico del texto que se este procesando; basicamente, contiene las tablas de sintaxis, la rutina para manejar la tabla y la rutina para realizar las acciones semánticas. Para obtener las tablas de sintáxis referirse al apéndice B. Su descripción e implementación se presentan en la sección 4.3.4.

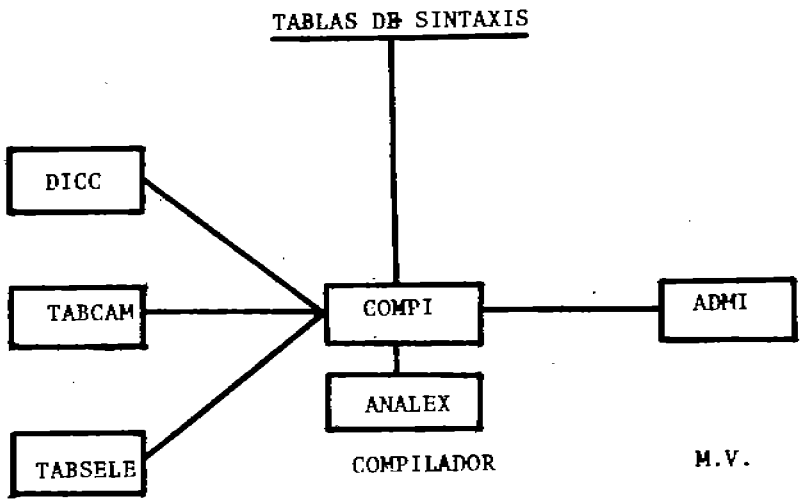

TABLAS DE SINTAXIS
DICC
TABCAM
TABSELE
COMPI
ANALEX
ADMI
TABLAS DEL SISTEMA
COMPILADOR
M.V.

FIG. 4.10

4.3.1 ESTRUCTURAS GLOBALES.

Las principales estructuras globales del sistema mantenidas por el compilador son las siguientes:

1. Diccionario Global del Sistema [DICC].

2. Tabla de Nombres de Variables de Instancia [TABCAM].

3. Tabla de Selectores [TABSEL].

El DICCIONARIO mantiene las descripciones de los diferentes objetos globales usados por el sistema.

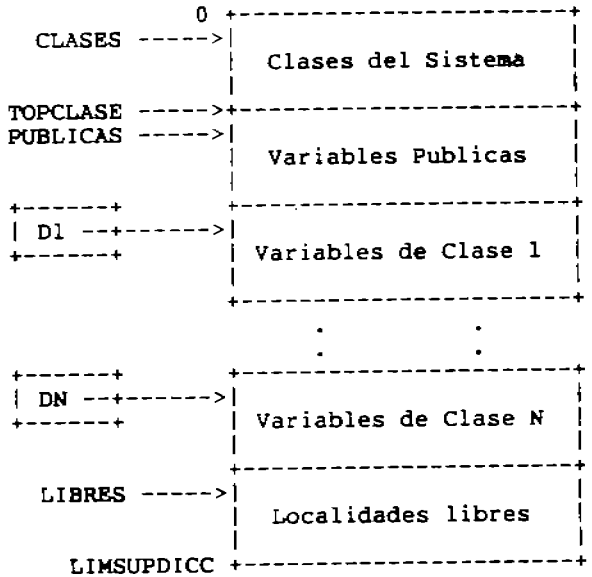
La TABLA DE CAMPOS contiene la descripción de los nombres de las variables de instancia declaradas en las diferentes clases compilada por el sistema.

La TABLA DE SELECTORES mantiene los selectores formados por los diferentes métodos de cada clase del sistema. El selector es unico en esta tabla.

A continuación se presenta la estructura y organización de cada una de estas, la terminología y símbolos se apega hasta donde sea posible a la utilizada en los programas. Con mayúsculas se indican las variables que se usan para esa estructura.

4.3.1.1 EL DICCIONARIO GLOBAL DEL SISTEMA. -

El DICCIONARIO es un arreglo de registros de 0..LIMSUPDIC. Se encuentra dividido lógicamente en un número variable de secciones, los registros de cada sección en particular se encuentran ligados entre sí, por tal razon no se encuentran contiguos uno del otro, pero para la explicación se presentarán como si lo estuvieran.



Clases del sistema. Aqui se describen las clases existentes en el sistema. Hay dos tipos de clases: primitivas y compiladas por el sistema. Las primeras se describen totalmente en esta sección, sin embargo las compiladas por el sistema se les asocia un descriptor (Descriptor de Clase). Las clases son accesadas por el sistema a través del apuntador CLASES inicializado en un principio. TOPCLASE indica el último registro que contiene una clase.

Variables públicas. Aquí se describen los objetos públicos del sistema. Son accesados por el sistema a través del apuntador PUBLICAS inicializado en un principio.

Variables de clase. Aquí se describen las variables de clase declaradas en una clase en particular. Son accesadas por el sistema a través de un campo del del Descriptor de Clase. Sólo las clases compiladas por el sistema tienen variables de clase.

Localidades libres. Es una lista de registros libres de donde el sistema va tomando los que necesite.

Un registro del Diccionario tiene los siguientes campos

NOMBRE	ADMI	SUPER	BANDERA	VALOR	SIGUE

	OBJETO	CLASE
NOMBRE	- Nombre Objeto	Nombre Clase
ADMI	- Administrador del Objeto.	NADIE
SUPER	- -	Superclase asociada a esta clase.
BANDERA	- Indica tipo valor.	Siempre 0
VALOR	- Valor del objeto.	Si es negativo indica que la clase es primitiva y valor indica número de módulo. Si es positivo es una clase compilada por el sistema y el valor apunta al descriptor de la clase.
SIGUE	- Al siguiente objeto.	A la siguiente clase.

4.3.1.2 TABLA DE CAMPOS. -

La Tabla de Campos TABFIELD es un arreglo de registro de 0..LIMSUPFIE. Los campos de registro son los siguientes:

IDEN - Nombre de la variable de instancia.

ADMI - Administrador del objeto asociado a la VI.

SIGUE - Indice al siguiente registro.

Inicialmente todos los registros se ligan para formar un almacen de registros controlados a través de la variable DISPFIELD. Cuando el sistema requiere algún registro este es tomado de esta lista.

Como en esta tabla se definen los campos declarados en cada una de las clases, esta organizada como un conjunto de listas, cada una definiendo los campos declarados en la clase y son accesados a través de un campo del Descriptor de Clase. Como se representó en la fig. 4.6.

4.3.1.3 TABLA DE SELECTORES. -

La tabla de selectores TABSELE es un arreglo de registros de LIMINFSEL..LIMSUPSEL en donde se describe al selector. El registro tiene los siguientes campos:

POSI - Posición del inicio del selector en la tabla de cadenas.
LONG - Longitud en caracteres del selector.

La tabla de cadenas TABCADE es un arreglo empacado de caracteres de 0..LIMSUPCAD en donde se guarda la cadena del selector.

De 0 a LIMINFSEL - 1 se encuentran los caracteres especiales usados como selectores. Estos no se guardan en estas tablas ya que su reconocimiento corre a cargo del Analizador Lexicográfico.

4.3.2 MODULO ADMI.

Es un módulo que es parte de la máquina virtual, sin embargo el compilador interactua con él para la creación y acceso de objetos en la memoria de esta máquina. Las principales rutinas que se utilizan son las siguientes:

- Function Newinstance [ADMI]. Crea un nuevo objeto no inicializado en memoria.
- Function getfield [ADMI]. Accesa un campo de algun objeto.
- Procedure storefield [ADMI]. Almacena información en un campo de algún objeto.

4.3.3 MODULO ANALEX.

Es el analizador léxico de texto escrito por el usuario; es llamado por el compilador cada vez que éste requiere del siguiente símbolo terminal. Consiste de los siguientes procedimientos:

```
PROCEDURE SCANNER (var token : integer; { Número del símbolo
                                          terminal reconocido }
                   var posprime : integer; {Primer valor semántico
}
                   var possegun : integer; {Segundo valor semántico
}
                   var posterce : integer  {Tercer valor
semántico});
```

Regresa en TOKEN el número de terminal reconocido y en POSPRIME, POSSEGUN y POSTERCE los valores semánticos de ese símbolo.

El funcionamiento es el siguiente: lee un símbolo de la cadena de entrada hasta que éste no sea un caracter de edición. Después lo analiza para saber a que símbolo terminal puede corresponder y, una vez determinado, asigna en la variable TOKEN su valor y establece sus valores semánticos.

Los símbolos de puntuación no tienen valores semánticos únicamente en la variable token es puesto su valor.

Los símbolos complejos tienen los valores semánticos siguientes:

<identificador> - En POSPRIME es puesta la posición del identificador en TABTEMP.

<caracter-especial> - En POSPRIME es puesta la posición del caracter en TABSELE (son caracteres usados como selectores en un método).

<class-inst> - En POSPRIME es puesta la posición que identifica la clase en TABTEMP.

<object-decl> - En POSPRIME es puesta la posición que identifica al objeto en TABTEMP. Si es un <identificador>.<identificador> pone en POSSEGUN la posición que identifica el administrador del objeto en TABTEMP.

<literal> - En POSPRIME pone la clase de la literal.

```
  Classfix - Un número entero.
Classfloa - Un número real.
  Classstr  - Una cadena de caracteres.
  Classbool - Si detecta TRUE o FALSE.
  Classund  - Si detecta NIL.
```

En POSSEGUN pone el tipo del valor (ver [ADMI]). Además, en la variable PRIMLIT pone el valor de la literal en caso de ser un número o la posición en memoria en donde instaló la literal en los casos restantes. Hay que hacer notar que los objetos true,

false y nil aunque se encuentran en memoria tienen una posición fija determinada en la inicialización del sistema.

El tercer tipo, las palabras reservadas, no generan ningun valor semántico.

Una característica de este procedimiento es la capacidad de saltarse los comentarios. Estos son delimitados por (* *). Después de saltarselo tiene que hacer una llamada recursiva a si mismo para regresar algún símbolo terminal.

Otra característica es la de tener cuidado de no "comerse" un caracter de más. Para lograrlo, utiliza el procedure retrae. Se usa para cuando se necesita otro caracter para decidir entre un terminal y otro. Como un ejemplo esta al leer el caracter ':' tiene que leer el siguiente caracter para decidir si es un caracter especial o un operador de asignación (:=). Si el siguiente caracter no fue un '=' lo regresa a la cadena por compilarse.

El procedimiento scanner, comprende las rutinas siguientes:

```
FUNCTION INSTALAID ( identificador :  tyiden ) :  integer;
```

Instala el identificador dado en la tabla de temporales TABTEMP y regresa como valor de la función su posición en ésta.

```
PROCEDURE TRAE ( var simbolo :  char );
```

Da en la variable SIMBOLO el siguiente caracter en la cadena que se esta compilando. Cuando esta se acaba da el caracter 'eot' (chr(3)). Lleva control de la linea del texto detectando el <CR> y además de la posición en esa linea (detectando los TAB's también).

```
PROCEDURE RETRAE ( simbolo :  char );
```

Regresa una posición el apuntador de la cadena que se esta compilando. Si el SIMBOLO es <CR> decrementa el contador de lineas.

```
PROCEDURE DAMEIDEN ( var identificador : tyiden;
                     var       simbolo : char;
                     var          cual : integer );
```

Da en la variable IDENTIFICADOR el identificador que empieza con el símbolo que recibe en la variable SIMBOLO. En caso de que el identificador sea una palabra reservada regresa en la variable CUAL la posición de éste en la tabla de palabras reservadas TABRESER, en caso contrario, CUAL es igual a cero.

Verifica que el identificador tenga como máximo 12 caracteres y si se pasa de este número lo trunca y manda un mensaje para notificarle al usuario de este hecho.

La variable SIMBOLO tiene como valor final el primer caracter que le sigue al identificador.

```
PROCEDURE DAMENUMERO ( var simbolo : char;
                           signo   : integer;
                       var valor   : real;
                       var tipo    : integer );
```

(* Dado el primer símbolo de una cadena de caracteres, esta rutina regresa el valor número de esa cadena y su tipo *)

```
PROCEDURE DAMECADENA ( var valor :  real );
```

(* Regresa una cadena del lenguaje. Da en valor su posición en memoria de esa cadena *)

```
PROCEDURE DAMEARREGLO ( VAR VALOR :  REAL );
```

(* Este procedimiento es el encargado de reconoser e instalar un arreglo constante en el sistema *)

4.3.4 MODULO COMPI.

Es el compilador del sistema. En este módulo, se encuentran las tablas de sintaxis, la rutina que maneja esas tablas y las rutinas de acciones semánticas.

4.3.4.1 ESTRUCTURAS GLOBALES DEL COMPILADOR. -

Son estructuras mantenidas sólo por el compilador. A continuación se mencionan las principales

- PILPARSER.- Es la pila utilizada por el manejador de las tablas de análisis sintáctico para realizar las acciones de shift-reduce. Tiene cuatro campos

  ESTA - Estado del analizador.
  PRIME - Primer valor semántico.
  SEGUN - Segundo valor semántico.
  TERCE - Tercer valor semántico.

- TABLIT.- Tabla en donde se guardan referencias a literales y variables globales. Es utilizada por el compilador para resolver referencias a este tipo de objetos.

- TABLOC.- Tabla de variables locales. En ésta se instalan los argumentos del método, sus variables locales y los argumentos de los bloque. También es utilizada por semántica para resolver referencias.

- TABCODI.- Tabla en donde se pone el código generado.

- PILCOMPI.- Pila para almacenar temporalmente listas de elementos de los que se necesita saber más antes de tomar una acción semántica.

4.3.4.2 RUTINAS PRINCIPALES DE COMPI. -

PROCEDURE CARGATABLAS;

Carga las reglas de la gramática y las tablas de análisis sintáctico.

```
PROCEDURE INSSELE ( donde : integer; { Localidad de TABSELE }
                    cadena : tyiden   { Nombre selector } );
```

Instala un selector en TABSELE. Se usa para la inicialización de ésta. La variable DONDE indica la localidad en TABSELE en donde será instalado y CADENA el selector en sí.

PROCEDURE INICOMPI;

Tiene por objetivo inicializar todas las variables y estructuras globales del módulo. Realiza las siguientes funciones:

- Llama a cargatablas.
- Inicializa diccionario:
  - Liga todos los registros del diccionario.
  - Carga las clases primitivas del sistema.
  - Inicializa CLASES, TOPCLASE, PUBLICAS.
- Inicializa TABFIELD:
  - Liga todos los registros.
  - Inicializa DISPFIELD.
- Inicializa TABSELE usando el procedure inssele.

```
PROCEDURE COMPILADOR(VAR texcade :  tytexcade);
```

Dada una cadena de entrada la compila. Contiene la rutina que maneja la tabla de sintaxis. Cada vez que realiza una reducción llama al procedure semantica.

```
PROCEDURE SEMANTICA ( numregla : integer; {Número de regla de
prod.}
                      var valuno : integer; {primer valor
semántico}
                      var valdos : integer; {segundo valor
semántico}
                     var valtres : integer; {tercer valor
semántico});
```

Dada una regla de producción **NUMREGLA**, cuyo lado derecho fue reconicido, realiza la acción semántica sobre esa regla y regresa los valores semánticos **VALUNO,VALDOS, VALTRES** que son el resultado de la acción semántica. El número de valores semánticos (en este caso de tres) fue determinado empiricamente, ya que analizando cada una de la reglas de producción, se determinó que como máximo necesitan de tres valores para representar la acción sobre esa regla pudiendo variar este número de cero a tres dependiendo de la regla.

```
FUNCTION BUSCAIDEN ( iden : tyiden;
                     apun : integer ) : integer;
```

Busca el identificador **IDEN** en el **DICCIONARIO** a partir de la lista apuntada por **APUN**. Si lo encuentra regresa como valor de la función la posición de **IDEN** en el diccionario. En caso contrario, regresa 0.

```
FUNCTION INSTALAIDEN ( iden : tyiden;
                       apun : integer ) : integer;
```

Si el identificador **IDEN** no se encuentra en el DICCIONARIO en la lista apuntada por **APUN**, lo instala al final de esa lista y regresa como valor de la función la posición en el diccionario en donde lo dejo. En caso contrario, regresa 0.

```
FUNCTION BUSCAFIELD ( indice : integer;
                      ncampo : integer;
                      iden   : tyiden;
                   var admi  : integer ) : integer;
```

Dado el índice en la variable **INDICE** sobre **TABFIELD** de la lista de campos de una clase en particular y el número de campos definidos, busca el nombre del campo **IDEN** en esa lista y si lo

encuentra regresa como valor de la función el offset que le corresponde a ese campo del objeto cuyo administrador es puesto en la variable ADMI. En caso contrario, regresa 0.

```
FUNCTION INSTALAFIELD ( iden : tyiden;
                        admi : integer;
                       sigue : integer ) : integer;
```

Instala IDEN, ADMI, SIGUE en la siguiente localidad libre de TABFIELD y regresa como valor de la función su posición en esa tabla. Si las localidades de TABFIELD se terminan, se manda un mensaje y el sistema aborta.

```
FUNCTION BUSCALOC ( iden :  tyiden ) :  integer;
```

Busca el identificador IDEN en TABLOC. Si lo encuentra regresa como valor de la función su posición en esta tabla. En caso contrario, regresa 0.

```
FUNCTION INSTALALOC ( iden : tyiden;
                      admi : integer;
                     valor : integer ) : integer;
```

Si el identificador IDEN no se encuentra en TABLOC lo instala junto con sus atributos ADMI y VALOR y regresa como valor de la función la posición en donde lo instalo. En caso contrario, regresa 0.

```
FUNCTION INSTALALIT ( valor : real;
                      flag : integer ) : integer;
```

Instala en TABLIT el valor de la literal VALOR y su tipo FLAG. Regresa como valor de la función su posición.

```
PROCEDURE CODIGO ( op : integer; { Operador }
                   p1 : integer; { parametro uno }
                   p2 : integer; { parametro dos }
                   p3 : integer  { parametro tres } );
```

Codifica la intrucción indicada y luego la instala en la siguiente localidad de TABCODI.

```
FUNCTION INSTALACOD :  integer;
```

Construye el objeto código y lo instala en memoria, regresando como valor de la función su posición en ésta.

```
FUNCTION INSTALAMET ( primero : integer;
                      actual : integer ) : integer;
```

Instala el objeto metodo apuntado por ACTUAL en la lista de metodos apuntada por PRIMERO, siguiendo la política de que se ordenan en orden ascendente de acuerdo al selector del método. Regresa como el valor de la función el primero de la nueva lista. Si ocurre algún error tal como que el selector del método ya existía regresa la misma lista que le llego.

En el capítulo 5, se complementan algunos aspectos vistos en éste capítulo, por ejemplo, se presentará el conjunto completo de instrucciones generadas por el compilador, ya que algunas dependen del contexto donde se encuentren las expresiones, aspecto que ahí se trata.

4.4 REFERENCIAS.

[1].- Alfred V. Aho, Jeffrey D. Ullman; Principles of Compiler Design; E.E.U.U., Addison-Wesley (1977); Cap. 4.

[2].- Alfred V. Aho, Jeffrey D. Ullman; Principles of Compiler Design; E.E.U.U., Addison-Wesley (1977); Cap. 6.

CAPITULO 5

MAQUINA VIRTUAL

La Máquina Virtual es un programa que ejecuta las instrucciones generadas por el compilador. Para ello, requiere accesar objetos; crear nuevos objetos y destruir lo no utilizados; crear el medio ambiente para la ejecución de un método, es decir, localizar todas las variables a que ese método tiene acceso; y por último, tiene un conjunto de métodos primitivos en donde el envio de mensajes termina.

En la presente implementación, la Máquina Virtual esta inspirada en el diseño de la Máquina Virtual del sistema Smalltalk-80 [1]. Las diferencias existentes serán presentadas cuando se trate cada una de sus componentes.

El concepto de Máquina Virtual nace del hecho de que el Hardware debe de adaptarse a las necesidades del Software, tomando en cuenta que éste último es más costoso. Por otro lado, el hecho de que el software se realice en función del hardware trae problemas de transportabilidad del primero de una configuración de hardware a otra (el software se hace dependiente del harware). Mientras que, si se realiza el software para una Máquina Virtual en específico, sólo ésta tiene que ser reescrita cuando el sistema se transporte a otra configuración de hardware, facilitando la labor de instalar un sistema.

El uso de la Máquina Virtual acarrea una desventaja: como las instrucciones son interpretadas por un programa, la ejecución es mas lenta que si fueran ejecutadas por hardware. Pero, para el desarrollo de nuevos sistemas, este hecho no es de mucha importancia, ya que, en principio, únicamente se esta interesado en la funcionalidad del sistema. Cuando se disponga, en un futuro, una versión del sistema bien probada, es factible pensar en la construcción del hardware para la mayor rapidez del sistema. Por el momento, el desarrollo del software se facilita mucho usando una Máquina Virtual, ya que lo hace más flexible, adaptandose la máquina a las necesidades de nuestro sistema.

5.1 ESTRUCTURA DE LA MAQUINA VIRTUAL.

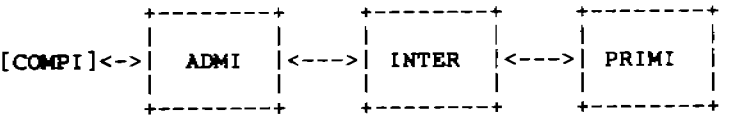
La Máquina Virtual debe tener una estructura tal que facilite la implementación de un Lenguaje Orientado a Objetos y al envio de mensajes. Una de las estructuras que ayuda alcanzar éste propósito es la Máquina de Stack. Esta, comprende dos estructuras principales: una memoria, en donde son depositados los objetos del sistema, y una pila de ejecución, la cual se utiliza en la evaluación de las expresiones. El manejo de estas estructuras corre a cargo de tres módulos implementados:

1. Un Administrador de Memoria [ADMI].

2. Un Intérprete [INTER]

3. Un conjunto de Rutinas Primitivas [PRIMI].

La representación siguiente muestra la interacción entre estos tres módulos:



El Administrador de Memoria, tiene las siguientes funciones:

1. Mantener la integridad de todos los objetos del sistema.

2. Realizar un manejo eficiente de la memoria.

3. Crear nuevos objetos.

4. Accesar sus variables de instancia.

5. Liberar el espacio que ocupan los objetos no referenciados.

La función del intérprete es ejecutar las instrucciones generadas por el compilador, para ello utiliza la pila de ejecución, en donde es creado el medio ambiente para la ejecución de un método, además ahí son depositados los objetos referenciados por las instrucciones para la evaluación de las expresiones. El intérprete es auxiliado por el Administrador de Memoria para el acceso a los objetos.

El conjunto de rutinas primitivas responden a mensajes que se envían a los objetos primitivos del sistema, tales como números y cadenas. Es en estas rutinas en donde termina la recursión de envio de mensajes. Es decir, para realizar una

respuesta a un mensaje enviado, no requiere de enviar mensajes a otros objetos, sino que, siempre depositan un objeto en la pila de ejecución como valor al mensaje recibido.

5.2 ADMINISTRADOR DE MEMORIA.

El administrador de memoria provee al intérprete una interfase de acceso a los objetos. El administrador de memoria consiste de dos estructuras principales: La Memoria y la Tabla del Administrador.

La memoria es un conjunto contiguo de localidades en donde se localizan todos los objetos del sistema. Todo en el sistema es un objeto y desde el punto de vista de almacenamiento, la memoria esta dividida en bloques, uno para cada objeto. Esos objetos son creados en memoria, pero algunos se utilizan sólo temporalmente y después son deshechados. Por ésta razón, es necesario llevar un control de los mismos: saber cuando estan referenciados y cuando dejan de estarlo para liberar el espacio que ocupan. Para ello se creó el administrador de memoria, el cual utiliza una tabla para llevar ese control.

El administrador de memoria provee una interfase para el acceso a los objetos. A cada objeto se le asocia un identificador único llamado "Apuntador al Objeto" (A.O.). El administrador de memoria y el intérprete se comunican con los objetos a través del A.O..

Si un A.O. fuera la dirección de memoria ocupada por el objeto, sería rápido accesar un objeto dado su A.O.. Sin embargo, en esta implementación el A.O. es un apuntador indirecto al objeto a través de la tabla mantenida por el administrador de memoria. Esto permite al administrador de memoria llevar un control de las referencias a ese objeto y del espacio de memoria que ocupa. También, esa tabla permite administrar el espacio de memoria disponible para la creación de objetos dinámicamente. La tabla le permite al administrador mover un objeto en memoria sin afectar a los objetos que lo refieren. Esto es útil para la implementación de una Memoria Virtual (no contemplada en este trabajo). Todo ésto asegura que el administrador de memoria es la única entidad en el sistema que maneja y permite cambiar el valor de una localidad de memoria.

El administrador de memoria asocia a cada A.O. con un conjunto de otros A.O.. Este conjunto corresponde a las variables de instancia declaradas para ese objeto. Además, a todo A.O. le es asociado el administrador del objeto (la clase). La referencia a ésta es a través de la posición de la clase en el diccionario (las clases son objetos públicos). La referencia a un objeto es mostrada en la fig. 5.1. En la figura, las localidades 1..n corresponden a las variables de instancia del objeto; la localidad 0 a su administrador.

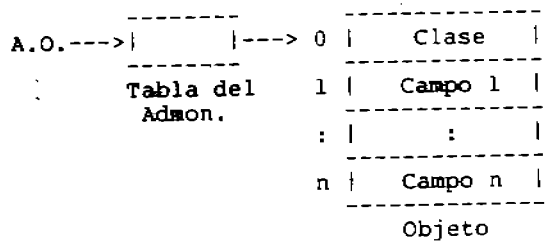


FIG. 5.1

El valor asociado a una variable de instancia puede ser cambiado continuamente, y su acceso es a través del A.O., más un desplazamiento asociado a ella. El administrador una vez establecido no puede cambiar.

El Administrador de memoria provee las siguientes funciones al intérprete:

1. La de accesar el valor de las variables de instancia del objeto. La llamada es:

   valor := getfield(Objeto, Indice)

   Objeto - Es la referencia al objeto (A.O.).
   Indice - Es el número de variable de instancia a accesar.
   valor - Es el valor que corresponde a la variable de instancia.

2. La de cambiar el valor de una variable de instancia. La llamada es:

   storefield(Objeto, Indice, Valor, Tipo)

   Objeto - Es la referncia al objeto (A.O.).
   Indice - Es el número de variable de instancia.
   Valor - Es el nuevo valor de la variable de instancia.
   Tipo - Es el tipo del valor.

3. La de crear un nuevo objeto. La llamada es:

   Objeto := newinstance(Long)

   Long - Es el número de variables de instancia del nuevo objeto.
   Objeto - Es el A.O. del nuevo objeto.

   La clase es puesta automáticamente por el sistema por medio de la función storefield.

4. La de obtener la longitud de un objeto dado. La llamada es:

```
Long := getlong(Objeto)
```

Objeto - Es el A.O. del objeto en cuestión.
Long - Es la longitud del Objeto.

5. Obtener el administrador del objeto. La llamada es

```
admi := getclass(Objeto)
```

Objeto - Es el A.O. del objeto.
admi - Administrador de ese objeto.

No hay una función explícita del administrador de memoria para remover un objeto que ya no esta siendo usado, porque es reclamado automáticamente. Un objeto es reclamado cuando no hay nadie que lo referencie. Esto se realiza, contando los objetos que apuntan a un objeto determinado (cuenta de referencias). Cuando ésta llega a cero el objeto se libera de inmediato.

Por lo anterior, se tienen las siguientes funciones que no son vistas por el sistema sino sólo por el administrador:

- Increrefe(Objeto)
- Decrerefe(Objeto)
- Retinstance(Objeto)

Cuando a un campo de un objeto le es asociado otro valor (en una asignación), al nuevo valor le es aplicada la función increrefe que incrementa en uno la cuenta de referencias. Al valor anterior se le aplica la función decrerefe que decrementa en uno la cuenta de referencias y verifica que no se igual a cero, si lo es, invoca a la función retinstancia que se encarga de liberar ese espacio y ponerlo en el espacio disponible.

Las tres operaciones anteriores son bastantes costosas, tomando en cuenta que su frecuencia de uso es muy grande. Para disminuir su uso, ciertos objetos no tienen cuenta de referencias. Estos objetos son las literales primitivas que no ocupan más de una localidad de memoria, a saber, éstas son los números enteros, los números reales y los caracteres. Para estos objetos el propio A.O. lleva implícito su valor. Para diferenciar estos objetos de los demás a cada localidad de memoria le es asociado un par de bits. Los valores posibles de esos bits nos da la información que contiene la localidad de memoria. Estos son

00 - El valor es un apuntador a un objeto complejo
(un objeto con variables de instancia).
01 - El valor es un número entero.
10 - El valor es un número real.
11 - El valor es un caracter.

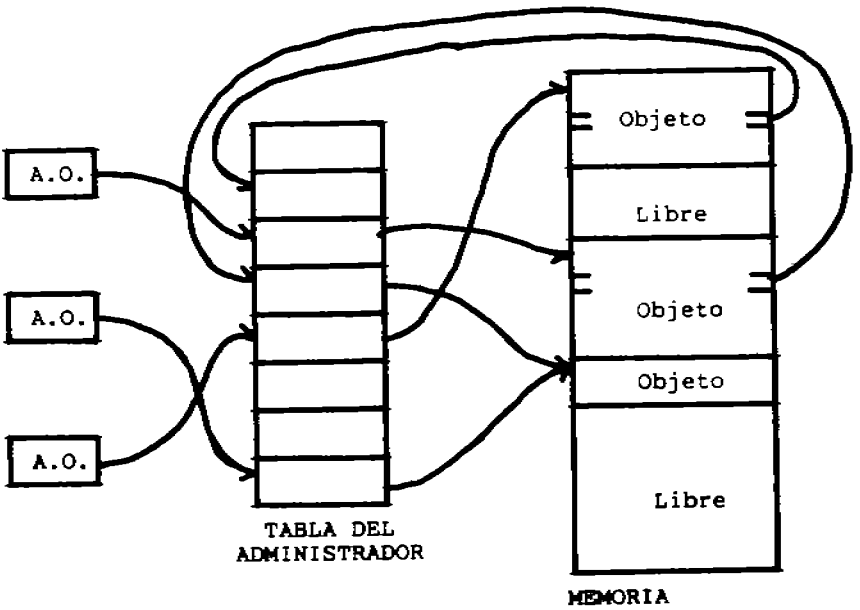


FIG 5.2

La configuración del administrador de memoria es vista en la fig. 5.2. En esta figura se aprecia la memoria, la tabla utilizada por el administrador y como son referenciados los objetos por otros. Hay algunos A.O. que no provienen de memoria, éstos se encuentran en la pila de ejecución del intérprete o bien en el diccionario global del sistema. Esto es porque éstas estructuras son independientes de la memoria y su manejom se realiza con otras rutinas.

Cada localidad de la Tabla del Administrador se compone de los siguientes campos:

DIRE - Es la dirección en memoria de la localidad en donde empieza el objeto.

LONG - Es la longitud del objeto.

REFE - En este campo se lleva el número de otros objetos

que referencian a éste.

SIGUE - Es una dirección a la misma tabla que nos indica cual es la siguiente localidad ocupada por el administrador. Este campo es usado para la aplicación del algoritmo de creación y liberación de objetos.

Las funciones del administrador de memoria del sistema SS son las mismas que provee el administrador del sistema ST-80, sin embargo, su diseño e implementación son diferentes. Algunas ideas para la realización del administrador fueron tomas de la referencia [2], en la cual se discuten los algoritmos que se presentan a continuación.

5.2.1 ALGORITMOS PARA ADMINISTRAR MEMORIA.

Inicialmente la memoria es un sólo bloque libre de longitud N. Cuando es requerida por el sistema memoria de longitud L, es tomada del principio del bloque libre y la dirección del comienzo de ese bloque de memoria es regresada al sistema, entonces, la longitud del bloque libre se reduce en L localidades.

Mientras el sistema no libere área no hay problema, ya que se obtiene de un sólo lugar. Pero cuando un bloque es liberado por el sistema, el espacio ocupado por éste puede ser utilizado nuevamente. Esto requiere mantener una lista de bloques libres, en donde se van insertando los bloques liberados. Entonces, el método para seleccionar un bloque de memoria cuando es requerida no es tan directo, ya que un bloque de la lista no puede ser lo suficientemente grande para satisfacer un requerimento, o puede ser más grande que el requerido.

Hay dos estrategias para mantener una lista de bloques libres cuando su longitud varía (el cual es el caso): REG y GARB [1].

REG - Los bloques libres de la lista se ligan en orden ascendente de acuerdo a su longitud. Cuando un bloque es requerido se busca en la lista hasta que un bloque de longitud igual o mayor se encuentra. Si es igual, se regresa la dirección del bloque al sistema y el bloque se quita de la lista. Si el bloque encontrado es mayor se divide en dos, una parte es la longitud requerida y se regresa su dirección. La otra parte es reinsertada en el lugar correspondiente de la lista. Cuando un bloque se libera, se inserta en el lugar apropiado de la lista (de acuerdo a su longitud).

GARB - La lista de bloques libres se ordena en función de su dirección en memoria. Es decir, la lista de los bloques libres es tal que las direcciones se incrementan continuamente. El rastreo es similar que al mostrado en la estrategia anterior,

sólo que cuando el bloque es más grande que el requerido se regresa la parte solicitada y el resto es dejado en la misma posición de la lista.

Cuando el sistema libera un bloque, se inserta en el lugar apropiado de acuerdo a su dirección. Pero una revisión extra es hecha antes para ver si es contiguo con algún otro bloque libre (ya sea el anterior, posterior o ambos). En este caso, se juntan los bloques para formar uno solo. Por ejemplo, se analiza el caso de la fig. 5.3, que muestra una configuración de la memoria. Si el objeto 2 es liberado la memoria queda como se muestra en la fig. 5.4.

Ambos algoritmos tienen sus ventajas y desventajas (tomadas de la referencia [1]), y a continuación se presentan:

	VENTAJAS	DESVENTAJAS
REG	Más rápido acceso a bloques libres y liberación de ocupados.	Rápida fragmentación de la memoria.
GARB	Retarda en mucho la fragmentación de memoria.	Busqueda lenta de un bloque libre y lenta liberación debido al cheque extra.

Si se toma como criterio que al fragmentarse la memoria se tiene que aplicar un algoritmo para compactarla, lo cual es bastante lento, se prefiere retardar este hecho utilizando el segundo algoritmo. Así, en lugar de compactar la memoria frecuentemente, se distribuye el tiempo ocupado en esto para la creación y liberación de objetos.

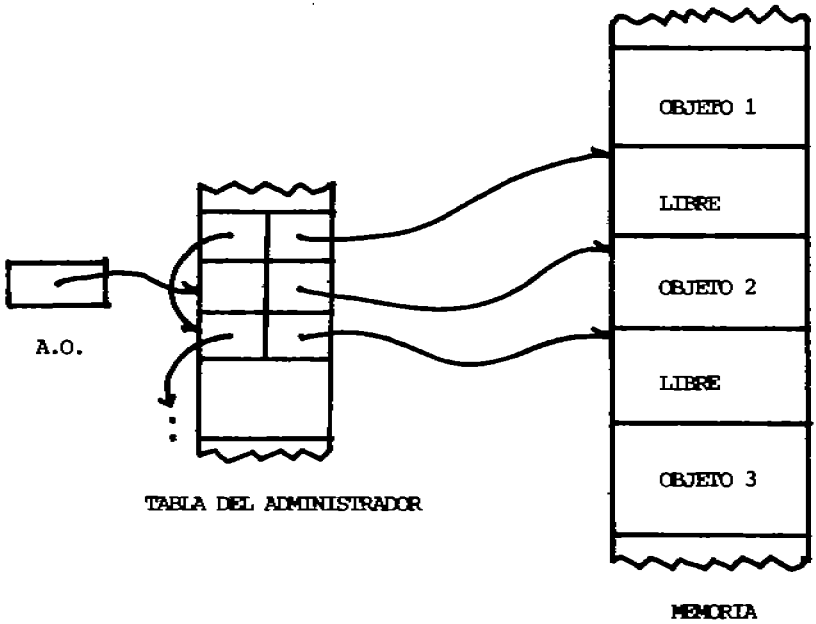

OBJETO 1
LIBRE
OBJETO 2
LIBRE
OBJETO 3
A.O.
TABLA DEL ADMINISTRADOR
MEMORIA

FIG. 5.3

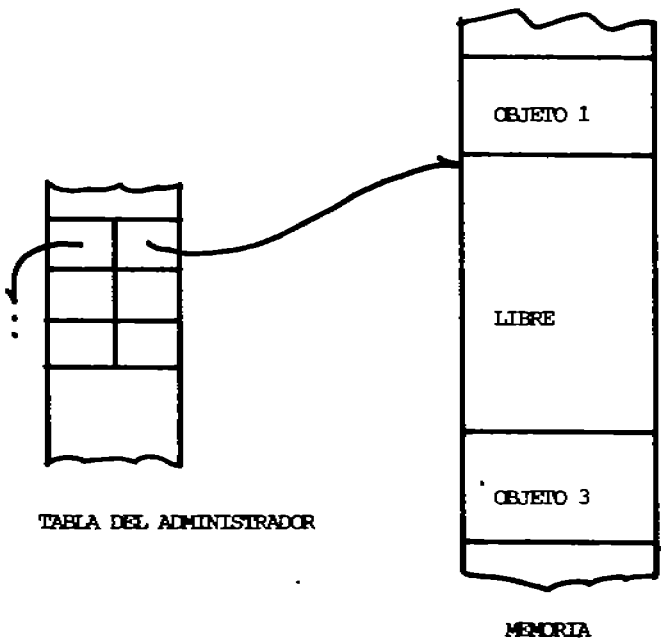


FIG. 5.4

Pero aún existe la probabilidad de que se fragmente la memoria, así que se realizó un algoritmo para compactar tanto la memoria como la misma tabla del administrador. La memoria, se compacta cuando al tratar de localizar un bloque libre de longitud L no existe un bloque de tal longitud, pero la suma de varios bloques libres si lograrían satisfacer esa petición. La tabla del administrador, se compacta cuando al tratar de localizar un bloque libre de longitud L si existe el bloque en memoria para satisfacer la petición, pero no hay lugar en la tabla del administrador para referenciarlo. En ambos casos el algoritmo se aplica cuando se trata de localizar un nuevo objeto.

Se podría hacer un algoritmo rápido para compactar (en ambos casos) si se tuvieran los bloques ocupados ligados entre si en orden ascendente por direcciones, pero esto retarda la creación y liberación de objetos. Como se vió que la fragmentación es poco probable, se decidio no hacerlo. Así, los bloques ocupados se encuentran desligados en la tabla del administrador y en el caso que se requiera compactación, ésta será lenta en favor de la creación y liberación de objetos.

5.3 EL INTERPRETE.

El intérprete es la parte de la Máquina Virtual que ejecuta las acciones descritas en las instrucciones del método. La información necesaria para implementar el intérprete es la descripción de las instrucciones, la representación de los métodos y la técnica para encontrar el método a ejecutarse cuando se envía un mensaje.

El intérprete utiliza una pila para la evaluación de las expresiones. El mecanismo es el siguiente: el receptor y los argumentos del mensaje son introducidos en la pila, entonces se ejecuta la instrucción que envía el mensaje. Por los mecanismos que posteriormente se explican, se localiza el método que responde a ese mensaje. Este evalua las expresiones que contiene y deja en la pila un valor, que es el resultado del mensaje. Entonces, el intérprete remplaza el receptor y los argumentos por ese resultado y termina la evaluación de la expresión; por ejemplo la expresión siguiente:

```
3 <= * ( 4 <= + 5 )
```

El compilador la traduce a las instrucciones siguientes:

instrucción	stack
-1- PUSH 3	(3)
-2- PUSH 4	(3 4)
-3- PUSH 5	(3 4 5)

```
-4- SEND +              (3 9)
-5- SEND *              (27)
```

En las primeras tres instrucciones los objetos son introducidos a la pila. Cuando la cuarta instrucción se ejecuta, el mensaje + se envía al segundo objeto del stack (4) con el objeto del tope como argumento (5). Cuando el intérprete ejecuta el método correspondiente al mensaje + en la clase de los números enteros, los objetos 4 y 5 son usados por éste y posteriormente remplazados por el valor del método, en este caso el objeto 9. Continúa la ejecución con la instrucción -5-. Esta instrucción produce un efecto similar a la anterior, dejando el objeto 27 en la pila.

Otra de las funciones del intérprete es almacenar el objeto localizado en el tope de la pila como el valor de una variable; por ejemplo, la expresión

```
A := 3 <= + 4
```

se traduce en las instrucciones

```
instrucción             pila

-1- PUSH 3              (3)
-2- PUSH 4              (3 4)
-3- SEND +              (7)
-4- STORE en A          (7)
```

Las instrucciones 1-3 tienen el mismo efecto al ejemplo anterior. Cuando se ejecuta la instrucción -4- el intérprete almacena el tope de la pila en la variable A.

El tope de la pila puede ser regresado como el valor de un método. Por ejemplo

```
A := 3
<- A
```

genera

```
-1- PUSH 3              (3)
-2- STORE en A.         (3)
-3- POP                 ()
-4- PUSH A.             (3)
-5- RETURN              (3)
```

La instrucción return tiene como función terminar la ejecución de un método y regresar un objeto como valor de éste, es decir, remplazar el receptor y argumentos por el tope del stack.

5.3.1 DIRECCIONAMIENTO DE VARIABLES.

Los métodos son implementados como objetos especiales cuyos campos contienen instrucciones, más un grupo de apuntadores a otros objetos llamado el marco de literales (fig. 4.9 del capítulo 4). El intérprete usa la función getfield del administrador de memoria para traer (fetch) la siguiente instrucción a ejecutar. El intérprete toma las acciones correspondientes para ciertas acciones como pop y return, pero para otras necesita de más información. En particular, para las instrucciones PUSH y STORE necesita conocer donde encontrar los objetos a introducir a la pila ó el objeto al cual se le asignará el tope de la pila; para la instrucción send, se necesita conocer el selector del mensaje y el número de argumentos del mensaje.

El código fuente para un método contiene nombres de variables y literales, pero las instruciones de la Máquina Virtual están definidas sólo en términos de desplazamientos relativos al objeto tratado. Desde el punto de vista de la Máquina Virtual, hay tres tipos de variables: variables locales al método (temporales), variables locales al receptor del mensaje (variables de instancia) y variables globales que son mantenidas en un diccionario. Notese que tanto variables públicas como de clase se tratan como globales, el compilador se encarga de permitir o no los accesos a unas y otras.

El compilador translada referencias a esas variables en instrucciones que son referencias a desplazamientos de alguna de las siguientes áreas:

1. Area del Receptor. Para referenciar las VI del receptor del mensaje.

2. Area de Temporales. Para referenciar al receptor, a los argumentos o a las variables locales.

3. Area de Literales. Para referenciar a literales o variables globales.

Las variables de instancia se transladan usando un campo del Descriptor de Clase que contine una lista de variables de instancia accesibles por el método, el desplazamiento es calculado con respecto a la posición de la variable de instancia en esa lista.

La asignación de desplazamientos respecto al área de temporales se realiza cuando el compilador translada un método asociando el nombre de la variable temporal a su desplazamiento en el área de variables temporales del método compilandose en ese momento.

El compilador crea instancias para las literales, pone el apuntador a esas instancias en el marco de literales del método, y produce instrucciones de acuerdo a su posición en el área de

literales. Para variables globales, el compilador usa el diccionario del sistema que asocia nombres globales a referencias indirectas a objetos. Apuntadores a una localidad del diccionario, que hace referencia indireta al objeto, son puestos también en el marco de literales del método. Las instruciones para accesar globales son codificadas como referencia indirecta a través de un desplazamiento en el marco de literales.

Cuando el intérprete ejecuta un método mantiene las siguientes estructuras: una pila de ejecución, un área de temporales (dentro de la misma pila), un apuntador al receptor del mensaje, un apuntador al mismo método y uno a la siguiente instrucción. Se pone como ejemplo el siguiente método:

```
+ puntol =>
    DCL sumX sumY ;
    sumX := x <= + ( puntol <= x ) ;
    sumY := y <= + ( puntol <= y ) ;
    <- .punto <= creaX: sumX y: sumY
END
```

Este es un método de la clase Punto que realiza la suma de las VI de un punto receptor y otro punto como argumento. Las instancias de punto tienen dos variables de instancia que son "x" y "y", que representan la posición del punto en el plano. El método contiene dos variables locales que recuerdan la suma de las VI temporalmente. También se referencia a una variable global, la clase Punto. El estado del intérprete al ejecutar este método se muestra en la fig. 5.5. En esta figura se aprecia las áreas donde se encuentran variables referenciables por el método.

El intérprete usa las funciones getfield y storefield del administrador de memoria para accesar y modificar valores del área de temporales, del área del receptor, del área de literales y del objeto método. El manejo de la pila no lo hace a través de estas funciones, ya que la pila es una estructura independiente de la memoria. Su manejo es directo para introducir y retirar objetos. Por su propia independencia la pila debe de tener un mecanismo para liberar el espacio no usado y al mismo tiempo los objetos que referencia en ese espacio.

Al término de la ejecución de un mensaje, al ejecutar la instrucción de retorno, el intérprete remplaza el área de temporales por el objeto que esta en el tope de la pila. Al remplazarla, los objetos referenciados en esta área se liberan, o sea, el intérprete le comunica al administrador de memoria que esos objetos ya no se usan, el administrador toma las medidas mencionadas en la sección anterior.

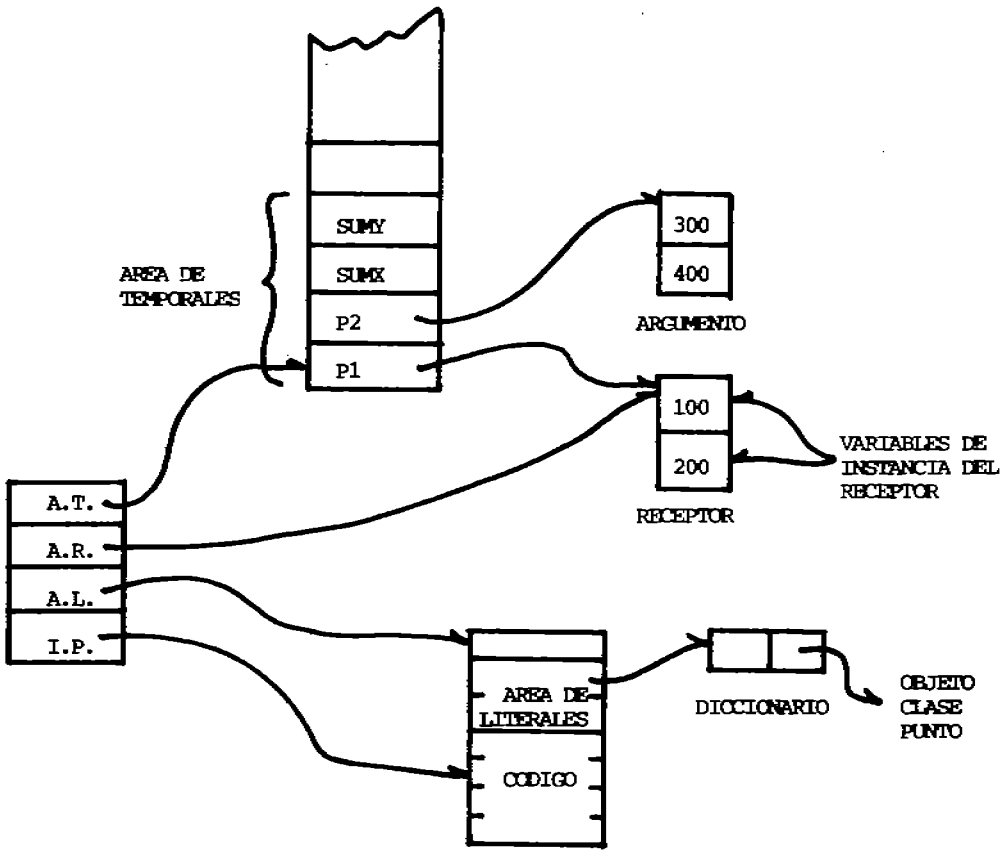

SUMY
SUMX
P2
P1
AREA DE
TEMPORALES
300
400
ARGUMENTO
100
200
RECEPTOR
VARIABLES DE
INSTANCIA DEL
RECEPTOR
A.T.
A.R.
A.L.
I.P.
AREA DE
LITERALES
CODIGO
DICCIONARIO
OBJETO
CLASE
PUNTO

FIG. 5.5

5.3.2 BUSQUEDA DEL METODO.

Cuando un mensaje se envía, el receptor y argumentos deben de ser localizados, y entonces el método apropiado debe de ser encontrado por el intérprete. La técnica usada es incluir en el Descriptor de Clase una lista de métodos, que asocia el selector con el objeto código (del método). La lista de métodos fué descrita en el capítulo 4. El selector y un apuntador al objeto código son mantenidos en el método. La instrucción que envía un mensaje (SEND) le indica al intérprete el selector y número de argumentos necesarios por el método. El intérprete realiza las siguientes acciones:

1. Encontrar el receptor del mensaje. El receptor esta debajo de los argumentos en la pila. El número de argumentos es indicado en la instrucción.

2. Accesar la lista de métodos. Esta se localiza en el Descriptor de Clase de la clase del receptor.

3. Buscar el selector del método en la lista de métodos. El selector es indicado en la instrucción.

4. Si el selector se encuentra, se manda a ejecutar el nuevo objeto código.

5. Si el selector no se encuentra, una nueva lista de métodos debe de ser rastreada (regresar al paso 3). La nueva lista de métodos se localiza en la superclase de la última clase cuya lista de métodos fue accesada. Este ciclo puede ser repetido varias veces, viajando a través de la cadena de superclases.

Si el selector no se encuentra en la clase del receptor ni en sus superclases, se notifica un error y la ejecución de las intrucciones siguientes a la instrucción SEND se suspende. El mensaje de error es "Mensaje no entendido".

5.3.3 EJECUCION DE UN METODO.

El intérprete ejecuta las instrucciones contenidas en el objeto tipo código del método. Para realizarlo necesita conocer la siguiente información:

1. Apuntador al receptor del mensaje que invocó al método (área del receptor).

2. Apuntador al área de temporales del método.

3. Apuntador al mismo objeto código en donde se encuentran las instrucciones a ser ejecutadas y el área de literales.

4. Apuntador a la siguiente instrucción a ejecutar.

Estos cuatro valores representan el estado del intérprete.

La primera acción que toma el intérprete es accesar con la función getfield la primera instrucción a ejecutarse, esto lo indica el campo 1 del objeto código. Con éste valor inicializa el apuntador de instrucciones e inicia el siguiente ciclo

- Traer la instrucción a ejecutar (fetch).
- Incrementar el apuntador de instrucccciones.
- Ejecutar la instrucción.

Este ciclo se repite hasta que una instrucción de retorno se ejecute.

5.3.4 CONTEXTO DE UN METODO.

Ciertas instrucciones como PUSH, STORE y JUMP cambian en muy poco el estado del intérprete. Sólo el apuntador de instrucciones cambia una vez que se ejecutan. Los otros valores del estado del intérprete no cambian. Existen instrucciones tales como el SEND y el RETURN que cambian completamente el estado del intérprete durante su ejecución. Cuando un mensaje se envía (con la instrucción SEND), las cuatro partes del estado del intérprete se modifican para ejecutar un nuevo método en respuesta al mensaje. El estado anterior del intérprete debe recordarse, ya que, las instrucciones después del SEND se ejecutarán una vez que la instrucción RETURN se ejecuta en el nuevo método y un valor del mensaje es regresado.

El intérprete guarda su estado en un objeto llamado contexto. Puede haber muchos contextos en el sistema a la vez. El contexto que representa el estado actual del intérprete se llama contexto activo. Cuando la instrucción SEND en el contexto activo requiere de un nuevo método a ser ejecutado, el contexto activo se suspende y un nuevo contexto es creado y hecho activo. El contexto suspendido retiene el estado asociado con el método original hasta que el contexto llegue a ser activo otra vez. El nuevo contexto activo debe recordar el contexto que fue suspendido para que pueda ser activado cuando un resultado sea regresado.

5.3.5 CONTEXTO DE BLOQUE.

El contexto que se presentó en la sección anterior representa una instancia de la clase contextoMet. Un contextoMet representa la ejecución de un objeto código. Hay otro tipo de

contexto en el sistema el cual se representa por las instancias de la clase contextoBlo. Un contextoBlo representa a un bloque en el fuente de un método. Este contexto es creado en la ejecución de un contextoMet. El siguiente ejemplo muestra las instrucciones generadas por el compilador al compilar un método que contiene un bloque.

```
inicia: arreglo =>
    1 <= to: (arreglo<=long) do:
        [:indice| arreglo<=en:indice pon: 0]
END
```

- mete (1) en la pila
- mete (arreglo) en la pila
- Envía el mensaje (long) con un argumento.
- Instruccion BLOCKCOPY
- Salta las siguientes 6 instrucciones
- Saca el valor del tope de la pila y almacenalo en la segunda localidad del área de temporales.
- mete (arreglo) en la pila.
- mete (indice) a la pila.
- mete (0) a la pila.
- Envía el mensaje (en:pon:) con dos argumentos.
- Regresa el valor del tope de la pila como valor del bloque.
- Envía el mensaje (to:do:) con dos argumentos.
- Regresa al receptor del mensaje como valor del método.

Una nueva instancia de contextoBlo se crea al ejecutar la instrucción BLOCKCOPY. Este nuevo objeto comparte muchas partes del contexto en el cual fue creado: el receptor, el área de temporales y el mismo objeto código. El nuevo contextoBlo tiene además su propio apuntador a instrucciones. La siguiente instrucción al BLOCKCOPY es el JUMP que salta todas las acciones descritas en el bloque.

El método del ejemplo crea un nuevo contexto pero no ejecuta sus instrucciones. Estas serán ejecutadas en otro ambiente cuando se le envie a este contexto el mensaje ejecuta: con un argumento (indice). La instancia de contextoBlo responde a ese mensaje siendo el nuevo contexto activo, lo cual causa que las instrucciones del bloque sean ejecutadas por el intérprete. Antes de que el bloque llegue a ser activo, el argumento de ejecuta: es metido en la pila de ejecución. La primera instrucción ejecutada por el bloque es almacenar ese valor en una localidad libre del área de temporales.

Un bloque puede regresar valores de dos maneras. Después de que las instrucciones han sido ejecutadas, el valor final en la pila es regresado como el valor del mensaje ejecuta ó ejecuta:. El bloque también puede regresar un valor al mensaje que invoca al método que creo la instancia de contextoBlo. Esto es hecho cuendo se usa un operador de retorno en el bloque. De ejemplo

pondremos un método del ejemplo 3.4

```
indiceDe: nombre =>
    1 <= to: (nombres<=long) do: [:indice |
        (nombres<=en:indice)<=eq: nombre<=
                iftrue:[<-indice];
    <- 0
END
```

En éste método si el nombre se encuentra en el arreglo de nombres el bloque regresa como valor del método ese índice, si no el método regresa 0.

5.3.6 DESCRIPCION DE CONTEXTOS.

En esta sección se analizará la estructura de los objetos que representan un contexto. Las instancias de contextoMet (contexto de un método) tienen los siguientes campos.

- Llamador. Se refiere al contexto que lo activo.
- Apuntador a la siguiente instrucción a ejecutarse.
- Apuntador al objeto código que contiene las instrucciones a ejecutar del método.
- Apuntador al receptor del mensaje que invocó al método (área del receptor).
- Apuntador al área de temporales.

Las instancias de contextoBlo (contexto de bloque) tienen los siguiente campos

- Llamador. Apuntador al contexto que activo el bloque. Es decir el que mando el mensaje ejecuta.
- Apuntador a la siguiente instrucción del bloque que se ejecutará.
- Número de argumentos del bloque.
- Primera instrucción del bloque.
- Apuntador al contexto dueño del bloque (en donde se creo).

La fig. 5.6 muestra un contexto de método y su objeto código asociado y la fig. 5.7 muestra la configuración de un contexto bloque y su dueño asociado.

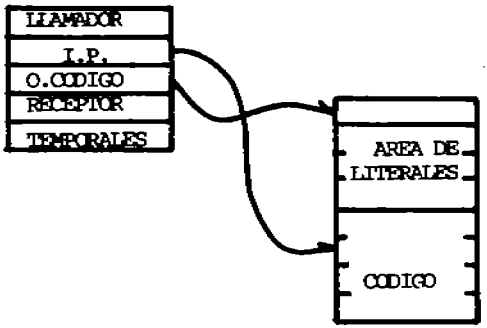

LLAMADOR
I.P.
O.CODIGO
RECEPTOR
TEMPORALES
AREA DE LITERALES
CODIGO

FIG. 5.6

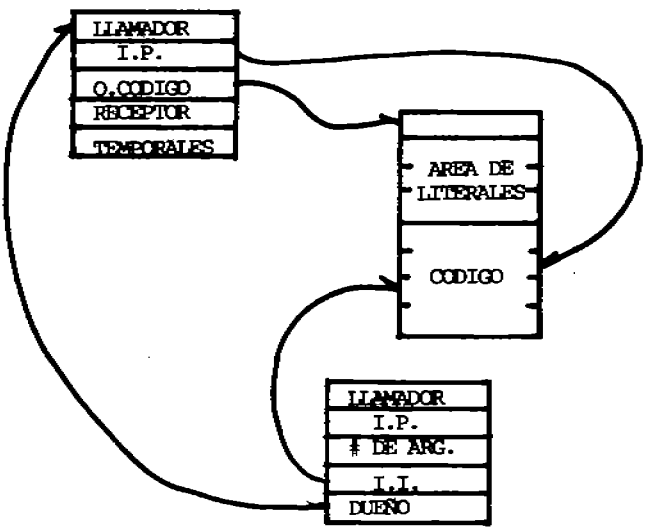

LLAMADOR
I.P.
O.CODIGO
RECEPTOR
TEMPORALES
AREA DE LITERALES
CODIGO
LLAMADOR
I.P.
DE ARG.
I.L.
DUEÑO

FIG. 5.7

Notese que, un contexto bloque guarda en uno de sus campos el número de argumentos que necesita, al momento de activarlo se revisa que ese número sea correcto, en caso contrario se manda un mensaje de error.

El contexto que representa el método o bloque que actualmente esta ejecutandose se le llama contexto activo. El intérprete carga en sus registros ese contexto para mayor rapidez de ejecución. Los registros que tiene el intérprete son:

ContextoActivo	- Contiene el contexto en si mismo. Puede ser un método o un bloque.
ContextoDueño	- Si el contexto activo es un método, el contexto dueño es el mismo contexto. Si el contexto activo es un bloque, el contexto dueño es el contenido del campo 5 del contexto activo.
Método	- Es el objeto código que contiene las instrucciones que se estan ejecutando.
Receptor	- Es el objeto que recibe el mensaje que invocó al método.
Temporales	- Apuntador al área de temporales del método.
I.P.	- Apuntador a la siguiente instrucción a ejecutarse.
Llamador	- Al contexto que activó al actual.

Siempre que el contexto activo cambia todos los registros se deben de actualizar. Antes de activar un nuevo contexto el intérprete guarda en el actual el I.P. porque es el único que cambia.

El intérprete del sistema SS es similar al del sistema ST-80 [1], pero hay algunas diferencias en la estructuras de los contextos y en la manera de ejecutar las instrucciones; en el sistema ST-80 cada contexto tiene su propia pila de ejecución; en el sistema SS la pila es una estructura independiente a los contextos.

5.3.7 CONJUNTO DE INSTRUCCIONES.

En esta sección, se presenta el conjunto de instrucciones que el intérprete reconoce; el formato de una instrucción consiste de cuatro campos, el primero se refiere al código de operación (CO) y los otros tres (C1, C2, C3) dependen de la instrucción. A continuación se listan las instrucciones:

- PUSH.- Introduce un objeto a la pila de ejecución.

  CO=0; C1=AREA; C2=DESPLAZAMIENTO; C3=ADMI

- POP.-Retira un objeto de la pila de ejecución.

  CO=1; C1=X; C2=X; C3=X

- CHECK.-Valida si un argumento es del tipo que se esperaba.

  CO=2; C1=AREA; C2=DESPLAZAMIENTO; C3=ADMI

- ASIG.-Asigna el tope de la pila a una variable.

  CO=3; C1=AREA; C2=DESPLAZAMIENTO; C3=ADMI

- SEND.-Envía un mensaje.

  CO=4; C1=X; C2=NUMARG; C3=SELE

- BLOCKCOPY.-Crea un contexto de bloque.

  CO=5; C1=X; C2=X; C3=X

- DESP.-Despliega en la pantalla del usuario el valor de un mensaje.

  CO=6; C1=X; C2=X; C3=X

- JUMPN.-Salto incondicional relativo hacia atraz.

  CO=7; C1=X; C2=DESPLAZAMIENTO; C3=X

- SENDP.-Envía un mensaje a un método primitivo.

  CO=8; C1=X; C2=X; C3=NUMPRI

- RET1.-Retorno de un método.

  CO=9; C1=X; C2=X; C3=X

- RET2.-Retorno de un bloque.

  CO=10; C1=X; C2=X; C3=X

- JUMPP.-Salta incondicional relativo hacia adelante.

  CO=11; C1=X; C2=DESPLAZAMIENTO; C3=X

- DUP.-Duplica el tope de la pila de ejecución.

  CO=12; C1=X; C2=X; C3=X

- JMPF.-Salta hacia adelante si el tope de la pila es el objeto FALSE.

  CO=13; C1=X; C2=DESPLAZAMIENTO; C3=X

- JMPT.-Salta hacia adelante si el tope de la pila es el objeto TRUE.

CO=14; C1=X; C2=DESPLAZAMIENTO; C3=X

En donde:

- AREA.-Se refiere a alguna de las tres áreas existentes: Area del Receptor (0), Area de literales (1), Area de temporales (2).

- DESPLAZAMIENTO.-Es un desplazamiento dentro del área ó en los saltos.

- ADMI.-Se refiere al administrador del objeto referenciado por AREA y DESPLAZAMIENTO.

- NUMARG.-Número de argumentos.

- SELE.-Posición del selector en la tabla de selectores.

- NUMPRI.-Número de rutina primitiva.

5.4 PRIMITIVAS DEL SISTEMA.

Hay ciertas clases en el sistema cuyos métodos no son ejecutados por el intérprete, o sea no estan compilados por el sistema y por lo tanto carecen de instrucciones de la Máquina Virtual. Estan codificadas en el lenguaje de implementación del sistema, en este caso Pascal. La razón de utilizar este tipo de métodos nos ofrece dos ventajas principales

- Rapidez en la ejecución de esos métodos.

- Son en donde termina la recursividad en el envio de mensajes.

El uso de primitivas entonces es necesario por un lado y deseable por el otro. La única desventaja de las clases primitivas es que el usuario no puede extender el conjunto de mensajes a que pueden responder los objetos primitivos, la única forma es modificar el código fuente de la Máquina Virtual.

Las clases que se implementaron como primitivas son aquellas cuya frecuencia de uso exigía rapidez. Estas son:

1. Números. Con dos subclases: enteros y reales.

2. Caracter.

3. Cadenas.

4. Arreglos.

5. Bloques.

6. Estructuras de control.

5.4.1 CLASES PRIMITIVAS.

A continuación se presentan las clases primitivas del sistema y los mensajes actualmente implementados.

5.4.1.1 CLASE OBJETO. -

Mensajes a la clase.

- Crea - Crea una instancia de una clase con sus campos no inicializados.
- error: - Reporta un error ocurrido en una metaclase. Se suspende la ejecución de las instrucciones que le siguen. Su argumento es una cadena.

Mensajes a las instancias.

- Eq: - Comapara si un A.O. apunta a la misma localidad que otro A.O.. Regresa TRUE si apuntan al mismo objeto y FALSE en el caso contrario.
- Ne: - Regresa TRUE si un A.O. es diferente al A.O. pasado como argumento. FALSE en el caso contrario.
- Error: Notifica un error ocurrido en una método de una instancia. Suspende la ejecución. Su argumento es una cadena.
- Hash - Regresa la dirección de memoria (entero) del A.O. receptor del mensaje.
- Longitud - Regresa la longitud del objeto receptor del mensaje.
- esNil - Regresa TRUE si el objeto apunta a NIL.

5.4.1.2 CLASE NUMEROS. -

Métodos de las Instancias

- Operaciones Aritméticas (+, -, *, /).
- Operaciones lógicas (eq:, ne:, lt:, le:, gt:, ge:).

5.4.1.3 CLASE ENTEROS. -

Métodos de las instancias.
- Mod: - Regresa el módulo de los operadores.
- Div: - Regresa la división entera de los operadores.

5.4.1.4 CLASE REALES. -

Métodos de las instancias.
- Trunc - Trunca un número real.

5.4.1.5 CLASE CADENAS. -

Métodos a las instancias
- + - Concatena dos cadenas.

5.4.1.6 CLASE BOOLEANA. -

Métodos a las instancias
- iftrue: - Ejecuta un bloque si el tope de la pila es TRUE.
- iftrue:iffalse: - Si el tope de la pila es TRUE ejecuta el primer bloque. En caso contrario ejecuta el segundo bloque.
- or: - Realiza el OR lógico.
- and: - Su argumento es un bloque. Si el tope de la pila es TRUE, ejecuta el bloque y si el valor de éste es TRUE, regresa TRUE como resultado del método. En caso contrario, regresa FALSE.

5.4.1.7 CLASE ARREGLO. -

Métodos a la clase
- Crea: - Crea un arreglo de la longitud especificada.

Métodos a las instancias.
- En: - Regresa el objeto localizado en el índice pasado como argumento.
- En:pon: - Pone un objeto en la localidad indicada.
- Todos: - Pone todos los elementos del arreglo al valor pasado como argumento.

5.4.1.8 CLASE BLOQUE -

Métodos a las instancias.

- ejecuta - Ejecuta un bloque.
- Ejecuta: - Ejecuta un bloque con un argumento.
- Ejecuta:ejecuta: - Ejecuta un bloque con dos argumentos.
- Whiletrue: - Mientras el tope de la pila sea TRUE ejecuta un bloque.

5.5 REFERENCIAS.

[1].-A. Goldberg, D.Robson; Smalltalk-80 The language and its Implementation;E.E.U.U.. Addison-Wesley Pub. (1983), Caps. 26-30.

CAPITULO 6

CONCLUSIONES

Los resultados obtenidos en el presente trabajo fueron los planteados en un principio. Estos fueron: construir un sistema en el cual un usuario pueda conocer los sistemas orientados a objetos, elaborando aplicaciones pequeñas e interactuar con ellas a través de mensajes.

El sistema SS cumple con los especificaciones generales de un Sistema Orientado a Objetos; la funcionalidad del sistema fué comprobada, con diversos ejemplos, algunos se presentan en el desarrollo del trabajo.

La interacción con el usuario esta en su forma más primitiva, puesto que consiste sólo de un medio ambiente donde los mensajes son leídos; la construcción de la clase se tiene que hacer fuera de ese medio ambiente, hecho que desperdicia las ventajas del sistema orientado a objetos, el cual es independencia entre las diferentes componentes de la clase; es decir, la definición de la clase es independiente de los métodos, y viceversa; además los métodos también son independientes entre sí. Por lo anterior, se piensa que para futuros trabajos sobre este tema se contruya un medio ambiente adecuado, lo cual, reduciría en un 50% el tamaño del compilador.

El tiempo de ejecución al enviar mensajes en el sistema deja mucho que desear, debido a que no fué uno de los objetivos elaborar una Máquina Virtual eficiente en cuanto al manejo de los objetos, sino que, se construyó de tal manera que ayudará al desarrollo del sistema, es decir que fuera fácilmente modificable (por lo que se implementó en Pascal). Una mejor implementación de la máquina podría hacerse en futuras versiones (de hecho ya se esta desarrollando actualmente en el I.I.M.A.S.), esto ayudará a intentar desarrollar aplicaciones grandes usando el sistema. Por ahora, se piensa que sólo debe utilizarse para aplicaciones pequeñas, y más que otra cosa, para introducir a los usuarios en el uso de estos sistemas, despertando así el interes de más gente

para trabajar y mejorar sistemas de éste tipo, además de que ya tienen una base firme con la cual pueden establecer puntos de comparación.

Actualmente, el sistema esta funcionando en la computadora VAX-730, propiedad del Instituto de investigaciones Eléctricas, bajo el sistema operativo VMS versión V4.1. El sistema es totalmente transportable, ya que se elaborŏ en Pascal estandard. La documentación para la instalación del sistema en otras máquinas (que tengan Pascal estandard), así como los programas fueron entregados al jefe del proyecto TM Dr. Miguel Gerzso investigador del I.I.M.A.S.. Modificaciones tendrán que hacerse para la entrada/salida.

6.1 SUGERENCIAS PARA FUTUROS TRABAJOS.

En esta sección se presentan algunos puntos que sería interesante tomar en cuenta para futuros trabajos. Se tratan tres puntos:

1. Programar en este sistema debe de ser una actividad interactiva.

2. Desarrollo de un sistemas de información.

3. Organización de las clases por categorías.

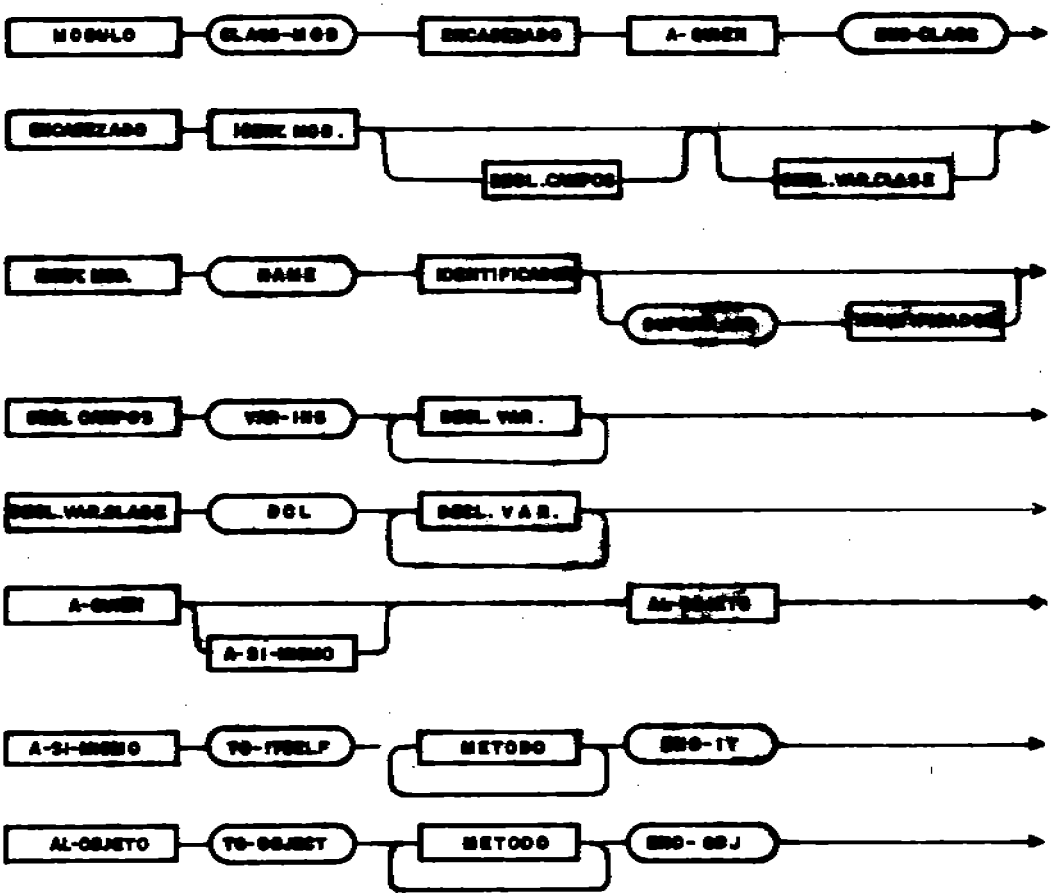
En la presente implementación, la actividad de programar no se sale mucho de los cánones, es decir, sigue el ciclo: edición del programa, compilación del mismo y por último su uso, desaprovechando las ventajas que nos proporcionan los sistemas de este tipo. Una de ellas es que un método en una clase es completamente independiente a cualquier otro método de la misma clase, por lo cual no es necesario que se compilen en un mismo módulo. La manera más fácil y mejor es la de compilar los métodos independientemente a través de un medio ambiente interactivo, en el cual se editara el método y se agregara a la clase correspondiente. Lo primero que se tendría que hacer es definir la clase, tambien interactivamente, proporcionando el nombre de ésta, la superclase asociada (si la hay), las variables de instancia y las variables de clase. El paso siguiente es agregar los métodos uno a uno. Esto evita tener que compilar todo un módulo cuando algún método sea modificado o agregado.

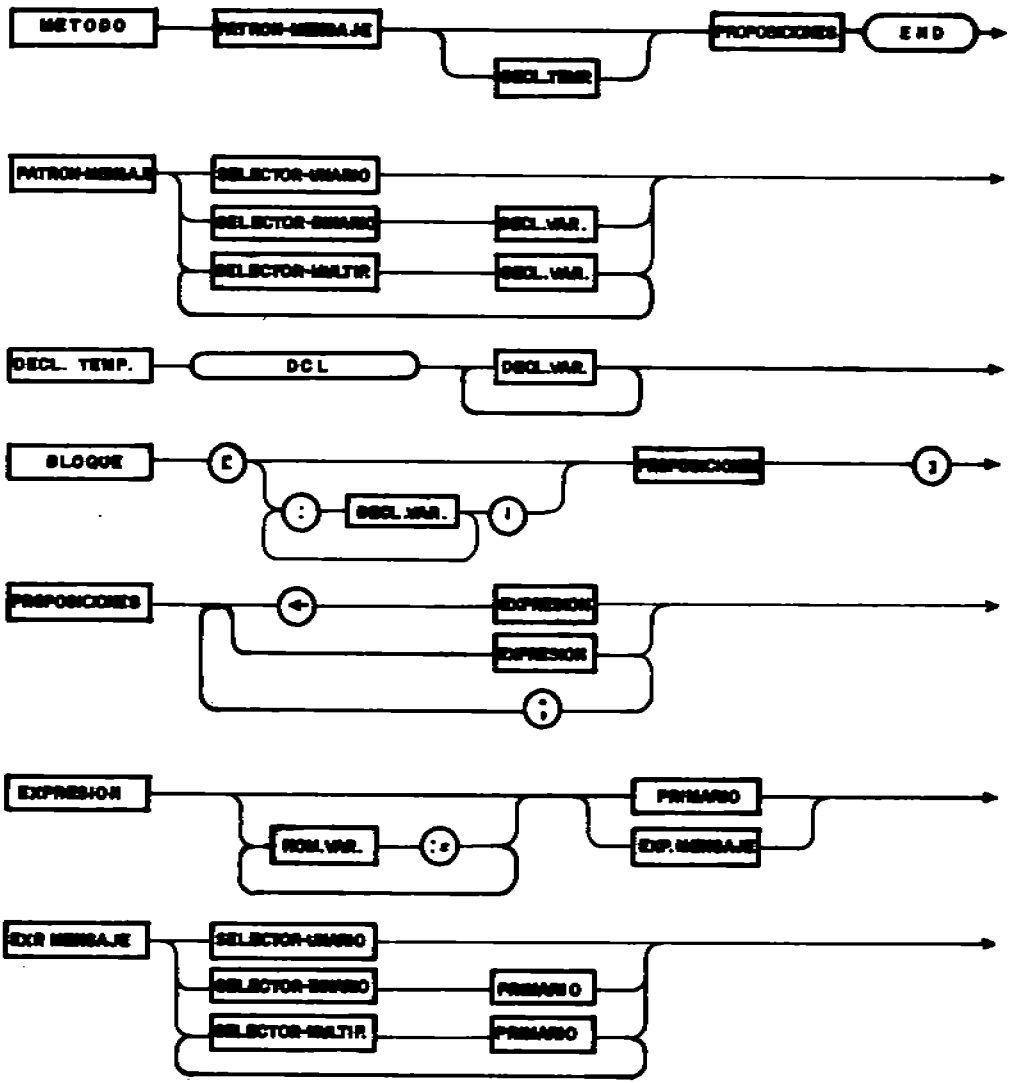
Como una consecuencia del punto anterior, se necesitaría un sistema de información en el cual el usuario podría solicitar las clases existentes en el sistema, así como de los mensajes a que pueden responder una instancia de una clase o la clase misma. También en ese sistema se almacenarían los fuentes de los métodos para posteriores accesos, modificaciones, adiciones o remplazos.

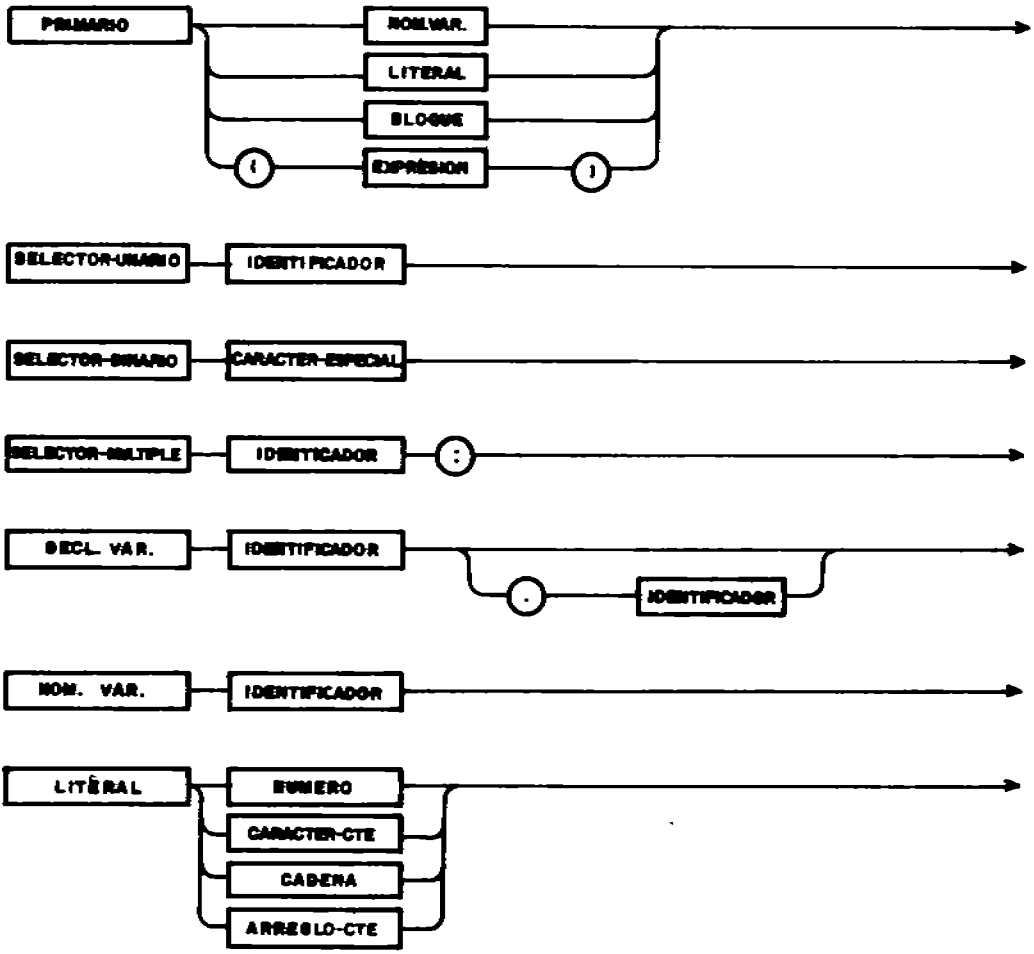
La organización de las clases por categorías sería una herramienta de mucha utilidad, tanto para el usuario como para el funcionamiento del sistema. Una categoría es una agrupación de clases encaminadas a una aplicación en particular. El usuario tendría más facilidad para el manejo de clases, ya que el sólo estará interesado en el acceso a ciertas categorías de clases. Por decir algo, si el usuario esta realizando una aplicación gráfica usaría las clases que lo ayudaran para tal fin, es decir, clases que manejen puntos, rectas, rectángulos, etc.. Por otro lado, otro usuario estaría interesado en clases que lo ayudaran para una aplicación que utilice estructuras de datos, en tal caso utilizaría las clases de la categoría de colecciones, en ésta entrarian las listas, árboles, conjuntos, etc.. Desde el punto de vista del sistema, éste daría prioridad de permanencia en memoria a las clases seleccionadas, ya que llega un momento en que es imposible mantener todas las clases del sistema en memoria.

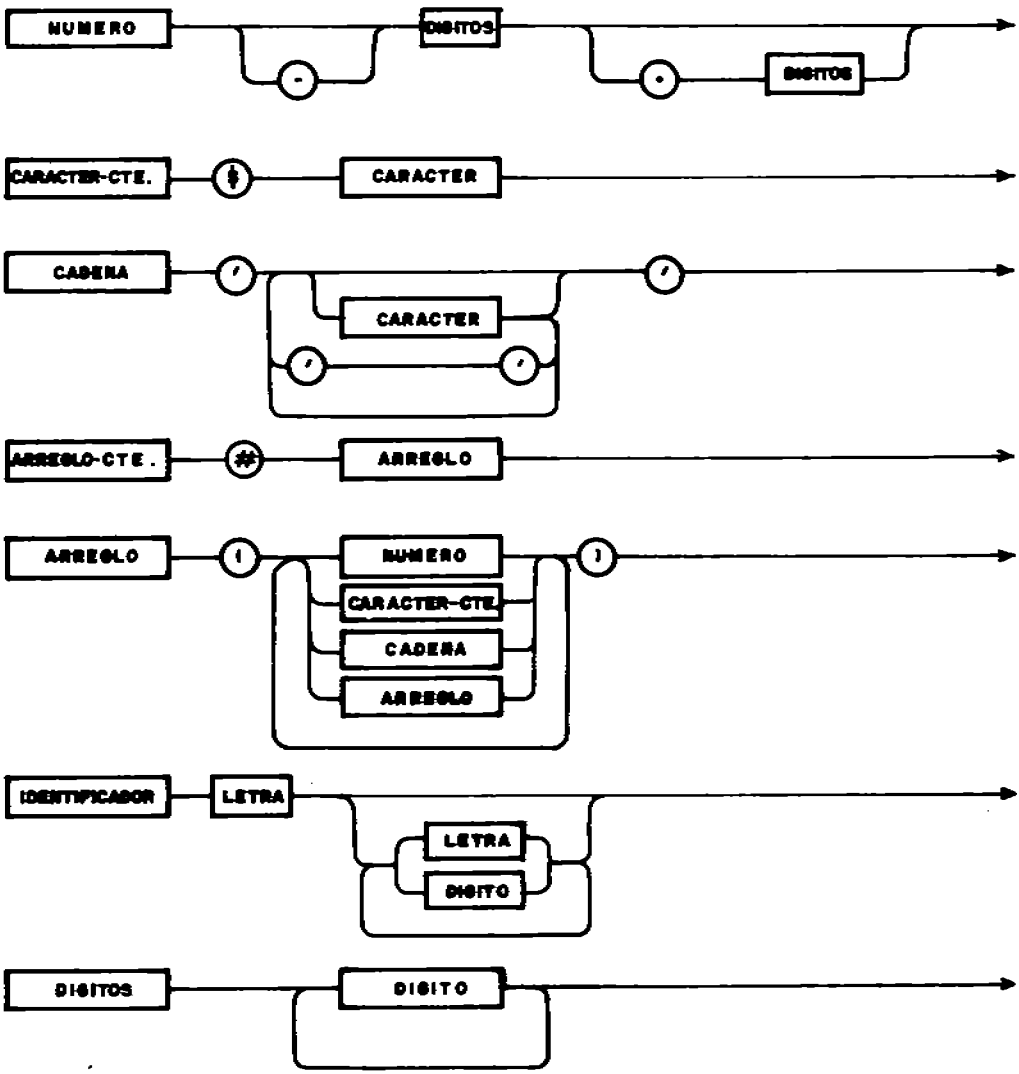
APENDICE A

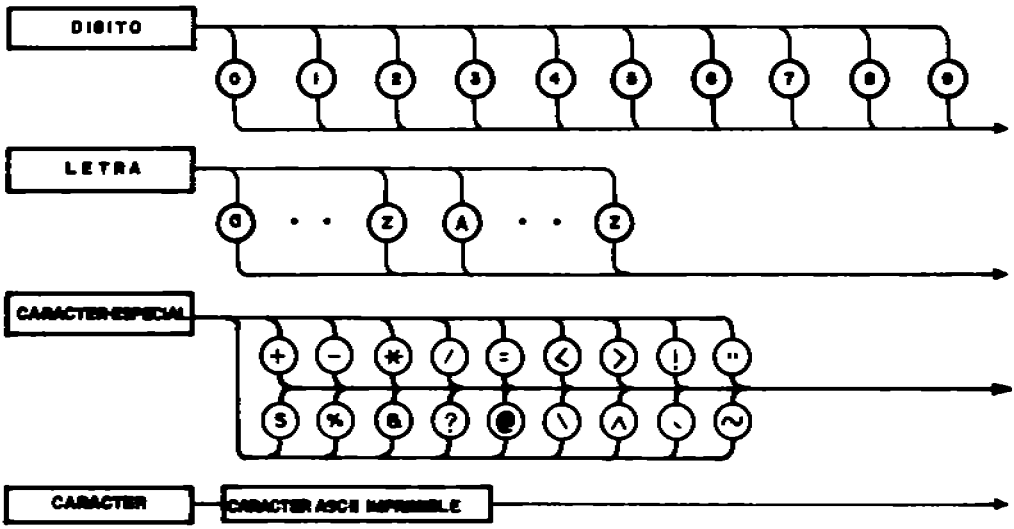
A 1 :- DIAGRAMAS DE SINTAXIS DEL LENGUAJE SS




METODO
PATRON-MENSAJE
DECL.TEMP.
PROPOSICIONES
END
SELECTOR-UNARIO
SELECTOR-BINARIO
SELECTOR-MULTIP.
DECL.VAR.
DECL.VAR.
DECL. TEMP.
DCL
DECL.VAR.
BLOQUE
[
:
DECL.VAR.
|
PROPOSICIONES
]
PROPOSICIONES
←
EXPRESION
EXPRESION
;
EXPRESION
NOM.VAR.
:=
PRIMARIO
EXP.MENSAJE
EXP.MENSAJE
SELECTOR-UNARIO
SELECTOR-BINARIO
SELECTOR-MULTIP.
PRIMARIO
PRIMARIO


PRIMARIO
NOM.VAR.
LITERAL
BLOQUE
(
EXPRESION
)
SELECTOR-UNARIO
IDENTIFICADOR
SELECTOR-BINARIO
CARACTER-ESPECIAL
SELECTOR-MULTIPLE
IDENTIFICADOR
:
SECC. VAR.
IDENTIFICADOR
.
IDENTIFICADOR
NOM. VAR.
IDENTIFICADOR
LITERAL
NUMERO
CARACTER-CTE
CADENA
ARREGLO-CTE


NUMERO
-
DIGITOS
.
DIGITOS
CARACTER-CTE.
$
CARACTER
CADENA
'
CARACTER
'
'
ARREGLO-CTE.
#
ARREGLO
ARREGLO
(
NUMERO
CARACTER-CTE
CADENA
ARREGLO
)
IDENTIFICADOR
LETRA
LETRA
DIGITO
DIGITOS
DIGITO


DIGITO
0 1 2 3 4 5 6 7 8 9
LETRA
a . . z A . . Z
CARACTER-ESPECIAL
+ - * / = < > ! "
$ % & ? @ \ ^ _ ~
CARACTER
CARACTER ASCII IMPRIMIBLE

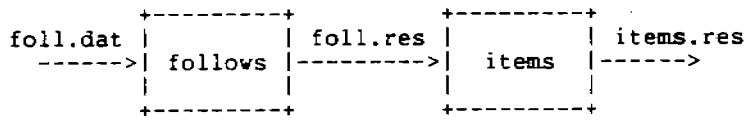
APENDICE B

B.1 GENERADOR DE TABLAS DE SINTAXIS.

Para el funcionamiento del compilador se requieren que las tablas de sintaxis esten generadas. Estas tablas se tienen que actualizar cada vez que se presente un cambio en la sintaxis del lenguaje. A continuación se muestra el procedimiento para generarlas, mencionando los archivos de datos necesarios y los programas que los procesan.

En principio, el usuario tiene que formar un archivo en donde represente la gramática del lenguaje que se quiere construir. Para lograr esto, necesita de la representación de la gramática en la notación BNF (sección B.2). Una vez que se tiene, a cada símbolo no terminal se le asocia un número consecutivo, en la sección B.2, ese número de muestra a la izquierda de cada producción) y también un número se le asocia a cada símbolo terminal (sección B.3); con estos números, se contruye un archivo como el que se presenta en la sección B.4 (la primera línea de este archivo representa el lenguaje en el cual se esta interesado).

Una vez que se tiene el archivo, los datos pasan por el siguiente proceso:



El programa follows (foll.pas), calcula los follows de los símbolos no terminales. Los follows es el conjunto de símbolos terminales que pueden aparecer inmediatamente a la derecha de un símbolo no terminal contenido a la derecha de una regla de producción. Para la definición formal de éste termino y el algoritmo para determinarlo referirse a [1].

El programa items (items.pas), utiliza el resultado que arroja el programa anterior y genera las tablas de sintáxis depositandolas en el archivo ITEMS.RES, que es leído por el compilador. Las técnicas utilizadas para la construcción del generador son las que se presentan en la referencia [2]. El autor de este programa es Salvador Barra estudiante de la Maestría en Ciencias de la Computación del Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas de la Universidad Nacional Autónoma de México.

B.2 GRAMATICA DEL LENGUAJE SS EN NOTACION BNF.

```
 1-51 <fuente>          ::= <modulo>
 2-51 <fuente>          ::= <lista-esta-ma>

 3-52 <modulo>          ::= <encabezado> <a-quien> END_CLASS

 4-53 <encabezado>      ::= <definicion>
 5-53 <encabezado>      ::= <definicion> <lista-decl>

 6-54 <definicion>      ::= <clase-mod>
 7-54 <definicion>      ::= <clase-mod> <decl-vi>

 8-55 <clase-mod>       ::= CLASS_MOD <ident-mod>

 9-56 <ident-mod>       ::= NAME <IDENTIFICADOR>
10-56 <ident-mod>       ::= NAME <IDENTIFICADOR> SUPERCLASS <IDENTIFICAD

11-57 <decl-vi>         ::= VAR_INS <decl-objeto>
12-57 <decl-vi>         ::= <decl-vi>  <decl-objeto>

13-58 <lista-decl>      ::= DCL <decl-objeto>
14-58 <lista-decl>      ::= <lista-decl> <decl-objeto>

15-59 <decl-objeto>     ::= <IDENTIFICADOR>
16-59 <decl-objeto>     ::= <IDENTIFICADOR> , <IDENTIFICADOR>

17-60 <bloque>          ::= <par-izq> <estatutos> ]

18-61 <par-izq>         ::= [
19-61 <par-izq>         ::= [ <lista-arg> |

20-62 <lista-arg>       ::= : <decl-objeto>
21-62 <lista-arg>       ::= <lista-arg> : <decl-objeto>

22-63 <a-quien>         ::= <a-clase>
23-63 <a-quien>         ::= <a-instancia>
24-63 <a-quien>         ::= <a-clase> <a-instancia>

25-64 <a-clase>         ::= TO_ITSELF <lista-metodo> END_IT

26-65 <a-instancia>     ::= TO_OBJECT <lista-metodo> END_IT

27-66 <lista-metodo>    ::= <metodo>
28-66 <lista-metodo>    ::= <lista-metodo> <metodo>

29-67 <metodo>          ::= <patron-men> => <cpo-metodo> END

30-68 <patron-men>      ::= <IDENTIFICADOR>
31-68 <patron-men>      ::= <CARACTER-ESP> <decl-objeto>
32-68 <patron-men>      ::= <patron-mult>
```

```
33-69 <patron-mult>    ::= <sub-sel> <decl-objeto>
34-69 <patron-mult>    ::= <patron-mult> <sub-sel> <decl-objeto>

35-70 <cpo-metodo>     ::= <estatutos>
36-70 <cpo-metodo>     ::= <decl-temp> ; <estatutos>
37-70 <cpo-metodo>     ::= <primitiva> ; <estatutos>

38-71 <decl-temp>      ::= <lista-decl>

39-72 <estatutos>      ::= <lista-esta>
40-72 <estatutos>      ::= <exp-retorno>
41-72 <estatutos>      ::= <lista-esta> ; <exp-retorno>

42-73 <lista-esta>     ::= <estatuto>
43-73 <lista-esta>     ::= <lista-esta> ; <estatuto>

44-74 <estatuto>       ::= <expresion>
45-74 <estatuto>       ::= <exp-asig>

46-75 <expresion>      ::= <primario>
47-75 <expresion>      ::= <exp-mensaje>

48-76 <primario>       ::= <nom.var>
49-76 <primario>       ::= <literal>
50-76 <primario>       ::= <bloque>
51-76 <primario>       ::= ( <estatuto> )

52-77 <exp-mensaje>    ::= <expresion> <= <mensaje>
53-77 <exp-mensaje>    ::= <expresion> <coma> <mensaje>

54-78 <coma>           ::= ,

55-79 <nom.var>        ::= <IDENTIFICADOR>
56-79 <nom.var>        ::= . <IDENTIFICADOR>

57-80 <literal>        ::= <LITERAL-PRIM>

58-81 <mensaje>        ::= <sel.unario>
59-81 <mensaje>        ::= <sel.binario> <primario>
60-81 <mensaje>        ::= <mensaje-mult>

61-82 <sel.unario>     ::= <IDENTIFICADOR>

62-83 <sel.binario>    ::= <CARACTER-ESP>

63-84 <mensaje-mult>   ::= <sub-sel> <primario>
64-84 <mult-massage>   ::= <mensaje-mult> <sub-sel> <primario>

65-85 <sub-sel>        ::= <IDENTIFICADOR> :

66-86 <exp-asig>       ::= <list-val-izq> := <expresion>
```

```
67-87 <list-val-izq>   ::= <nom.var>
68-87 <list-val-izq>   ::= <list-val-izq> := <nom.var>

69-88 <exp-retorno>    ::= <- <estatuto>

70-89 <lista-esta-ma> ::= <esta-ma>
71-89 <lista-esta-ma> ::= <lista-esta-ma> ; <esta-ma>

72-90 <esta-ma>        ::= <estatuto>
73-90 <esta-ma>        ::= <temp-decl>

74-91 <primitiva>      ::= PRIMITIVE <LITERAL-PRIM>
```

B.3 SIMBOLOS TERMINALES DEL LENGUAJE SS.

```
 1  - (
 2  - )
 3  - ,
 4  - ;
 5  - [
 6  - ]
 7  - :=
 8  - <=
 9  - =>
10  - <-
11  - .
12  - :
13  - |
14  - <CARACTER-ESP>
15  - <IDENTIFICADOR>
16  - <LITERAL-PRIM>
17  - CLASS_MOD
18  - END_CLASS
19  - NAME
20  - SUPERCLASS
21  - VAR_INS
22  - DCL
23  - TO_ITSELF
24  - END_IT
25  - TO_OBJECT
26  - END_OBJ
27  - END
28  - PRIMITIVE
29  - SELF
30  - SUPER
31  - NIL
32  - TRUE
33  - FALSE
```

B.4 ARCHIVO DE DATOS PARA EL GENERADOR.

```
50 51 -1 -1 -1 -1
51 52 -1 -1 -1 -1
51 89 -1 -1 -1 -1
52 53 63 18 -1 -1
53 54 -1 -1 -1 -1
53 54 58 -1 -1 -1
54 55 -1 -1 -1 -1
54 55 57 -1 -1 -1
55 17 56 -1 -1 -1
56 19 15 -1 -1 -1
56 19 15 20 15 -1
57 21 59 -1 -1 -1
57 57 59 -1 -1 -1
58 22 59 -1 -1 -1
58 58 59 -1 -1 -1
59 15 -1 -1 -1 -1
59 15 11 15 -1 -1
60 61 72  6 -1 -1
61  5 -1 -1 -1 -1
61  5 62 13 -1 -1
62 12 59 -1 -1 -1
62 62 12 59 -1 -1
63 64 -1 -1 -1 -1
63 65 -1 -1 -1 -1
63 64 65 -1 -1 -1
64 23 66 24 -1 -1
65 25 66 26 -1 -1
66 67 -1 -1 -1 -1
66 66 67 -1 -1 -1
67 68  9 70 27 -1
68 15 -1 -1 -1 -1
68 14 59 -1 -1 -1
68 69 -1 -1 -1 -1
69 85 59 -1 -1 -1
69 69 85 59 -1 -1
70 72 -1 -1 -1 -1
70 71  4 72 -1 -1
70 91  4 72 -1 -1
71 58 -1 -1 -1 -1
72 73 -1 -1 -1 -1
72 88 -1 -1 -1 -1
72 73  4 88 -1 -1
73 74 -1 -1 -1 -1
73 73  4 74 -1 -1
74 75 -1 -1 -1 -1
74 86 -1 -1 -1 -1
75 76 -1 -1 -1 -1
75 77 -1 -1 -1 -1
76 79 -1 -1 -1 -1
```

```
76 80 -1 -1 -1 -1
76 60 -1 -1 -1 -1
76  1 74  2 -1 -1
77 75  8 81 -1 -1
77 75 78 81 -1 -1
78  3 -1 -1 -1 -1
79 15 -1 -1 -1 -1
79 11 15 -1 -1 -1
80 16 -1 -1 -1 -1
81 82 -1 -1 -1 -1
81 83 76 -1 -1 -1
81 84 -1 -1 -1 -1
82 15 -1 -1 -1 -1
83 14 -1 -1 -1 -1
84 85 76 -1 -1 -1
84 84 85 76 -1 -1
85 15 12 -1 -1 -1
86 87  7 75 -1 -1
87 79 -1 -1 -1 -1
87 87  7 79 -1 -1
88 10 74 -1 -1 -1
89 90 -1 -1 -1 -1
89 89  4 90 -1 -1
90 74 -1 -1 -1 -1
90 71 -1 -1 -1 -1
91 28 16 -1 -1 -1
```

B.5 REFERENCIAS.

[1].- Alfred V. Aho, Jeffrey D. Ullman; Principles of Compiler Design; E.E.U.U., Addison-Wesley (1977), Cap. 5 pp 186-189.

[2].- Alfred V. Aho, Jeffrey D. Ullman; Principles of Compiler Design; E.E.U.U., Addison-Wesley (1977), Cap. 6.