Tablas de símbolos

Book · May 2004

CITATIONS
0

READS
6,016

4 authors, including:

Aquilino Adolfo Juan
University of Oviedo
41 PUBLICATIONS **115** CITATIONS
SEE PROFILE

Francisco Ortin
University of Oviedo
131 PUBLICATIONS **918** CITATIONS
SEE PROFILE

Raúl Izquierdo Castanedo
University of Oviedo
22 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Universidad de Oviedo

Departamento de Informática

LENGUAJES Y SISTEMAS INFORMÁTICOS

TABLAS DE SÍMBOLOS

AUTORES

Aquilino Adolfo Juan Fuente
Juan Manuel Cueva Lovelle
Francisco Ortín Soler
Raúl Izquierdo Castanedo
María Cándida Luengo Díez
José Emilio Labra Gayo

Marzo DE 2006

INDICE DE MATERIAS

1 Introducción

1.1 Necesidad de las Tablas de símbolos

La fase de análisis semántico obtiene su nombre por requerir información relativa al significado del lenguaje, que está fuera del alcance de la representatividad de las gramáticas libres de contexto y los principales algoritmos existentes de análisis; es por ello por lo que se dice que captura la parte de la fase de análisis considerada fuera del ámbito de la sintaxis. Dentro del la clasificación jerárquica que Chomsky dio de los lenguajes [HOPCR02, CUEV03], la utilización de gramáticas sensibles al contexto (o de tipo 1) permitirían identificar sintácticamente características como que la utilización de una variable en el lenguaje Pascal ha de estar previamente declarada. Sin embargo, la implementación de un analizador sintáctico basado en una gramática de estas características sería computacionalmente más compleja que un autómata de pila [LOUDE97].

Así, la mayoría de los compiladores utilizan una gramática libre de contexto para describir la sintaxis del lenguaje y una fase de análisis semántico posterior para restringir las sentencias que "semánticamente" no pertenecen al lenguaje. En el caso que mencionábamos del empleo de una variable en Pascal que necesariamente haya tenido que ser declarada, el analizador sintáctico se limita a comprobar, mediante una gramática libre de contexto, que un identificador forma parte de una expresión. Una vez comprobado que la sentencia es sintácticamente correcta, el analizador semántico deberá verificar que el identificador empleado como parte de una expresión haya sido declarado previamente. Para llevar a cabo esta tarea, es típica la utilización de una estructura de datos adicional denominada tabla de símbolos. Ésta poseerá una entrada por cada identificador declarado en el contexto que se esté analizando. Con este tipo de estructuras de datos adicionales, los desarrolladores de compiladores acostumbran a suplir las carencias de las gramáticas libres de contexto.

1.2 Objetivos de la Tabla de Símbolos (TS)

Las Tablas de Símbolos (en adelante TS) son estructuras de datos que almacenan toda la información de los identificadores del lenguaje fuente.

Las misiones principales de la TS en el proceso de traducción son:

- Colaborar con las comprobaciones semánticas.
- Facilitar ayuda a la generación de código.

La información almacenada en la TS depende directamente del tipo de elementos del lenguaje específico a procesar y de las características de dicho lenguaje. Habitualmente los elementos del lenguaje que requieren el uso de la TS son los distintos tipos de identificadores del lenguaje (nombres de variables, de objetos, de funciones, de etiquetas, de clases, de métodos, etc.).

La información relativa a un elemento del lenguaje se almacena en los denominados *atributos* de dicho elemento. Estos atributos también varían de un tipo de lenguaje a otro y de un elemento a otro. Así ejemplos de atributos tales como nombre, tipo, dirección relativa en tiempo de ejecución, dimensiones de los arrays, número y tipo de los parámetros de procedimientos, funciones y métodos, tipos de acceso a los elementos de una clase (*public*, *private*, *protected*…), etc. se recogen y se guardan en la TS.

Los atributos se obtienen unas veces directamente del análisis del programa fuente, es decir, están en forma explícita (por ejemplo en la sección de declaraciones del programa fuente) y otras veces los atributos se obtienen de forma implícita a través del contexto en el que aparece el elemento en el programa fuente.

En el proceso de compilación se accede a la TS en unos determinados puntos que dependen inicialmente del número y la naturaleza de las pasadas del procesador de lenguaje y del propio lenguaje fuente a procesar.

En los traductores y compiladores, las TS existen únicamente en tiempo de compilación, aunque en depuración (*debug*) pueden estar almacenadas en disco y dar información en tiempo de ejecución para identificar los símbolos que se deseen inspeccionar.

En los intérpretes contienen información en tiempo de ejecución.

Las palabras reservadas no están en la TS.

1.3 Compiladores de una y de varias pasadas

1.3.1 Compiladores de varias pasadas

En un compilador de varias pasadas, tal como el de la Figura 1-1, la TS se crea durante el análisis léxico y sintáctico (pasada 1). En los compiladores modernos la TS se crea durante el primer recorrido del árbol AST, una vez creado éste mediante al analizador sintáctico (pasada 2). Cuando un compilador comienza a traducir un programa fuente, la TS está vacía o contiene unas pocas entradas para las funciones incorporadas (las palabras reservadas habitualmente son almacenadas en una tabla aparte y usadas exclusivamente por el analizador léxico).

El analizador léxico separa el programa fuente en tokens que compara con las palabras reservadas. Si el token comparado no es una palabra reservada se asume que es un identificador y durante el sintáctico (fase sintáctica) se añade a ella. Si el token comparado se encuentra en la TS, el analizador sintáctico accede directamente al índice que identifica a dicho identificador reconocido anteriormente, en caso contrario, el analizador sintáctico introduce dicho símbolo en la TS.

Según avanza la compilación, sólo se añade una entrada para cada identificador nuevo, pero se explora la Tabla una vez por cada nueva aparición de un identificador.

Por ejemplo en la Figura 1-1, X e Y ocupan las posiciones 1 y 2 respectivamente en la TS. El analizador sintáctico (fase sintáctica de la pasada 1) recibe la cadena de tokens, comprueba las especificaciones sintácticas del lenguaje y construye el árbol sintáctico (*Abstract Sintax Tree* ó AST), almacenando de alguna forma, habitualmente en los lenguajes orientados a objetos se usa el patrón *composite*.

Este código se comprueba por el analizador semántico y se usa en la fase de generación de código para la implementación de las instrucciones en código objeto.

Las hojas del árbol contienen los índices de la TS correspondientes a los identificadores detectados.

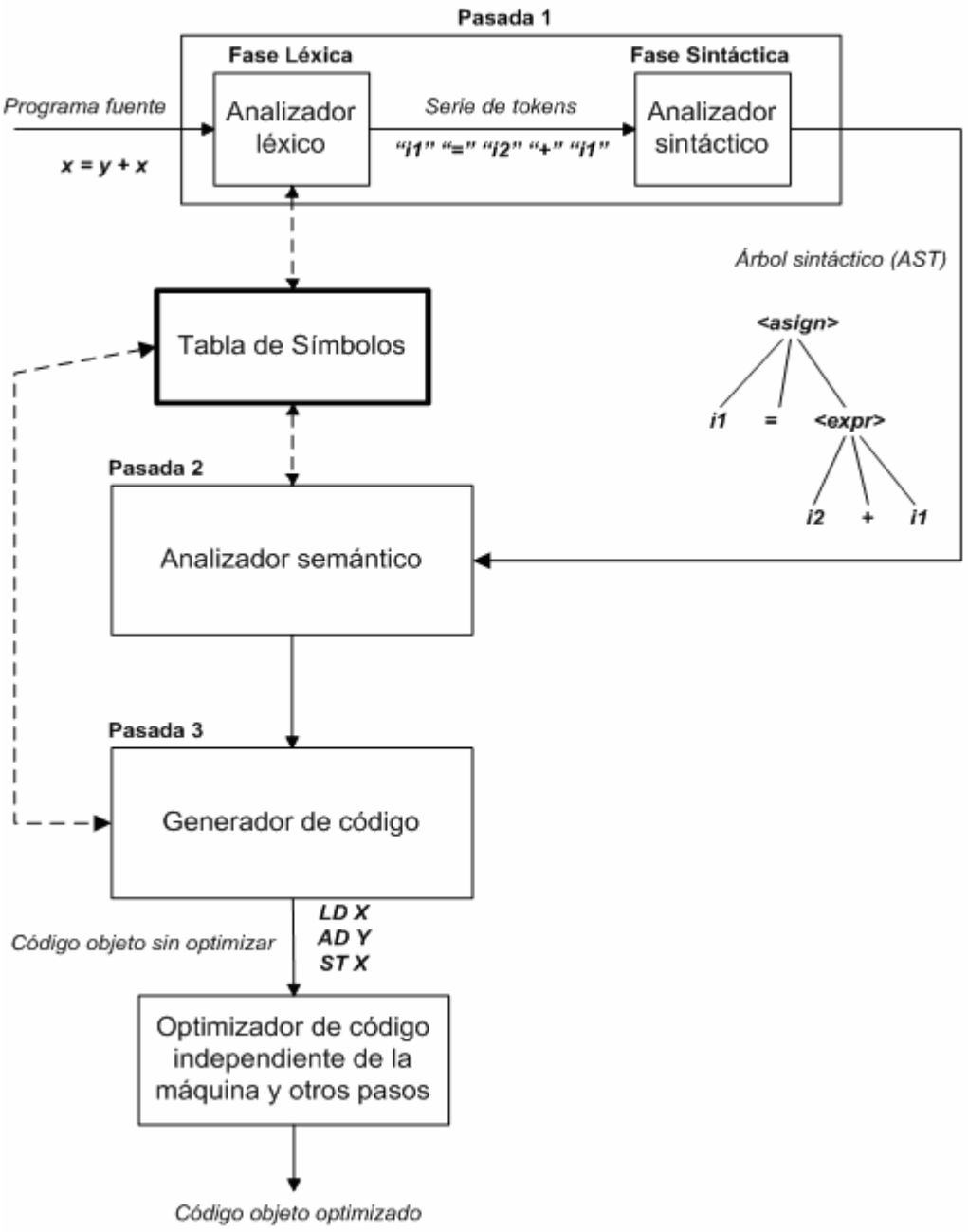


Figura 1-1. Compilador de varias pasadas

Durante la fase de análisis sintáctico no se usan procedimientos que manejen la TS, excepto que sean necesarias comprobaciones semánticas para resolver ambigüedades sintácticas.

No es hasta las fases de análisis semántico y de generación de código cuando vuelve a utilizarse la TS, pues en estas fases ocurre que alguno de los atributos asociados a un identificador se les pueda asignar un valor en la TS.

Por ejemplo, en un lenguaje con declaraciones explícitas, el tipo de una variable sólo será asignado a la TS cuando la variable se reconoce en su declaración.

Se puede intentar la asignación de atributos a la TS en otros puntos del proceso de traducción diferentes a la fase de análisis léxico. Esto nos obligará a realizar modificaciones en los analizadores sintáctico y semántico que producirían un compilador muy fragmentado en el sentido de poca optimización y estructuración, pues

las funciones del analizador léxico, sintáctico y semántico estarían mezcladas en distintos módulos del procesador de lenguaje.

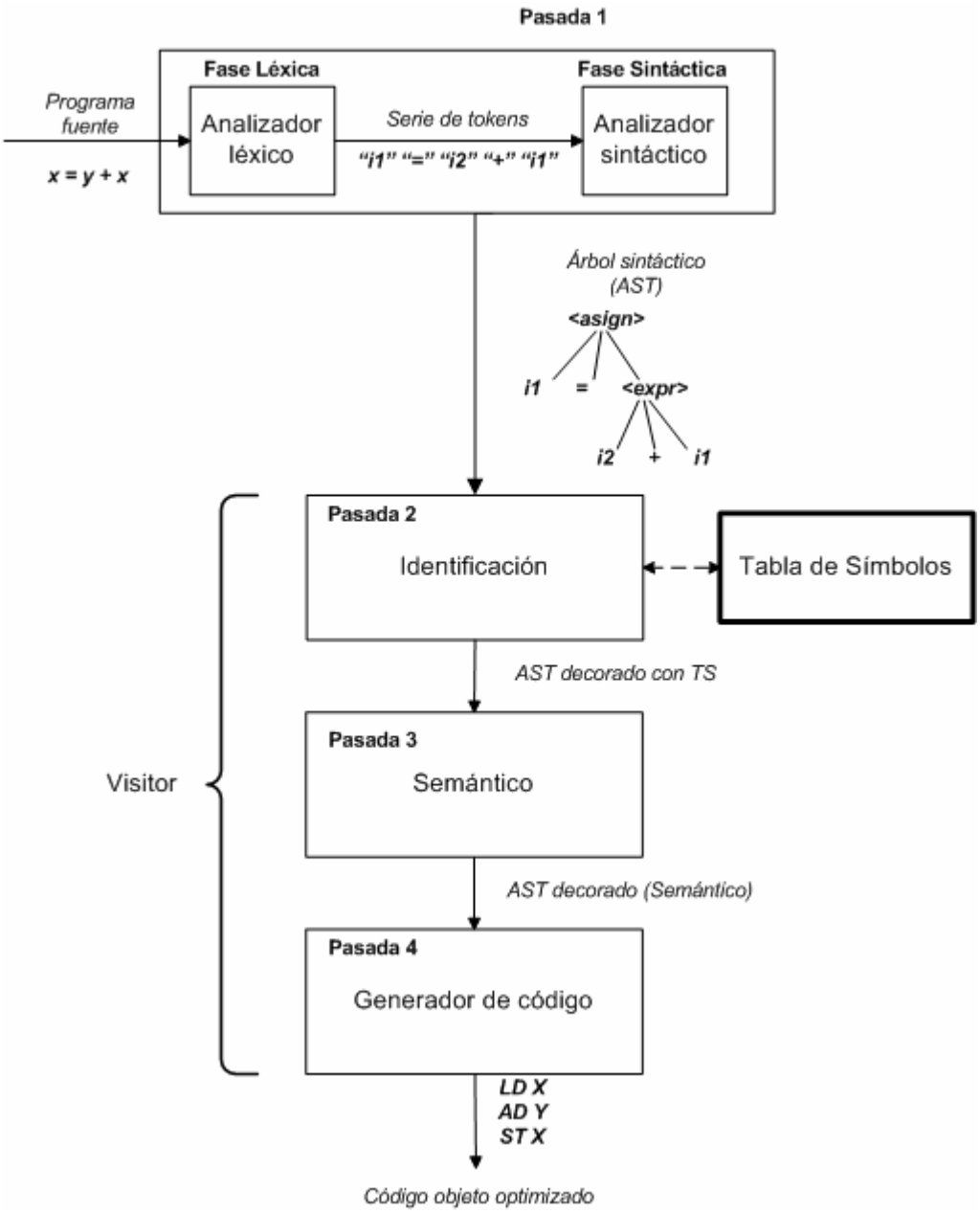


Figura 1-2. Tabla de símbolos en un compilador de varias pasadas con generación de AST

Cuando se utiliza una representación interna de un AST, tanto las declaraciones como las instrucciones quedan representadas por nodos de dicho árbol. En este caso es posible una primera pasada de identificación en la cual se crea una TS que sólo se una para decorar el árbol.

Una vez decorado el árbol con la información de los identificadores y tipos de usuario (mediante referencias cruzadas entre los nodos del AST) la TS ya no es necesaria, ya que toda la información necesaria (que estaba en el sintáctico) ha sido ahora referenciada adecuadamente señalando desde los lugares dónde se usa a los contextos en los que se creó.

1.3.2 Compiladores de una pasada

Una segunda aproximación al manejo de las TS se presenta en la Figura 1-3, donde el análisis léxico, sintáctico, semántico y la generación de código se realizan en una pasada, es decir, se explora el texto fuente sentencia a sentencia (o bloques de sentencias) realizándose los tres análisis y la generación de código.

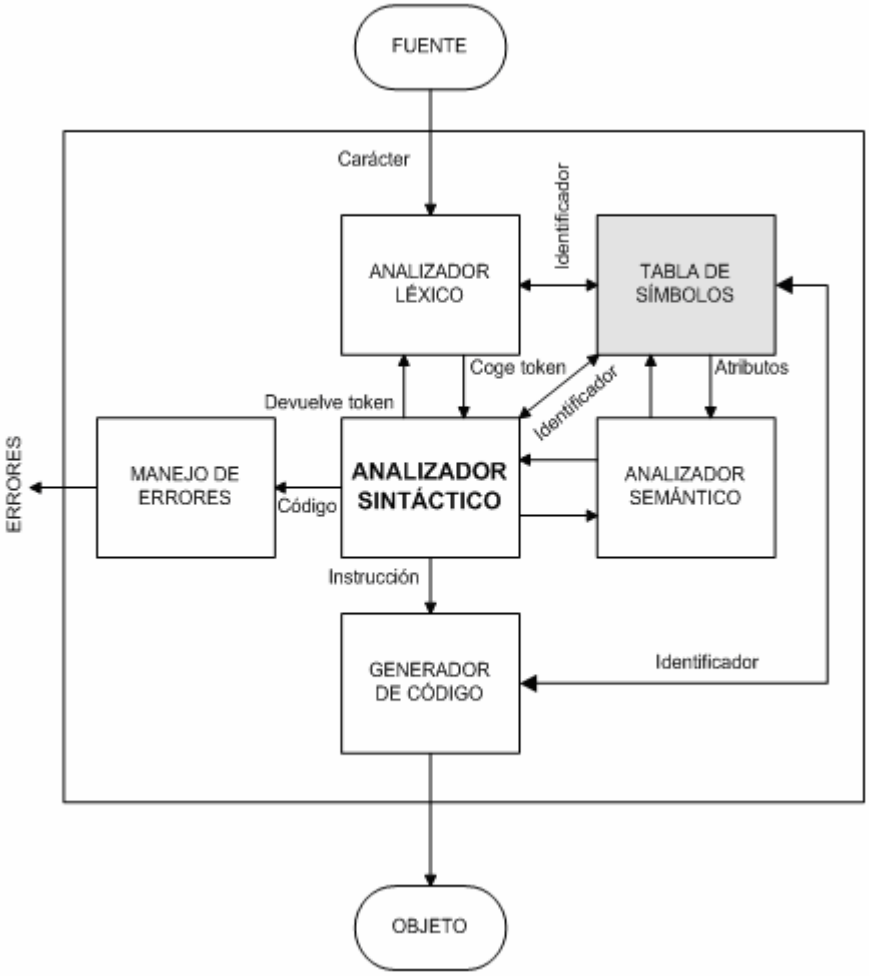


Figura 1-3. Compilador de una pasada.

En este caso puede suceder que una declaración de una variable sea procesada por el generador de código antes de que se acabe de explorar el texto fuente. Esto puede ser de gran ayuda, puesto que toda variable detectada después de su declaración permite que sean colocados sus atributos en la TS por el generador de código.

En este compilador de una pasada la TS sólo está conectada al análisis sintáctico y a través suyo al resto de los módulos.

Una excepción de lo anterior ocurre cuando el analizador sintáctico requiere cierta información del analizador léxico durante el proceso de compilación. Por ejemplo, en algunos lenguajes de programación es conveniente reconocer por medio de una tabla en el analizador léxico el tipo de un identificador particular. Con esta información el analizador sintáctico recibe un token con un significado adicional, tan como "identificador real" o "identificador entero", que es mejor que pasar "identificador".

Esta estrategia tiene dos ventajas:

a) Reduce la complejidad de la gramática para análisis sintáctico (por ejemplo, se pueden evitar ciertos problemas que se producen al utilizar una construcción sintáctica general tal como "identificador").

b) Permite una mejor especificación de los errores sintácticos por el compilador gracias a la utilización de construcciones menos generales.

2 Contenidos de la TS

Una TS se puede definir como una estructura de datos organizada en función de los identificadores que aparecen en el programa fuente.

Aunque su nombre parece indicar una estructuración en una tabla no es necesariamente ésta la única estructura de datos utilizada, también se emplean árboles, pilas, etc.

Lo que la estructura debe permitir es establecer un homomorfismo entre los ámbitos de utilización de los símbolos en el programa fuente y el modo en que aparecen en las sucesivas búsquedas en la tabla. Para ello debe manejar diferentes contextos de búsqueda que imiten los diferentes tipos de bloques del lenguaje fuente que se compila.

Los símbolos se guardan en la tabla con su nombre y una serie de atributos opcionales que dependerán del lenguaje y de los objetivos del procesador. Este conjunto de atributos almacenados en la TS para un símbolo determinado se define como **registro de la tabla de símbolos** (*symbol-table record*).

IDENTIFICADOR	DIRECCIÓN	TIPO	DIMENSIÓN	OTROS ATRIBUTOS
companyia	STATIC+0	C	10	...
x3	STATIC+10	I	0	...
forma1	STATIC+12	B	0	...
b	STATIC+13	F	3	...

Figura 2-1. Tabla de Símbolos sencilla.

Una forma de organización simple es imaginar la TS como una tabla con una serie de filas, cada fila contiene una lista de atributos que están asociados a un identificador, tal como se muestra en la Figura 2-1.

Las clases de atributos que aparecen en una TS dependen de la naturaleza del lenguaje de programación para el cual está escrito el compilador. Por ejemplo, un lenguaje de programación puede no tener tipos, entonces el atributo tipo no necesita aparecer en la tabla.

La organización de la TS variará según las limitaciones de memoria y tiempo de acceso donde se implemente el compilador.

La lista siguiente de atributos no es necesaria para todos los compiladores, sin embargo cada uno de ellos se puede utilizar en la implementación de un compilador particular.

- Nombre de identificador.
- Dirección en tiempo de ejecución a partir de la cual se almacenará el identificador si es una variable. En el caso de funciones puede ser la dirección a partir de la cual se colocará el código de la función.

- Tipo del identificador. Si es una función, es el tipo que devuelve la función.
- Número de dimensiones del array, o número de miembros de una estructura o clase, o número de parámetros si se trata de una función.
- Tamaño máximo o rango de cada una de las dimensiones de los arrays, si tienen dimensión estática.
- Tipo y forma de acceso de cada uno de los miembros de las estructuras, uniones o clases. Tipo de cada uno de los parámetros de las funciones o procedimientos.
- Valor del descriptor del fichero y tipo de los elementos del fichero en el caso de lenguajes basados en ficheros homogéneos.
- Número de la línea del texto fuente en que la variable está declarada.
- Número de la línea del texto fuente en que se hace referencia a la variable.
- Campo puntero para construir una lista encadenada que permita listar las variables en orden alfabético en las fases de depuración de código.

2.1 Nombre del identificador

Los nombres de los identificadores deben estar siempre asociados[1] en la TS, pues así son localizados por el analizador semántico y por el generador de código.

El primer problema en la organización de la TS es la variación en la longitud de los nombres de los identificadores. En las primeras versiones de los lenguajes de los años sesenta tales como el BASIC y el FORTRAN, los identificadores tenían como máximo seis caracteres significativos, el problema era mínimo y podía almacenarse el identificador completo en una longitud de campo con un tamaño fijado de antemano. Sin embargo las normas ANSI e ISO de los lenguajes C, C++ y PASCAL permiten un mínimo de 31 caracteres. Si se reservase un espacio fijo de 31 caracteres, las TS que utilicen esta forma de almacenamiento gestionarían la memoria de una forma poco eficiente, aunque el acceso a las tablas es rápido. La poca eficiencia se debe a los huecos dejados por los identificadores con nombres cortos.

La manera en que se implementará el nombre dependerá del lenguaje de programación en que se implemente la propia TS. En los lenguajes como C y C++ se puede utilizar un campo del tipo puntero a carater (char *) y reservar la memoria dinámica necesaria en cada caso. También en lenguajes como C++, Java, C#, etc. se puede utilizar directamente el tipo *String* (o equivalente) de la propia biblioteca del lenguaje.

Otra solución para almacenar los nombres de las variables es colocar un descriptor de cadenas de caracteres (*strings*) en el campo del nombre del identificador. El descriptor contiene la posición y longitud de los subcampos del *string* donde se encuentra el nombre del identificador tal y como se muestra en la Figura 2-2, esta forma de acceso a los identificadores es más lenta pero puede ahorrar bastante almacenamiento.

[1] Asociados indica que pueden ser simplemente elementos clave de una estructura de diccionario o de tabla hash, no necesariamente un atributo de la estructura símbolo que se guarda.

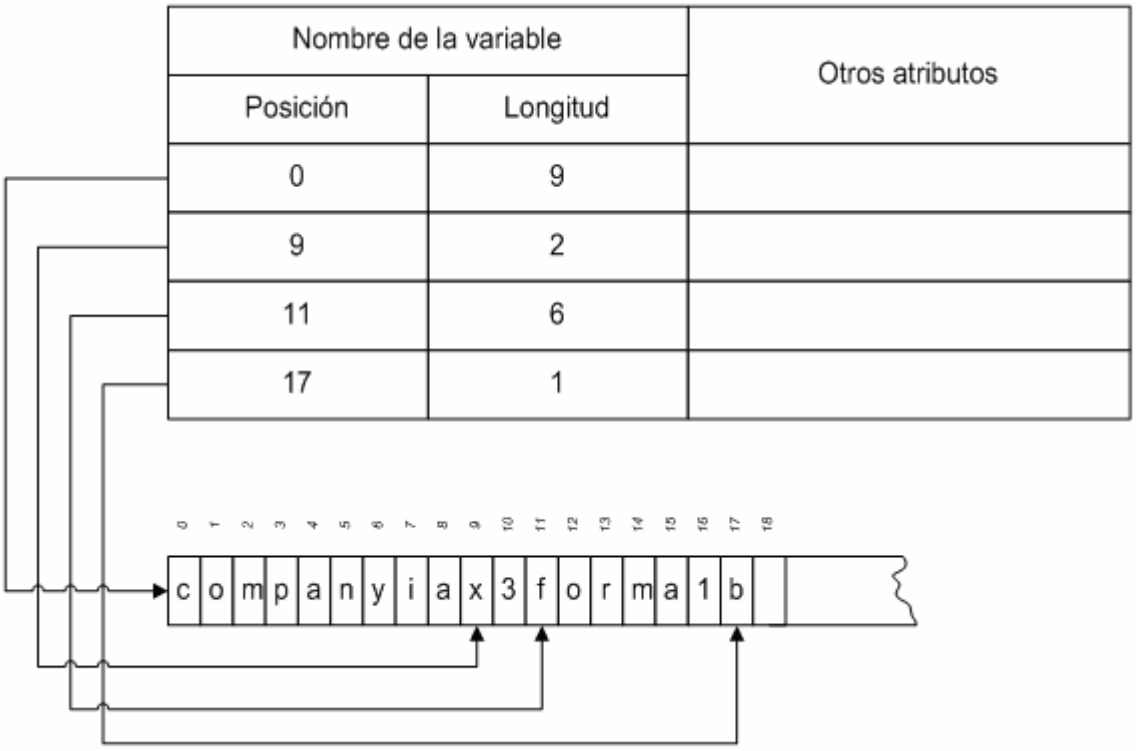


Figura 2-2. Descriptor de nombres de identificadores (strings)

2.2 Atributos de los identificadores

Los identificadores se describen por medio de los atributos que dependerán del lenguaje que se esté compilando. Algunos de estos atributos se describen en los siguientes párrafos.

2.2.1 Dirección en memoria (offset)

Los lenguajes de alto nivel tienen identificadores, sin embargo en código máquina no existen identificadores, tan solo hay las direcciones donde están colocados.

Si el código objeto que genera el compilador es de muy bajo nivel se tiene que asociar en todo momento a cada identificador su dirección de comienzo. En algunos casos puede que el código objeto sea a nivel de ensamblador, en dicho caso pueden no hacer falta direcciones dado que en el ensamblador se pueden utilizar identificadores.

La TS ayuda al generador de código a generar el código objeto, sustituyendo los identificadores por sus direcciones. Las direcciones suelen ser relativas, es decir desplazamientos (*offsets*) desde una dirección base.

Así en la Figura 2-1 se han colocado STATIC+X, señalando STATIC a la dirección de comienzo de los identificadores del segmento estático (habitualmente constantes y variables globales).

El montador de enlaces (*linker*) es el encargado de pasar direcciones relativas a absolutas[2] [CUEVA95c].

[2] Sobre la organización de la memoria en tiempo de ejecución consultar el capítulo de Gestión de Memoria.

2.2.2 Tipo

El atributo tipo se almacena en la TS cuando los lenguajes a compilar tienen distintos tipos de datos definidos explícita o implícitamente.

Por supuesto, los lenguajes sin tipos no tienen este atributo. Un ejemplo de tipos definidos implícitamente se da en el lenguaje FORTRAN, pues si no se asignan o se declaran previamente, todas las variables cuyo nombre comienza con I, J, K, L ó M son variables enteras. Todas las demás son reales.

El tipo de la variable se utiliza en las comprobaciones semánticas de las sentencias. El tipo también se usa como indicación de la cantidad de memoria que debe ser reservada en tiempo de ejecución. Por ejemplo, si el tipo es *integer*, suele ocupar la mitad de un *float*. Generalmente, el tipo de una variable se almacena en forma de código, así el tipo de *float* se puede codificar como *F*, *integer* como *I*, carácter como *C*, etc.

El tamaño de los tipos de datos dependerá de cada implementación del lenguaje, aunque el constructor del compilador suele aprovechar al máximo las características de máximo rendimiento de la máquina objeto.

2.2.3 Número de dimensiones, de miembros o de parámetros

Los atributos número de dimensiones, número de miembros y número de parámetros son importantes para la verificación semántica.

El número que indica la dimensión de un array también se utiliza como parámetro en la fórmula general de cálculo de la dirección de un elemento particular del array [CUEV95c].

El número de parámetros en la llamada a un procedimiento o función debe coincidir con el número que aparece en la declaración del procedimiento o función.

Dependiendo del lenguaje en que se implemente la TS, puede ser conveniente combinar los dos atributos anteriores en uno, ya que la verificación semántica de ambos es similar.

En la Figura 2-1 se ha considerado que la dimensión de los identificadores simples es 0, de los arrays unidimensionales 1, de los arrays bidimensionales 2, etc…

2.2.4 Valor máximo de las dimensiones o rangos de arrays

En la TS debe almacenarse el valor máximo que puede alcanzar un array, así cuando se declara una array en C o en C++, el rango de valores comienza en cero, pero debe almacenarse el número de elementos del array:

```
int vector [10], matriz[20][30];
```

el valor 10 (número de elementos del array) o 9 (último subíndice permitido) debe almacenarse como atributo del vector, dado que es necesario para calcular la posición de los elementos `vector[i]` en la generación de código. Lo mismo se puede decir de los valores 20 y 30 del array bidimensional `matriz`.

En lenguajes como PASCAL se permite definir un rango de valores entre los cuales varían los subíndices de los arrays, así por ejemplo:

```
VAR vector : ARRAY [10..20] OF INTEGER;
    matriz : ARRAY [-10..10, -25..100] OF INTEGER;
```

En estos casos es necesario almacenar el valor inferior y superior del rango.

Si el lenguaje de programación no tiene un valor máximo de dimensiones de los arrays, es necesario definir una lista dinámica con los valores máximos o rangos de cada dimensión del array.

Una solución más rápida, pero mucho más limitada, es definir un tamaño máximo de dimensiones de los arrays y dejar un campo de tamaño fijo para almacenarlos. Esto último era clásico en los compiladores de FORTRAN77, dado que en dicho lenguaje sólo se permitían arrays pentadimensionales.

2.2.5 Tipo y forma de acceso de los miembros de estructuras, registros, uniones y clases

El tipo de cada uno de los miembros de una estructura, unión o clase, debe ser almacenado en la TS. En el caso de clases también debe almacenarse la forma de acceso, así por ejemplo C++ permite las formas de acceso *public*, *private* y *protected*.

En el caso de la declaración de funciones amigas (*friend*), también el nombre de estas debe ir ligado a la clase.

En el caso de los tipos simples, en la TS, los tipos se representan habitualmente por las constantes indicadas en el apartado 2.2.2. Sin embargo una estructura puede tener anidada otra estructura. En el caso de aparecer otras estructuras definidas previamente, se utilizará el identificador que representa a dicha estructura.

También puede ocurrir que tengan varias variables de un tipo estructura al cual no se le ha dado un nombre, en ese caso puede optarse por un puntero que señale donde está definida la estructura o por repetir la estructura para cada identificador.

La representación de las uniones es similar a la de las estructura, excepto en el tamaño que ocupan, las uniones tan solo utilizan el tamaño del miembro mayor en bytes.

Las clases tienen datos y métodos. La parte de datos se representa como las estructuras, pero los métodos se almacenan de forma parecida a las funciones, indicando el nombre, tipo de resultado que devuelven, así como el número y tipo de parámetros.

Por otro lado, también tiene que asociarse los accesos (*private*, *public*, *protected*), se señalan las funciones amigas (*friend*) y se deben almacenar los constructores, destructores y métodos virtuales.

El tamaño de una clase, además, también debe reservar espacio para un puntero a la tabla de métodos virtuales[3].

Dado que los lenguajes de programación no colocan un límite al número de miembros de las estructuras, uniones y clases, será necesario implementar una lista dinámica para almacenarlos. En algunas implementaciones sencillas se deja un campo de longitud fija, restringiendo el valor máximo del número de miembros.

[3] Esto permitirá implementar el polimorfismo por herencia.

2.2.6 Tipo de los parámetros de las funciones, funciones libres, procedimientos o métodos de las clases

El tipo que devuelve una función suele ir asociado al tipo base del identificador (ver apartado 2.2.2), sin embargo los tipos de sus parámetros deben de ser almacenados en la TS para realizar las comprobaciones semánticas.

Los tipos de los parámetros se obtienen en lenguajes como C y C++ de la declaración de las funciones prototipo.

Dado que los lenguajes de programación no colocan un límite al número de parámetros de las funciones y procedimientos, será necesario implementar una lista dinámica para almacenarlos.

En algunas implementaciones sencillas se deja un campo de longitud fija restringiendo el valor máximo de números de parámetros.

2.2.7 Descriptor de ficheros

En la TS se puede almacenar el descriptor del fichero o *handle* de bajo nivel que está asociado al identificador del fichero y que se utilizará en la generación de código.

En el caso de lenguajes como PASCAL, en los cuales la declaración del fichero incluye la definición del tipo de sus elementos, también es necesario almacenar dicho tipo.

2.2.8 Otros atributos

Los depuradores de código (*debuggers*) y los analizadores de rendimiento (*profilers*) utilizan información de la TS en tiempo de ejecución, por lo que las distintas implementaciones de los compiladores incorporan información adicional en la TS para estas herramientas que acompañan a los compiladores.

Las listas de referencias cruzadas son otros tipos de atributos que proporcionan una ayuda importante cuando se está en fase de depuración. Estas listas contienen algunos de los atributos vistos anteriormente, también contienen el número de línea del texto fuente donde la variable se declaró (si está declarada explícitamente) o donde se le hace referencia por primera vez (si su declaración es implícita).

Además contienen los números de las líneas donde se hace referencia a la variable.

IDENTIFICADOR	TIPO	DIMENSIÓN	DECLARADA EN	REFERENCIAS
anho	int	0	5	11,23,25
b	real	0	5	10,11,13,23
companhia	int	1	2	9,14,25
forma1	char	2	4	36,37,38
m	proc	0	6	17,21

Figura 2-3. Tabla de Símbolos con referencias cruzadas como atributo de los símbolos.

La Figura 2-3 nos muestra una lista de referencias cruzadas.

Se pueden tener problemas cuando se representan todas las referencias a una variable, pues el número de líneas en que aparece puede ser grande y ocupar mucho espacio en la tabla. Generalmente se utiliza una lista.

Por último también existe la posibilidad de tener un atributo que sea un puntero cuya inclusión facilitará la construcción de la lista de nombres de las variables ordenados alfabéticamente.

3 Operaciones con la TS

Las dos operaciones que se llevan a cabo generalmente en las TS son las inserción y la búsqueda. La forma en que se realizan estas dos operaciones difiere levemente según que las declaraciones del lenguaje a compilar sean explícitas o implícitas.

Otras operaciones son activar (*set*) y desactivar (*reset*) las tablas de los identificadores locales o automáticos.

3.1 TS y Declaración explícita vs implícita

3.1.1 Lenguajes con declaraciones explícitas obligatorias

Las dos operaciones se realizan en puntos concretos del compilador. Es obvio que la operación de inserción se realiza cuando se procesa una declaración, ya que una declaración es un descriptor inicial de los atributos de un identificador del programa fuente.

Si la TS está ordenada, es decir los nombres de las variables están por orden alfabético, entonces la operación de inserción llama a un procedimiento de búsqueda para encontrar el lugar donde colocar los atributos del identificador a insertar. En tales casos la inserción lleva tanto tiempo como la búsqueda.

Si la TS no está ordenada, la operación de inserción se simplifica mucho, aunque también se necesita un procedimiento de colocación. Sin embargo la operación de búsqueda se complica ya que debe examinar toda la tabla.

La operación de búsqueda se lleva a cabo en todas las referencias de los identificadores, excepto en su declaración. La información que se busca (por ejemplo: tipo, dirección en tiempo de ejecución, etc.) se usa en la verificación semántica y en la generación de código.

En las operaciones de búsqueda se detectan los identificadores que no han sido declarados previamente emitiéndose el mensaje de error correspondiente. En las operaciones de inserción se detectan los identificadores que ya han sido previamente declarados, emitiéndose, a su vez, el correspondiente mensaje de error.

3.1.2 Lenguajes con declaraciones implícitas de los identificadores

Las operaciones inserción y búsqueda están estrechamente unidas. Cualquier identificador que aparezca en el texto fuente deberá ser tratado como una referencia inicial, ya que no hay manera de saber a priori si los atributos del identificador ya han sido almacenados en la TS.

Así, cualquier identificador que aparezca en el texto fuente llama al procedimiento de búsqueda, que a su vez, llamará al procedimiento de inserción si el nombre del identificador no se encuentra en la TS. Todos los atributos asociados con un identificador declarado implícitamente se deducen del papel desempeñado por el identificador en el programa.

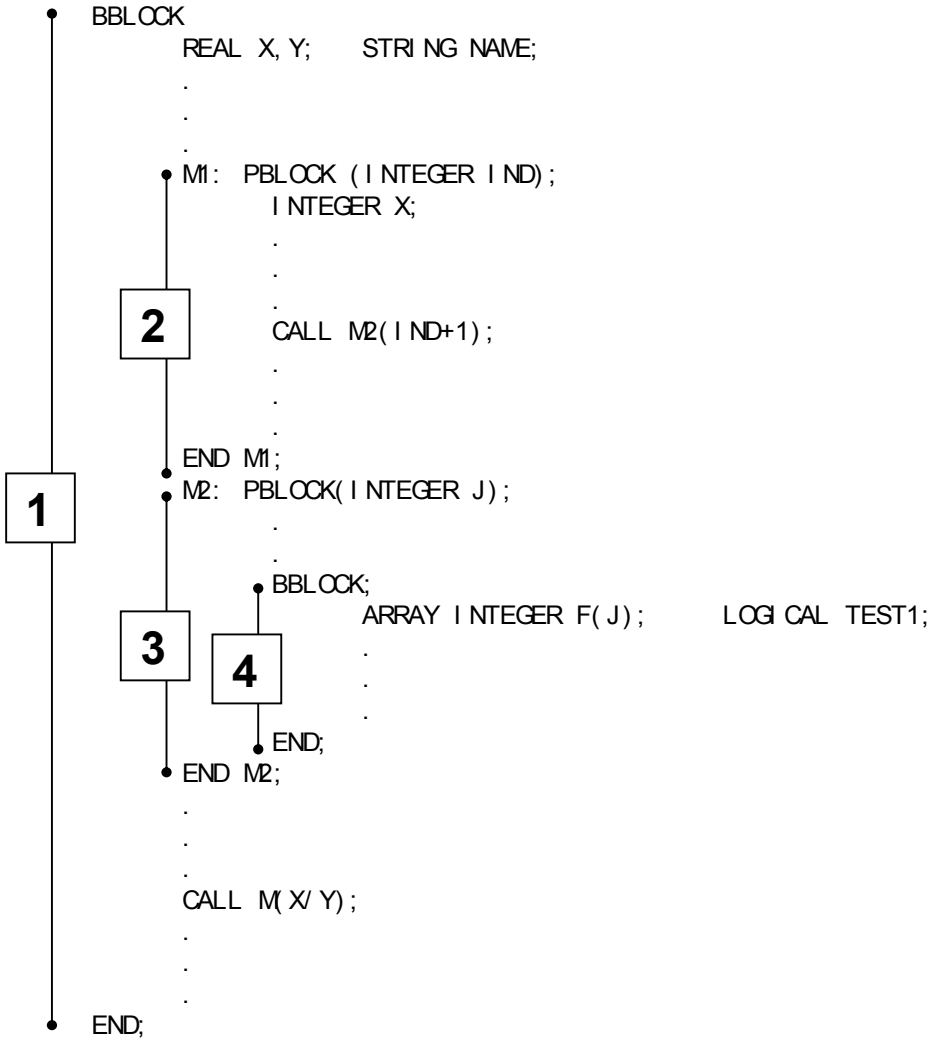
3.2 Operaciones con lenguajes estructurados en bloques

Definiremos en este apartado como lenguajes estructurados en bloques a todos los que tiene algún tipo de bloque que defina ámbitos de utilización y visión de identificadores, por tanto están incluidos en este apartado también los lenguajes orientados a objetos y no sólo los lenguajes estructurados.

3.2.1 Operaciones de activación y desactivación de tablas de símbolos

Los lenguajes **estructurados en bloques** tienen dos operaciones adicionales llamadas ***set*** y ***reset***.

La operación de *set* se utiliza cuando el compilador detecta el comienzo de un bloque o módulo en el cual se pueden declarar identificadores locales o automáticos. La operación complementaria *reset*, se utiliza cuando se detecta el final del bloque o módulo. La naturaleza y necesidad de estas operaciones se muestra en el siguiente fragmento de programa:



En el fragmento del programa anterior, la variable X se declara en más de un bloque y en cada uno de estos bloques X tiene distintos atributos. En un lenguaje anidado es necesario asegurarse, en cada caso, que el nombre de la variable se asocia con un

conjunto único de atributos o de huecos en la TS. La solución de este problema son las operaciones *set* y *reset*.

Con la entrada de un bloque, la operación *set* crea una nueva subtabla (dentro de la TS) en la cual los atributos de las variables declaradas en el nuevo bloque se almacenan. La forma de crear esta subtabla depende de la organización de la TS. En los apartados 4.2 y 4.3 se verán varias técnicas para la creación de las subtablas.

Mediante el uso de subtablas se soluciona la ambigüedad provocada por la búsqueda de identificadores del mismo nombre en distintos bloques.

Por ejemplo, supongamos que se realiza una operación de búsqueda de la variable X en el programa anterior y que las subtablas activas en este momento están ordenadas de manera que la inspección comience por la última creada y siga por las anteriores hasta llegar a la primera (como se muestra en la Figura 3-1).

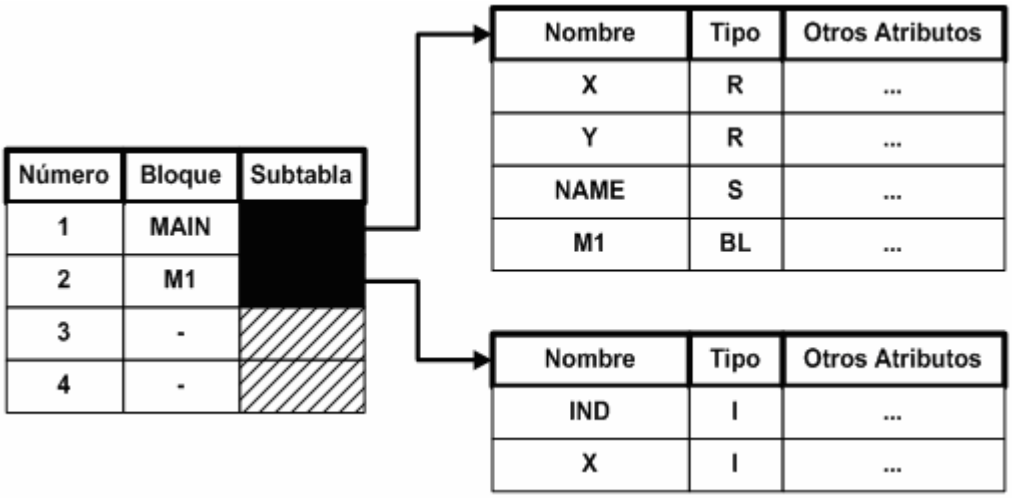


Figura 3-1. Subtablas de un programa estructurado en bloques.

Con esta inspección en orden inverso de creación se garantiza que la primera X que aparezca será la del bloque más cercano al uso de dicha variable que es lo que semánticamente pretende el compilador.

Nótese que no está permitido usar dos variables con el mismo nombre en mismo bloque, de esta forma queda resuelto el problema de la ambigüedad de los nombres de las variables.

Con la salida del bloque, la operación *reset* suprime las entradas a la subtabla del bloque que se acaba de cerrar. Esta supresión de las entradas significa que las variables del bloque recién acabado no tienen validez en el resto del programa. Las técnicas para el cierre de las subtablas se verán también en los apartados 4.2 y 4.3.

4 Organización de la TS

Para estudiar la organización de las tablas de símbolos se estudiará en primer lugar el caso más sencillo en el que no existen identificadores locales y por tanto tampoco subtablas, es el caso de los lenguajes no estructurados en bloques. En segundo lugar se estudiarán los lenguajes estructurados en bloques y por último los lenguajes orientados a objetos.

4.1 Lenguajes no estructurados en bloques

Se entiende por lenguaje no estructurado en bloques al lenguaje en el cual cada unidad compilada separadamente es un módulo simple que no tiene submódulos. Todas las variables declaradas en un módulo se conocen en todo el módulo.

Las organizaciones de la TS que se presentan en este apartado son conceptualmente las más simples. En ellas se estudiará la eficiencia de los accesos a las tablas para hacer comparativas entre las diferentes implementaciones.

Interesa determinar numéricamente la eficiencia de las distintas organizaciones de la TS.

El primer estimador que se usará para medir la complejidad de una operación en la TS es la **Longitud Media de Investigación** (*average length of search*), en adelante **LMI**.

Se define la LMI como el número medio de comparaciones que se necesitan para encontrar un registro de la tabla de símbolos en una organización dada.

Como se ha definido anteriormente, el conjunto de atributos almacenados en la TS para un símbolo determinado se define como **registro de la tabla de símbolos** (*symbol-table record*).

4.1.1 TS no ordenadas

El método más simple de organizar una TS es colocar los registros en la tabla según el orden en que las variables se declaran. En el caso de que los identificadores estén declarados implícitamente, los atributos se colocan según el orden en que las variables aparecen en el texto fuente.

Se examinan ahora las operaciones de inserción y búsqueda en los lenguajes con declaraciones explícitas.

En una operación de inserción no se necesitan comparaciones (a excepción de la comprobación de que la tabla no esté llena), ya que los atributos de cada nuevo identificador se añaden en la tabla a continuación del identificador leído anteriormente. Este procedimiento es un poco idealista, pues ignora el **problema de la duplicación del nombre** de las variables en las declaraciones.

Para detectar este tipo de error se deberá investigar toda la tabla antes de realizar operaciones de inserción. Esta comprobación siempre tiene una *LMI* que se calculará posteriormente en este mismo apartado. En cualquier caso, se debe tener en cuenta que al no estar ordenados los símbolos es necesario en cada caso investigar la tabla completa para garantizar la **no existencia** del símbolo, diferente de cuando el objetivo es encontrarlo en la tabla.

La *LMI* requerida para la operación de búsqueda en una TS no ordenados es:

$$LMI = \frac{\sum_{i=1}^{n} i}{n} = \frac{(n+1)\cdot n}{2\cdot n} = \frac{n+1}{2}$$

siendo *n* el número de registros de la TS.

La deducción de esta fórmula es sencilla, pues para acceder al primer registro hemos de hacer una comparación, para acceder al segundo dos, …, al enésimo n. Entonces el número de comparaciones es:

$$\sum_{i=1}^{n} i = \frac{(n+1)\cdot n}{2}$$

Dado que se ha definido la *LMI* como el número de comparaciones partido por el número de registros, queda demostrada la primera expresión.

La Figura 4-1 representa la *LMI* frente al número de registros de la TS.

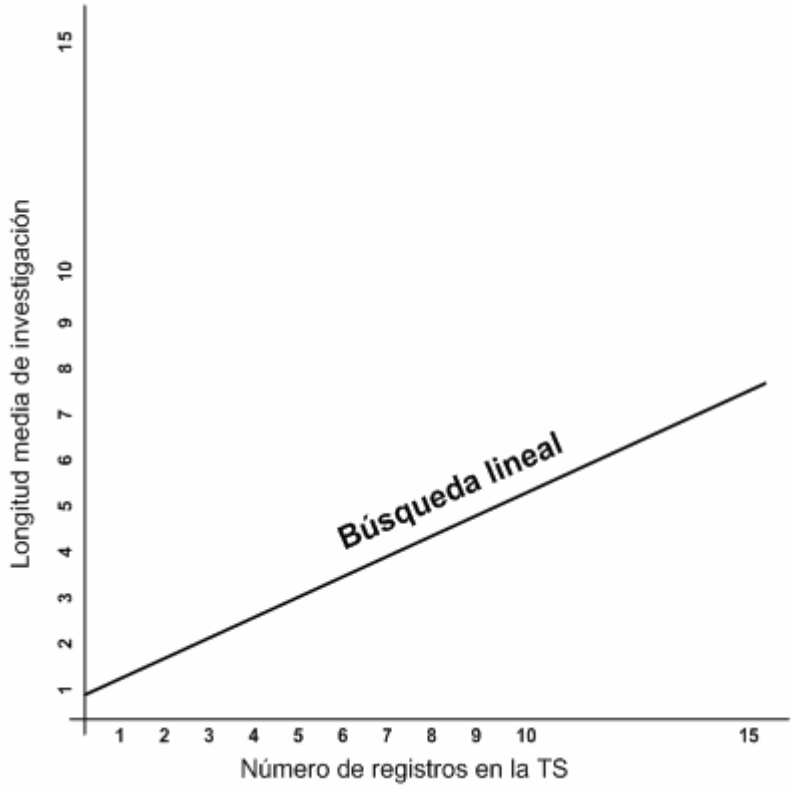


Figura 4-1. Longitud media de investigación

Como se había dicho antes, en los lenguajes con declaraciones implícitas, la operación de inserción debe ir precedida por la operación de búsqueda. Si la operación de búsqueda no detecta el identificador leído en el texto fuente, entonces dicho identificador se inserta en la TS a continuación del último identificador insertado.

Se necesitan *n* comparaciones y una inserción si el proceso requiere una operación de búsqueda seguida por una inserción (en el caso de la primera aparición del identificador). Si sólo fuera necesaria la operación de búsqueda (caso de que el identificador ya hubiera sido detectado), entonces se necesita una media de (*n* + 1)/2 comparaciones para localizar el conjunto de atributos de dicho identificador. Si denominamos *r* al cociente entre el número de primeras apariciones de los identificadores y el número total de apariciones de los identificadores, entonces los procesos de inserción y búsqueda requieren una longitud media de investigación de:

$$LMI = r \cdot n + \frac{(1-r)(n+1)}{2}$$

Una TS no ordenados sólo se usa en el caso de que su tamaño sea pequeño, pues el tiempo medio de búsqueda e inserción es directamente proporcional al tamaño de la tabla. A continuación se estudiarán otras organizaciones de la TS que reducirán el tiempo de investigación sustancialmente.

4.1.2 TS ordenadas

En este apartado y en los siguientes se describirán organizaciones de la TS que se basan en el nombre del identificador.

En tales circunstancias, la operación de inserción debe ser acompañada de un procedimiento de búsqueda que determina en qué lugar de la TS debe ser colocado el identificador.

La inserción en un momento dado de un nuevo registro puede obligar a que parte de los registros ya insertados en al TS se tenga que desplazar para lograr la inserción.

IDENTIFICADOR	TIPO	DIMENSIÓN	OTROS ATRIBUTOS
anio	1	0	...
b	1	0	...
companiia	2	1	...
first	1	0	...
forma1	3	2	...
m	6	0	...
x3	1	0	...

Figura 4-2. Tabla de símbolos ordenada

En la Figura 4-2 se muestra una TS ordenada. Nótese que la tabla está ordenada por el orden alfabético de los nombres de los identificadores.

En una TS organizada de esta forma se pueden aplicar distintas técnicas de búsqueda, veremos dos: la búsqueda lineal y la búsqueda binaria o dicotómica.

En el apartado 4.1.1 se vio la búsqueda lineal en una TS no ordenados. En este apartado se plantea la determinación de las ventajas que se encuentran al utilizar la búsqueda lineal en una TS ordenada.

La operación búsqueda aún requiere una *LMI* de $(n + 1)/2$ si el argumento a investigar está en la tabla. Valor idéntico al de las TS no ordenadas. Sin embargo la operación de inserción conlleva una complejidad adicional respecto a las TS no ordenadas, pues requiere $(n - c)$ comparaciones seguida por c traslaciones. Siendo c el lugar del nuevo registro a insertar. En total son n operaciones elementales: $n - c + c$.

Del análisis anterior se deduce que es poco práctico utilizar la búsqueda lineal en TS ordenadas.

Se puede demostrar que el camino más eficiente para investigar un TS ordenada (usando sólo comparaciones) es la búsqueda binaria. Véase [KNUTH73].

Una aproximación al cálculo de la *LMI*: en el primer paso del algoritmo se compara un registro, en el segundo paso dos registros son candidatos posibles a comparar, en el tercer paso cuatro registros son los candidatos potenciales, y en general, en el paso n los registros potenciales comparables son:

$$2^{n+1}$$

Del análisis anterior se puede deducir que la *LMI*, usando la búsqueda binaria, es:

$$(\log_2 n)+1$$

para una tabla de tamaño n.

Por ejemplo en la TS de la Figura 4-2, *n*=7, luego la longitud máxima a investigar es:

$$(\log_2 7)+1=3$$

La *LMI* se calcula a continuación:

$$\left(\frac{1}{n}\right)\left(1\cdot 1+2\cdot 2+4\cdot 3+\ldots+2^{(\log_2 n)}\cdot\left((\log_2 n)+1\right)\right)=\left(\frac{1}{n}\right)\sum_{i=1}^{(\log_2 n)+1}\left(2^{i-1}\cdot i\right)$$

Cuando *n* tiende a infinito (en nuestro caso a un número suficientemente grade), se demuestra [KNUTH73] que la serie anterior suma aproximadamente:

$$(\log_2 n)-1$$

Como conclusión práctica se puede decir que si el número medio de elementos de la TS por módulo de programa es menor de 10, entonces puede considerarse una estrategia de búsqueda lineal.

Sin embargo si el número de elementos es mayor de 25 se debe ir a búsqueda binaria.

Para calcular $\log_2 n$ se hace del modo siguiente:

$$\log_2 n = x$$

$$n = 2^x$$

$$\log_{10} n = x\cdot\log_{10} 2$$

$$x=\frac{\log_{10} n}{\log_{10} 2}$$

$$x=\frac{\log_{10} n}{0{,}301030}$$

$$\log_2 n = 3{,}3219\cdot\log_{10} n$$

4.1.3 TS con estructura en árbol (AVL)

El tiempo que dura la operación de inserción en una TS ordenada puede reducirse si se utiliza un almacenamiento organizado con una estructura de árbol.

Se comenzará mostrando un ejemplo de TS con estructura de árbol binario como la que se puede ver en la Figura 4-3.

Nótese que se añaden dos nuevos campos a la estructura del registro. Su nombre es *puntero.izquierdo* y *puntero.derecho*.

Se accede al árbol a través del nodo raíz (nodo más alto). El proceso de búsqueda se realiza descendiendo desde el nodo raíz hasta el nodo buscado o hasta un campo de tipo puntero nulo, que se denota por 0 en el ejemplo de la Figura 4-3.

En este último caso la búsqueda es infructuosa. Se puede observar que al final y al principio de cada registro están situados los campos de encadenamiento, que informan de las relaciones entre nodos y permiten un almacenamiento libre de la posición en la TS.

Cuando los punteros tienen un cero en su casillero, se está en el caso de encadenamiento nulo. En caso contrario contienen el número de la posición del campo a señalar, bien sea por la izquierda o por la derecha. El nodo raíz suele ser el primero de la TS.

POSICIÓN	IDENTIFICADOR	TIPO	DIMENSIÓN	OTROS ATRIBUTOS	PUNTERO	
					IZQUIERDO	DERECHO
1	first	1	0	...	2	5
2	b	1	0	...	3	4
3	anho	1	0	...	0	0
4	companhia	2	1	...	0	0
5	m	6	0	...	6	7
6	forma1	3	2	...	0	0
7	x3	1	0	...	0	0

Figura 4-3. Tabla de Símbolos con estructura de árbol.

Los nuevos registros se insertan como nodos hojas de la estructura del árbol binario, con ambos campos de encadenamiento nulos. La inserción se completa con el orden alfabético de la tabla de registros que se dicta por la estructura de encadenamiento del árbol.

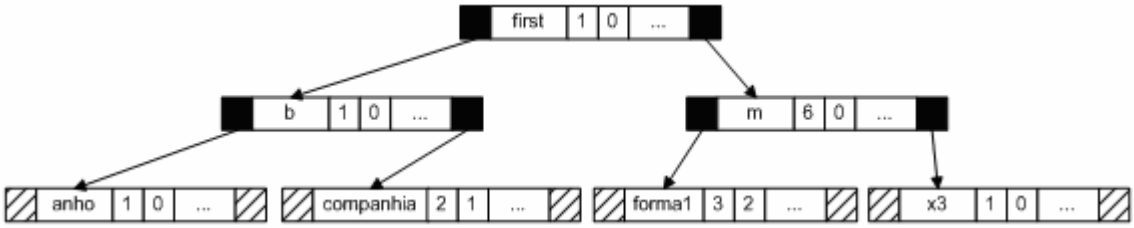


Figura 4-4. Estructura de árbol binario.

Para describir el orden alfabético de un árbol se introducen los conceptos de subárbol izquierdo y subárbol derecho.

Se llama subárbol izquierdo de un nodo a la parte del árbol que contiene todos los nodos accesibles desde dicho nodo por su rama izquierda.

La definición de subárbol derecho es idéntica sólo que por la parte derecha.

Por ejemplo, el conjunto de nodos de las variables *b*, *anho* y *conpanhia* están encadenados estructuralmente formando parte del subárbol izquierdo de la variable *first* en la Figura 4-4.

En un árbol binario, todo nodo situado en un subárbol izquierdo precede alfabéticamente a la raíz del subárbol.

Todo nodo situado en un subárbol derecho sigue alfabéticamente a la raíz del subárbol.

Por ejemplo, si se desea insertar un registro de la variable *cuenta* en el árbol del ejemplo anterior, se llega al árbol de la Figura 4-5.

Se observa que se inserta como una hoja del árbol a la derecha de la variable *companhía*.

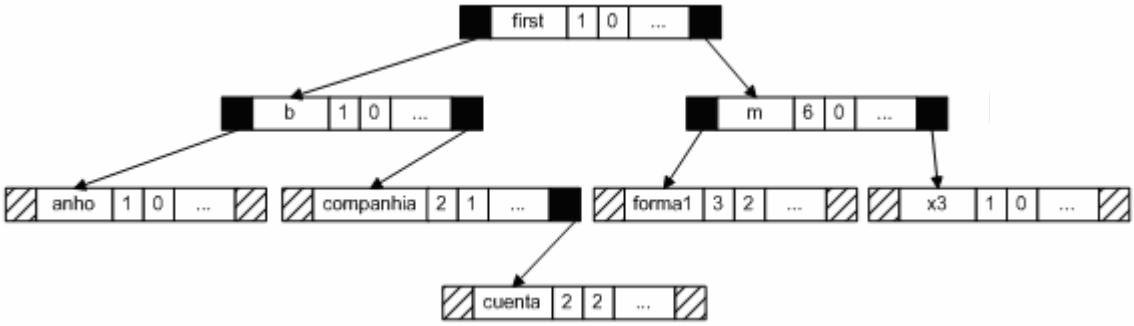


Figura 4-5. Estado del árbol después de introducir el identificador *cuenta*.

Es importante señalar que la estructura del árbol depende del orden de inserción. Para ilustrar la afirmación anterior, se va a examinar la estructura del árbol de la Figura 4-5.

Esta estructura se creó con los registros introducidos por el orden:

- *fisrt*, *b*, *anho*, *companhia*, *m*, *forma1* y *x3*

o también:

- *fisrt*, *b*, *m*, *anho*, *companhia*, *forma1* y *x3*

Sin embargo si el orden de inserción es:

- *forma1*, *b*, *companhia*, *anho*, *m*, *x3*, y *fisrt*

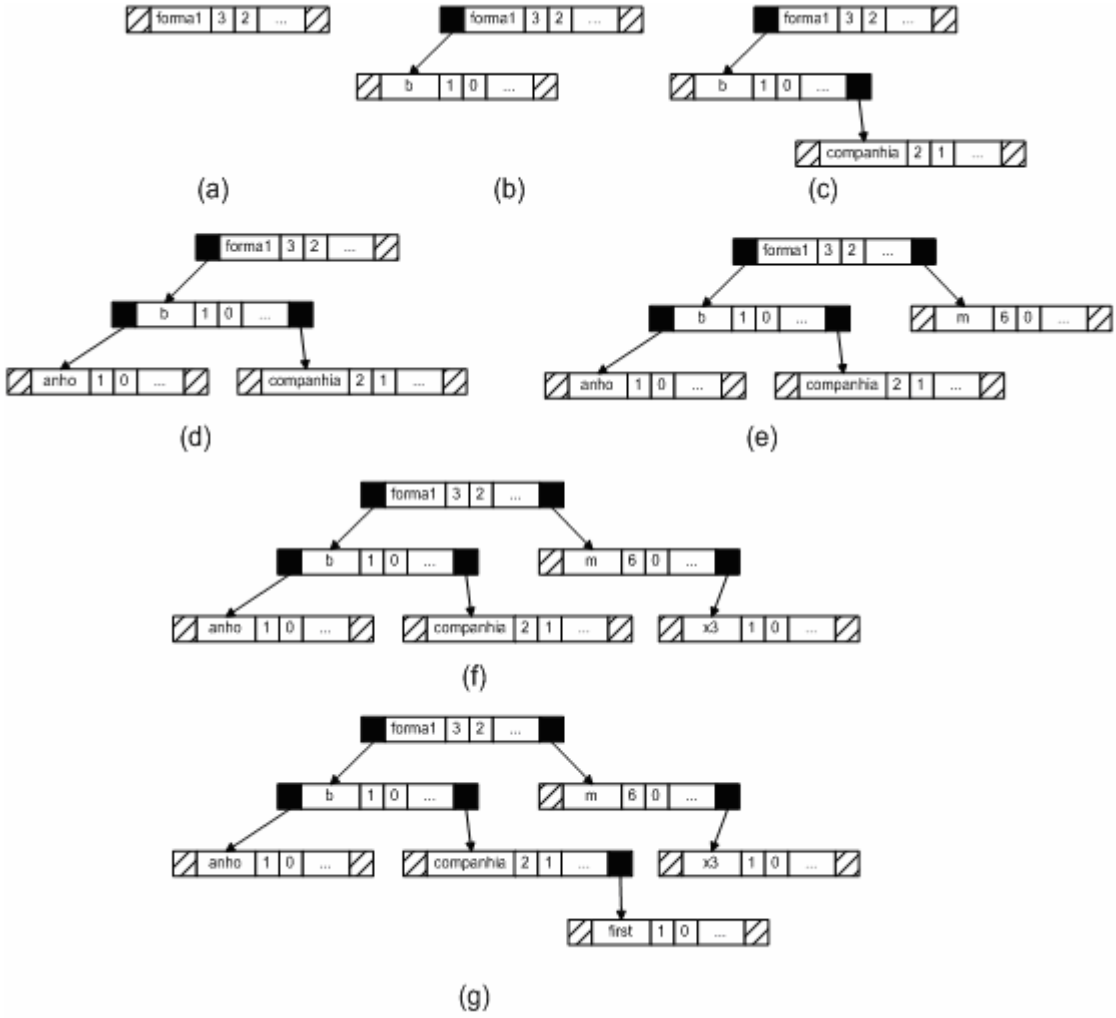


Figura 4-6. Etapas en el desarrollo del árbol binario.

el árbol que se crea es diferente. En la Figura 4-6 se muestran las distintas etapas en el desarrollo de este árbol.

Dado que los registros se insertan conservando un orden particular, una operación de inserción incluye siempre el mismo procedimiento de investigación del árbol que necesita la operación de búsqueda.

En una posible implementación de este algoritmo se tendría en cuenta la operación concreta a realizar.

Si la operación es de inserción y se realiza bien, entonces se devuelve la localización del nuevo nodo del árbol. Si encuentra un registro con el mismo nombre, devuelve el correspondiente error.

En las operaciones de búsqueda, una investigación con éxito devuelve la localización del registro con el nombre del campo que proporciona el argumento buscado.

Si la investigación es infructuosa (por ejemplo, la variable no fue previamente declarada), entonces se devuelve el error correspondiente.

El problema principal de este algoritmo, está en el proceso de inserción, pues se pueden crear árboles no equilibrados.

Por ejemplo, muchos programadores declaran variables por orden alfabético en programas muy largos para facilitar su comprensión y localización en la cabecera del programa.

Si esta práctica se hubiera realizado con las variables vistas en los ejemplos anteriores se crearía el árbol de la Figura 4-7.

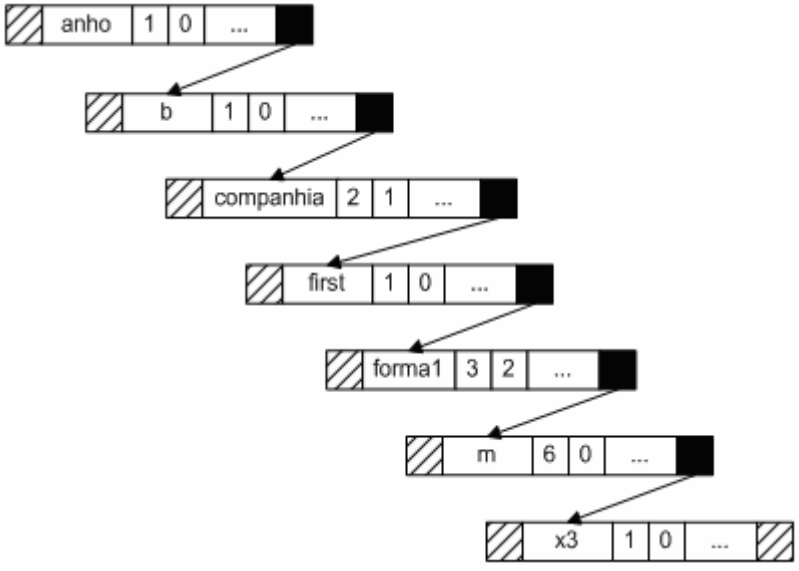


Figura 4-7. Árbol binario degenerado en una lista.

Es obvio que en el caso anterior la estructura del árbol binario ha degenerado en una lista con una *LMI* de:

$$\frac{n+1}{2}$$

Hibbard [HIBB62] demostró que para árboles generados aleatoriamente, la *LMI* más probable es $1{,}4 \cdot \log_2 n$.

Los árboles ideales para su investigación, son los que tienen todas las longitudes máximas de las trayectorias a investigar iguales o casi iguales. De esta forma se evitan casos como el mostrado en la Figura 4-7.

Esta situación es la que ocurre en los árboles binarios equilibrados óptimos. En tales estructuras las distancias (longitud de las trayectorias) de la raíz a cualquiera de los nodos incompletos del árbol difiere como máximo en una unidad.

Nótese que la definición anterior implica la misma probabilidad de acceso a todos los nodos.

Los árboles que se mostraron en la Figura 4-5 y en la Figura 4-6 son equilibrados óptimos.

Desafortunadamente, el principal problema que se presenta en los árboles binarios equilibrados óptimos, se produce en la inserción de registros en la estructura. Por ejemplo, sea el árbol binario equilibrado de la Figura 4-8.

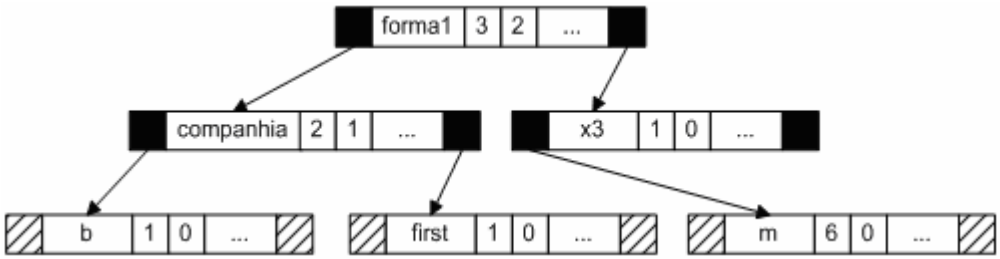


Figura 4-8. Árbol binario equilibrado.

La inserción del registro de la variable *anho* obliga a una completa reorganización, pues cambian todos los campos de encadenamiento de la estructura del árbol, el resultado es el árbol de la Figura 2-3.

Sin embargo, para lograr un árbol equilibrado óptimo se necesita examinar todos los nodos del árbol, lo cual requiere del orden de *n* operaciones elementales.

Para reducir los criterios de equilibrado óptimo, que algunos autores denominan equilibro perfecto, Adelson-Velskii y Landis [AVL62] propusieron otro criterio de equilibrio (no perfecto), que es capaz de buscar, insertar y borrar nodos del árbol con un número de operaciones del orden de $\log_2 n$.

1.1.1.1. Árboles AVL

Un árbol AVL (llamado así en honor de sus autores) es un árbol binario cuyos nodos están en uno de los estados siguientes:

1. Un nodo es pesado a izquierda (*left-heavy*) si la longitud de sus trayectorias en el subárbol izquierdo tienen una unidad más que las del subárbol derecho. Lo denotaremos por **L**.
2. Un nodo está equilibrado si la longitud de sus trayectorias en los dos subárboles son iguales. Lo denotaremos por **B** (*balanced*).
3. Un nodo es pesado a derecha (*right-heavy*) si la longitud de sus trayectorias en el subárbol derecho tiene una unidad más que las del subárbol izquierdo. Lo denotaremos por **R**.

Si cada nodo del árbol está en uno de estos tres estados, se dice que el árbol está equilibrado, en caso contrario está desequilibrado.

Cada nodo tiene un indicador de equilibrio en un campo del registro, que indica el estado del nodo en cada instante.

La Figura 4-9 muestra dos árboles equilibrados.

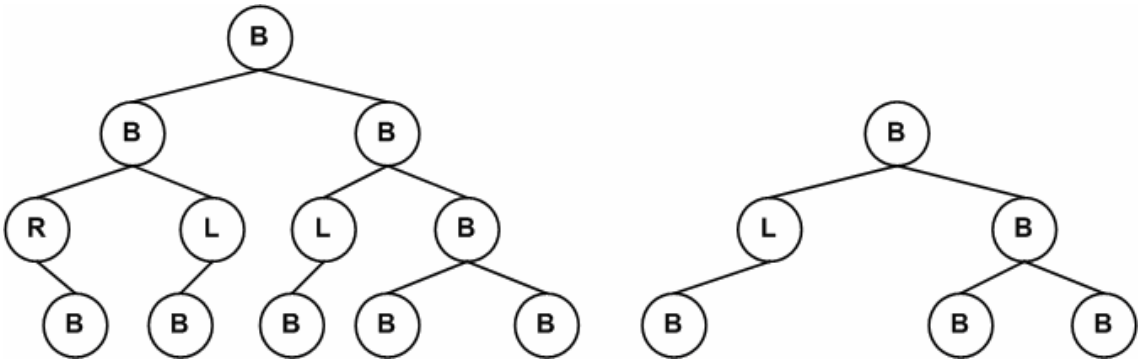


Figura 4-9. Árboles equilibrados.

La Figura 4-10 muestra dos árboles desequilibrados.

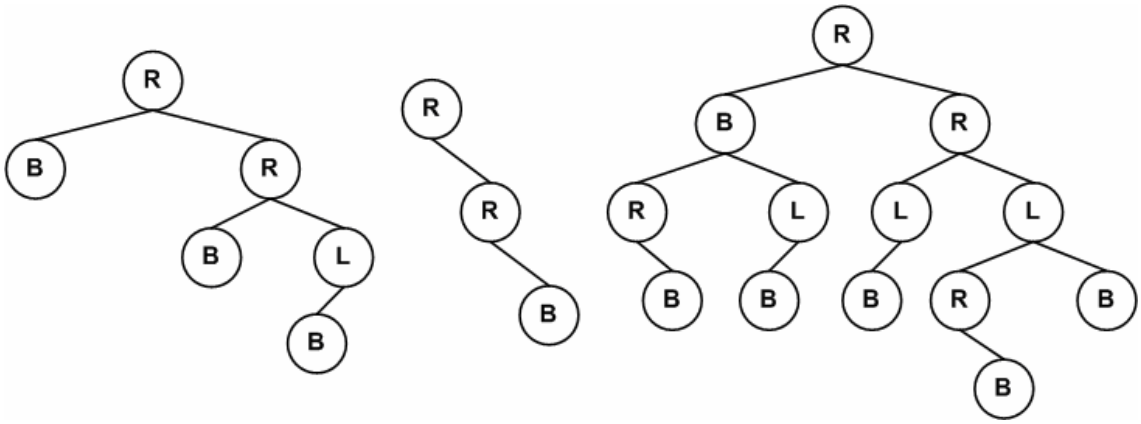


Figura 4-10. Árboles desequilibrados.

En cada nueva inserción, es posible que el árbol se desequilibre, si esto ocurre es necesario ejecutar un algoritmo de equilibrado del árbol, de este modo se volverá el árbol a su estado de equilibrio.

Stone (1972) demostró que la longitud de camino máxima de un árbol AVL de n nodos es:

$$1{,}5 \cdot \log_2(n+1)$$

Knuth [KNUTH73] hace el mismo cálculo con algunas suposiciones empíricas y obtiene la siguiente expresión:

$$\log_2(n+1) + constante$$

Para finalizar se puede mencionar que *Severance* [SEVE74] sugirió una organización del tipo mostrado en la Figura 4-11. Esta estructura es casi un híbrido entre la estructura en árbol y una tabla *hash*, ya que hay un proceso previo de búsqueda a través de la inicial del nombre del identificador.

De este modo el árbol se abre en anchura, pero es más corto en profundidad en la mayor parte de los casos mejorando el acceso a los símbolos.

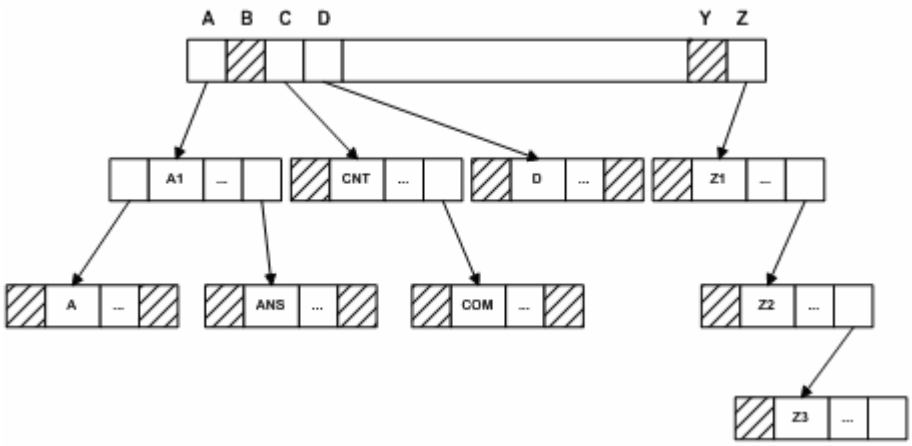


Figura 4-11. Implementación de TS propuesta por Severance.

4.1.4 TS con estructura de tablas hash.

Los métodos de organización de una tabla de símbolos vistos hasta ahora tienen una longitud media de investigación del orden de

$$\log_2 n$$

comparaciones. En este apartado se estudiarán métodos de organización de la tabla de símbolos cuyo tiempo de investigación es prácticamente del número de registros de la tabla.

Comenzaremos introduciendo unos conceptos básicos, que se definirán a continuación.

4.1.4.1 Conceptos básicos

Espacio de nombres o zona de nombres (también llamado espacio de identificación o espacio de claves y en inglés *name space*, *indentifier space* o *key space*). Es el conjunto K de los nombres de las variables que pueden aparecer en un programa.

Ejemplo: En FORTRAN el espacio de nombre es el conjunto de identificadores de una a seis caracteres, que comienzan con una letra y continúan con letras o dígitos. El compilador TURBO PASCAL admite que un identificador tenga 127 caracteres, todos ellos significativos.

Espacio de direcciones (*address space*) es el conjunto A de localizaciones de registros (1, 2, 3, …m) en una tabla.

Una forma de cuantificar la utilización del espacio en una tabla es por medio del **factor de carga** (*load factor*) que es el cociente del número de lugares de la tabla que están ocupados (n) dividido por el número de lugares totales (m).

$$\alpha = \frac{n}{m}$$

Si la tabla está vacía, $\alpha = 0$.

Función Hash (o transformación de claves). Es una aplicación H del conjunto K (de claves) en el conjunto A (de localizaciones de registros).

$$H : K \rightarrow A$$

La función hash H toma el nombre de una variable o identificador y calcula la dirección de la tabla en que se almacenan los atributos de dicho identificador o variable. Esta

función *H* realiza los cálculos por medio de operaciones aritméticas o lógicas con el nombre de la variable o parte de dicho nombre.

Antes de comenzar a describir las funciones hash se introducirá el concepto de **preacondicionamiento** (*preconditioning*).

Los elementos del conjunto *K* están formados por caracteres alfanuméricos y especiales que son muy adecuados para manejarse con operaciones aritméticas o lógicas. El proceso de transformación del nombre en formas fácilmente manejables por funciones hash se denomina preacondicionamiento.

Como ejemplo se puede ver el proceso de preacondicionamiento de la variable *AD#*.

La primera alternativa para realizar dicho preacondicionamiento es codificar las letras, números y caracteres especiales como sigue:

Letra	0	1	…	9	A	B	…	Z	#	+	-
Código	1	2	…	10	11	12	…	36	37	38	39

Si se utiliza la tabla anterior para codificar la variable de nombre *AD#* llegamos a:

$$AD\# \rightarrow 111437$$

El preacondicionamiento puede realizarse más eficientemente utilizando la representación interna del código numérico (por ejemplo ASCII, EBCDIC o UNICODE) de cada carácter del nombre de la variable.

Sobre una máquina concreta se puede codificar *AD#* como:

1000001	1000100	0100011	ASCII en binario
101	104	043	ASCII en octal
65	68	35	ASCII en decimal
C1	C4	7B	EBCDIC en hexadecimal
11000001	11000100	01111011	EBCDIC en binario
etc.…			

Nótese que se pueden tener problemas con la longitud de los números generados por los nombres de los identificadores. De hecho se describirán algunas funciones hash que efectúan varios tipos de transformaciones para reducir el tamaño del número generado.

En general se utilizan dos funciones hash, una para preacondicionar el nombre del identificador transformándolo en un número y la segunda para transformar el número en una dirección que señala su localización en la TS.

A continuación se estudiarán varias funciones hash que se pueden aplicar para el manejo de TS. A partir de ahora se utilizará el término **clave** para identificar la representación numérica preacondicionada del nombre de un identificador.

4.1.4.2 Métodos de organización de las tablas hash

Una de las funciones hash más conocidas es el **método de la división** que se define como:

$$H(x) = (x \bmod m) + 1$$

para un divisor m.

Esta función tiene la propiedad de preservar la distribución uniforme de las claves en el espacio de direcciones.

Cuando las claves son alfabéticas (es el caso habitual en las TS), debe evitarse que *m* sea una potencia de 2, pues en este caso la hipótesis de que todas las claves son igualmente probables es errónea. De hecho, es casi seguro que claves que difieran en sólo unas pocas letras sean asignadas a la misma dirección, produciéndose una distribución muy poco uniforme.

De todo esto se deduce que lo mejor es que *m* sea un número primo.

Por ejemplo, las claves 2000, 2001, 2002, …, 2007 se asignan a las direcciones 82, 83, 84, …, 92 con un valor de *m* = 101.

Cuando a varias direcciones diferentes se les asigna la misma dirección se dice que ha ocurrido una **colisión**. Al proceso de generar direcciones alternativas se le llama **manejo de colisiones** o **resolución de colisiones** (*collision resolution*).

Este es uno de los problemas principales de la transformación de claves, pues el conjunto de claves posibles es mucho mayor que el conjunto de direcciones de la TS disponibles.

Del ejemplo visto anteriormente se desprende que dos claves colisionan cuando son congruentes módulo 101.

La segunda función hash que se va a estudiar es el método hash cuadrático (*midsquare hashing method*). Este método se basa en multiplicar la clave por sí misma y extraer varios dígitos centrales que constituyen la dirección.

Por ejemplo, sea la clave 113586. Al elevarla al cuadrado obtendremos 12901779396, si las direcciones que usamos son de cuatro dígitos podemos escoger 1779. Deben de elegirse estas posiciones para todas las claves.

Este método fue criticado por Buchholz [BUCH63], pero da buenos resultados cuando se aplica a algunos conjuntos de claves (Lum et al. (LUM71]).

La siguiente función hash se basa en el **método de reducción** (*folding method*). Consiste en partir la clave en grupos de dígitos con el mismo número de dígitos excepto el último grupo que puede tener un número menor. Los distintos grupos de dígitos se suman excepto el último, pudiéndose dar la vuelta al orden de los dígitos. Por ejemplo, sea la clave 18724965321, si se divide en grupos de tres dígitos, se tiene:

187 249 653 21

Si se cambia de orden a los situados en lugar impar y se eliminan los de menos de tres cifras se tendrá:

781 249 356

Sumándolos se llega a 1356.

Evidentemente se pueden hacer muchas variaciones del caso mostrado en el ejemplo. Si la clave está en binario se puede utilizar la operación "o exclusiva" en vez de la suma. En general este método se utiliza para comprimir claves de palabras muy largas, pudiéndose usar posteriormente otras funciones hash.

Otra técnica muy utilizada para la realización de funciones hash es el **método dependiente de la longitud** (*length-dependent method*). Utiliza la longitud del nombre de la variable en conjunto con alguna parte del nombre para determinar una dirección de la tabla directamente. En otros casos se calcula una clave intermedia que puede usar, por ejemplo, el método de la división para determinar la dirección de la tabla.

Por ejemplo: sea la variable *ID#1*, se toma el código decimal del primero y último carácter, se suma y se añade el número de caracteres multiplicado por 16.

A la variable *ID#1* se le asigna el código decimal (EBCDIC)

201 196 123 241

Se realza la operación: 201 + 241 + (4*16) = 506. Si ahora se aplica el método de la división con *m*=29 resulta la dirección 14.

4.1.4.3 Manejo de colisiones

Anteriormente se ha definido "colisión" como el suceso que se produce cuando se asignan dos claves a la misma dirección.

Cuando ocurre esto se calcula una dirección alternativa para la segunda clave, llamándose a este proceso "manejo de colisiones".

Las dos técnicas más utilizadas para el manejo de colisiones son:

- Direccionamiento vacío o hash cerrado.
- Encadenamiento directo o hash abierto.

4.1.4.3.1 Direccionamiento vacío o hash cerrado

Consiste en buscar en otros lugares de la tabla hasta que se encuentre la variable buscada (caso de operación de búsqueda) o se llegue a un lugar vacío (en el caso de operación de inserción significa que no existe y se puede insertar, en el caso de búsqueda se termina la operación infructuosamente).

La forma de buscar otros lugares de la tabla puede hacerse por medio de varios métodos. El primer método que se va a estudiar es el método llamado **inspección lineal** (*linear probing*).

Es el método más sencillo. Consiste en mirar el lugar siguiente de la tabla y así sucesivamente hasta encontrar la variable buscada o una posición vacía. Cuando se llega al final de la tabla se pasa a mirar el primer lugar de la tabla y se continúa buscando. Si se llega al lugar de partida sin encontrar la variable buscada entonces no está en la tabla. Si no se encuentra el hueco para introducirla es que la tabla está llena. Este método tiene la desventaja de que los elementos tienden a agruparse alrededor de las claves primarias (claves que han sido insertadas sin colisionar).

Por ejemplo, al insertar los siguientes identificadores y en este orden:

NODE, STORAGE, AN, ADD, FUNCTION, B, BRAND y PARAMETER

En la siguiente tabla se muestra la clave generada para cada uno de los identificadores. En ella se puede comprobar que hay varias colisiones:

NOMBRE DEL IDENTIFICADOR	DIRECCIÓN
NODE	1
STORAGE	2
AN	3
ADD	3
FUNCTION	9
B	9
BRAND	9
PARAMETER	9

Si se utiliza la inspección lineal para resolver las colisiones en una tabla con $m = 11$ se obtiene el resultado de la Figura 4-12.

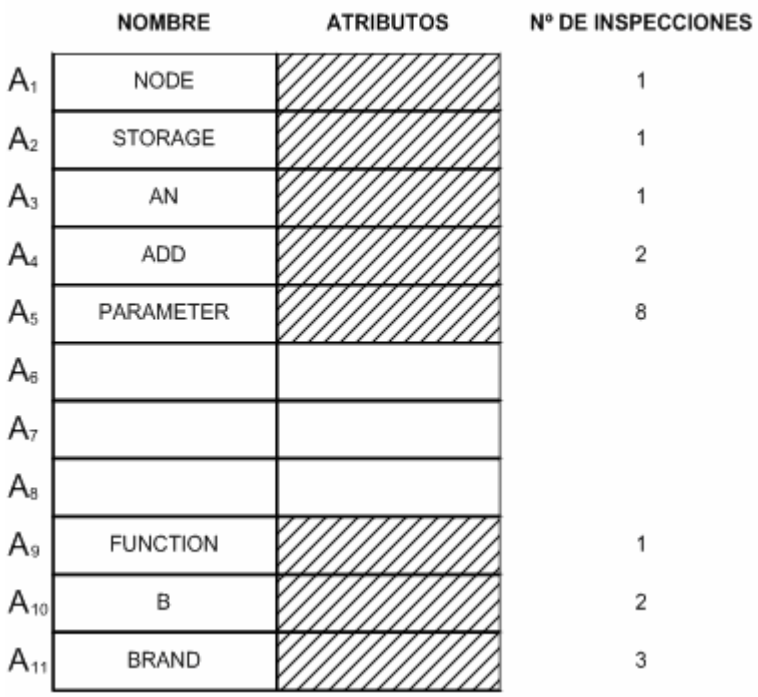


Figura 4-12. Tabla hash con colisiones.

La *LMI* es el número de variables dividido por el número de inspecciones. Knuth [KNUTH73] construyó un modelo probabilístico para analizar las técnicas de manejo de colisiones y desarrolló fórmulas que calculan la *LMI* esperada o probable en función del factor de carga $\alpha = n / m$.

Para la inspección lineal son:

$$E(LMI) = \alpha \frac{1 + \left(\frac{1}{1-\alpha} \right)}{2}$$

$$E(LMI) = \alpha \frac{1 + \frac{1}{(1-\alpha)^2}}{2}$$

En la siguiente tabla se representan los valores de las dos fórmulas:

FACTOR DE CARGA α	NÚMERO DE INSPECCIONES	
	CON ÉXITO	INFRUCTUOSAS
0,10	1,056	1,118
0,20	1,125	1,281
0,30	1,214	1,520
0,40	1,333	1,889
0,50	1,500	2,500
0,60	1,750	3,625
0,70	2,167	6,060
0,80	3,000	13,000
0,90	5,500	50,500
0,95	10,500	200,500

Se observa que el número de inspecciones esperadas es proporcional al factor de carga.

Anteriormente se comentó y se pudo observar en el ejemplo que uno de los problemas de inspección lineal es la agrupación alrededor de las claves primarias de las claves colisionadas. Tal fenómeno se llama **agrupación primaria** (*primary clustering*).

El problema anterior se puede reducir utilizando distintos métodos de inspección. Un método que puede lograr esto es el que se llama **inspección aleatoria** (*random probing*). Esta técnica consiste en generar posiciones aleatorias en vez de las posiciones ordenadas que se generaban en la inspección lineal. Se general *m* posiciones aleatorias entre 1 y *m* (siendo m el número de registros de la tabla). La tabla se considera completa cuando se repite el número generado. Nótese que la generación es cíclica. Por ejemplo:

$$R(y) = (y + c) \bmod m$$

donde *y* es la dirección inicial generada por la función hash, *c* es una constante y *m* es el número de registros de la tabla.

Supóngase $c = 7$, $m = 11$ e $y = 3$ en un momento dado, la secuencia generada es la siguiente:

10 6 2 9 5 1 8 4 0 7 3

si se añade 1 a cada número generado, transformaremos la secuencia el números del intervalo (1,11). Cuando se utilizan fórmulas de esta forma se debe cumplir que *c* y *m* sean números primos entre si.

Aunque la inspección aleatoria mejora el problema de la agrupación primaria, la agrupación de direcciones puede todavía aparecer. Esta situación surge cuando dos claves se transforman en la misma dirección. En tales casos se genera la misma secuencia para ambas claves por el método de inspección aleatoria. A este fenómeno se le llama **agrupación secundaria** (*secondary clustering*).

Este tipo de agrupamiento se puede atenuar por medio del **doble hashing** o **rehashing**. En este método el valor del incremento *c* se calcula utilizando una segunda función hash H_2 que es independiente de la función hash inicial H_1 y que genera un valor que es primo del número de registros de la tabla.

La función H_2 se usa para calcular el valor de *c* que se introduce en la fórmula de la inspección aleatoria ($(y + c) \bmod m$).

Kntuh [KNUTH73] sugiere utilizar:

$$H_1(k) = 1 + k \bmod m$$
$$H_2(k) = 1 + (k \bmod (m - 2))$$

siendo *k* la clave y *m* el número de registros de la tabla. Así por ejemplo si:

$$k = 125$$
$$m = 13$$

se tiene:

$$H_1(125) = 9$$
$$H_2(125) = 5$$

y la serie de valores generada es:

9 1 6 11 3 8 0 5 10 2 7 12 4

La *LMI* esperada para la técnica de rehashing, cuando H_1 y H_2 son independientes se muestra a continuación:

$$E(LMI) = \frac{-\ln(1-\alpha)}{\alpha}$$
$$E(LMI) = \frac{1}{1-\alpha}$$

Seguidamente se muestra la tabla representativa de las fórmulas anteriores para la inspección aleatoria con doble hashing.

Se puede observar como ha mejorado respecto a la inspección lineal.

FACTOR DE CARGA α	NÚMERO DE INSPECCIONES	
	CON ÉXITO	INFRUCTUOSAS
0,10	1,054	1,111
0,20	1,116	1,250
0,30	1,189	1,429
0,40	1,277	1,667
0,50	1,386	2,000
0,60	1,527	2,500
0,70	1,720	3,333
0,80	2,012	5,000
0,90	2,558	10,000
0,95	3,153	20,000

Hay tres dificultades principales para la utilización del método de direccionamiento vacío o hash cerrado:

1. Cuando se va a localizar un hueco para insertar un identificador se necesitan muchas inspecciones, pues suele haber muchos registros ocupados por identificadores que se introdujeron por medio de otras inspecciones.
2. EL tamaño de la tabla es fijo y no puede ajustarse según cambian las necesidades. Por tanto hay que establecer una buena estimación del mismo a priori si quiere evitarse una mala utilización de la memoria o un rendimiento pobre (o incluso un desbordamiento de la tabla), aún cuando se conoce exactamente el número de elementos (caso muy hipotético cuando se trata de un compilador). El problema del desbordamiento de la tabla no puede olvidarse en las tablas de símbolos de compiladores, ya que las necesidades de espacio en la tabla dependen directamente del tamaño del programa fuente y en especial del número de identificadores.
3. El último problema es la dificultad que se encuentra cuando se quieren borrar elementos de la tabla pues es un proceso lento y pesado. Una solución es utilizar la técnica del encadenamiento directo o hash abierto.

4.1.4.3.2 Encadenamiento directo o hash abierto

Este método enlaza todos los identificadores que tienen el mismo índice primario H(k) en forma de lista. Es decir, se resuelven las colisiones utilizando una lista encadenada que une todos los identificadores que tienen colisión. La implementación de la lista puede ser en forma dinámica o por medio de otra tabla que se llama *área de desbordamiento (overflow area*).

A la tabla primaria también se le llama *área principal* (*prime area*).Este método es muy eficiente, aunque tiene la desventaja de manipular listas secundarias y, además, tiene que almacenar un atributo adicional a cada variable para almacenar el puntero encadenamiento de la lista de identificadores colisionados.

En las siguientes figuras se esquematiza el encadenamiento directo.

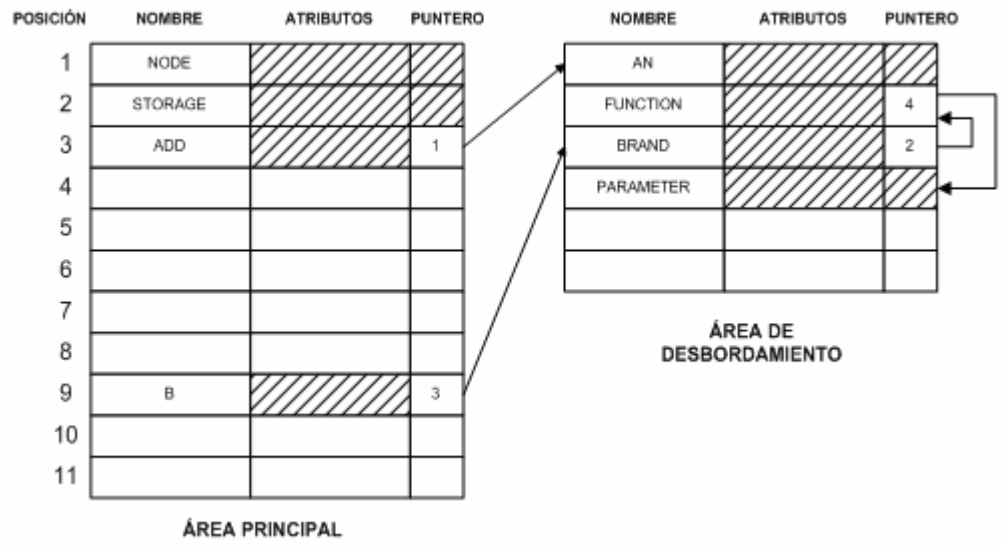


Figura 4-13. Encadenamiento directo con Tabla de Desbordamiento

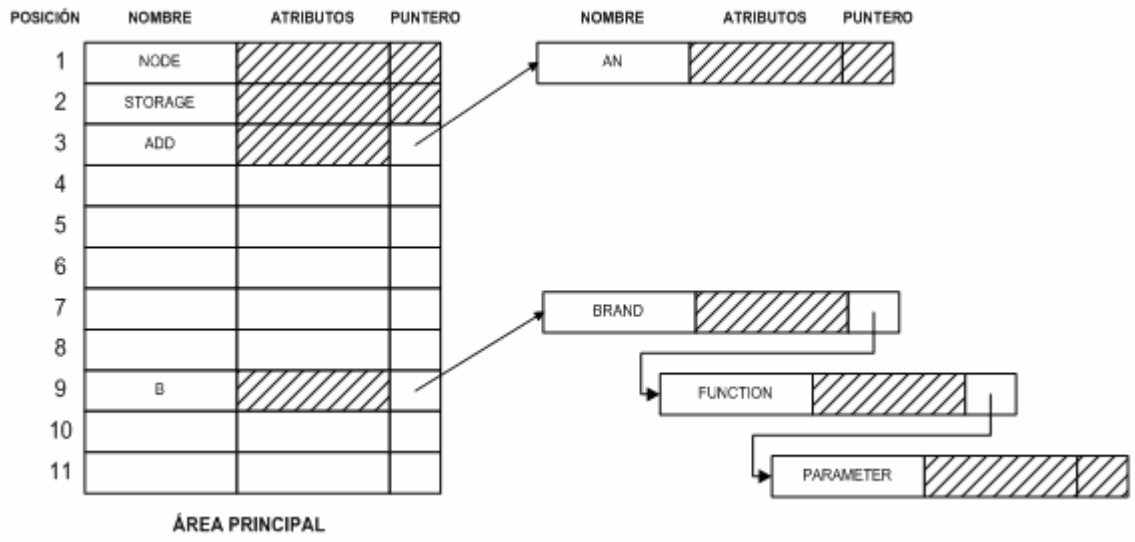


Figura 4-14. Encadenamiento directo con lista enlazada dinámicamente

Debe notarse que, en este caso, la inserción en la lista de registros colisionados se hacen por orden alfabético del nombre de los identificadores. Está demostrado que esta técnica reduce el tiempo de búsqueda en investigaciones infructuosas. Knuth [KNUTH73] demostró que la longitud media de investigación esperada para el encadenamiento directo es la siguiente:

$$E(l.m.\text{de } i) = 1 + \frac{\alpha}{2}$$

$$E(l.m.\text{de } i) = \alpha + e^{-\alpha}$$

A continuación se muestra una tabla con los valores más representativos de estas fórmulas.

FACTOR DE CARGA α	NÚMERO DE INSPECCIONES	
	CON ÉXITO	INFRUCTUOSAS
0,10	1,050	1,005
0,20	1,100	1,019
0,30	1,150	1,041
0,40	1,200	1,070
0,50	1,250	1,107
0,60	1,300	1,149
0,70	1,350	1,197
0,80	1,400	1,249
0,90	1,450	1,307
0,95	1,475	1,337

En estas tablas se observa que ha mejorado respecto al hash cerrado. Todavía puede optimizarse más la utilización de tablas de símbolos con direccionamiento hash. En la Figura 4-14 puede observarse que la *Tabla Primaria* tiene muchos huecos debido a los problemas de colisiones. Se puede optimizar aún más la tabla definiendo la *Tabla Principal* como un array de punteros a registros. De este modo los huecos libres son del tamaño de un puntero y no del tamaño de un registro. Se puede ver el esquema en la Figura 4-15.

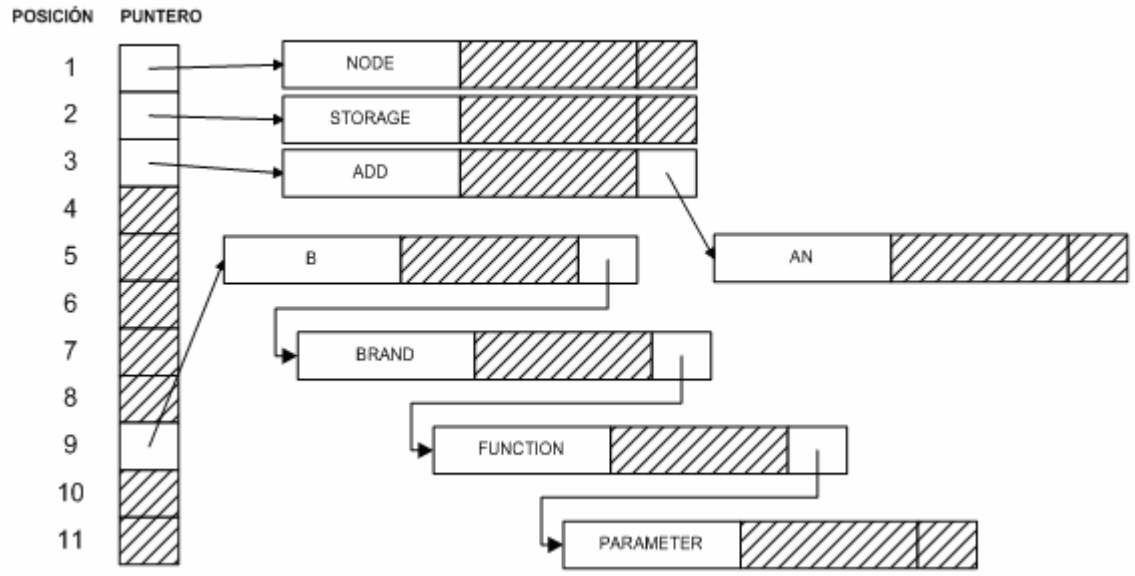


Figura 4-15. Tabla de símbolos con array de punteros a registros

4.1.5 Conclusiones

Si se va a compilar programas con menos de 10 variables, se pueden usar tablas de símbolos no ordenadas. Si el número de variables no supera a 25 se pueden usar tablas de símbolos ordenadas, utilizando el algoritmo de búsqueda binaria.

Las tablas de símbolos con estructura de árbol se pueden utilizar en ciertos lenguajes con unas características muy determinadas (por ejemplo BASIC y FORTRAN tienen limitados los nombres de los identificadores a unos pocos caracteres).

El mejor método cuando no se tienen problemas de memoria reducida es el direccionamiento hash abierto, su problema principal es la asignación de memoria a priori para la tabla de símbolos. Esto se palia en parte utilizando en el compilador algún parámetro que nos dé una idea a priori del tamaño de la tabla de símbolos (por ejemplo el número de líneas del programa). Otro método igualmente válido son los árboles AVL.

Como conclusión práctica, el método hash abierto con arrays de punteros a registros es el que ofrece las mejores prestaciones, teniendo en cuenta su fácil y cómoda implementación. El único inconveniente está en tener que fijar a priori el tamaño máximo del array de punteros a registros. Sin embargo el uso de listas dinámicas puede garantizarnos, si hay suficiente memoria, que no se colapsará el compilador si se alcanzan valores próximos al tamaño máximo del array.

Es muy importante elegir bien una función de preacondicionamiento para reducir las colisiones. La función hash más utilizada y de más fácil comprensión es la función módulo, que además asegura una distribución uniforme a través del array de punteros (para mejorar aún más este comportamiento se suele elegir un tamaño del array que sea número primo).

En los modernos lenguajes de compilación se poseen estructuras de contenedores con mejoras significativas de este comportamiento, es el caso de las tablas hash implementadas por *hashtable* de *Java* o *map* de *C++*. En estos casos, *m* es un valor dinámico que se va cambiando a medida que la tabla se llena, de manera que las colisiones se reducen o se eliminan completamente.

4.2 Lenguajes estructurados en bloques

En este apartado se estudiarán los problemas que se presentan en el manejo de tablas de símbolos cuando se compilan lenguajes estructurados en bloques. (Aunque también sean lenguajes estructurados los lenguajes orientados a objetos se tratarán en otro apartado)

Se entiende por lenguaje estructurado en bloques a todo lenguaje con estructura de bloques o módulos que a su vez puede contener submódulos anidados y de manera que cada submódulo pueda contener un conjunto de identificadores con ámbito local.

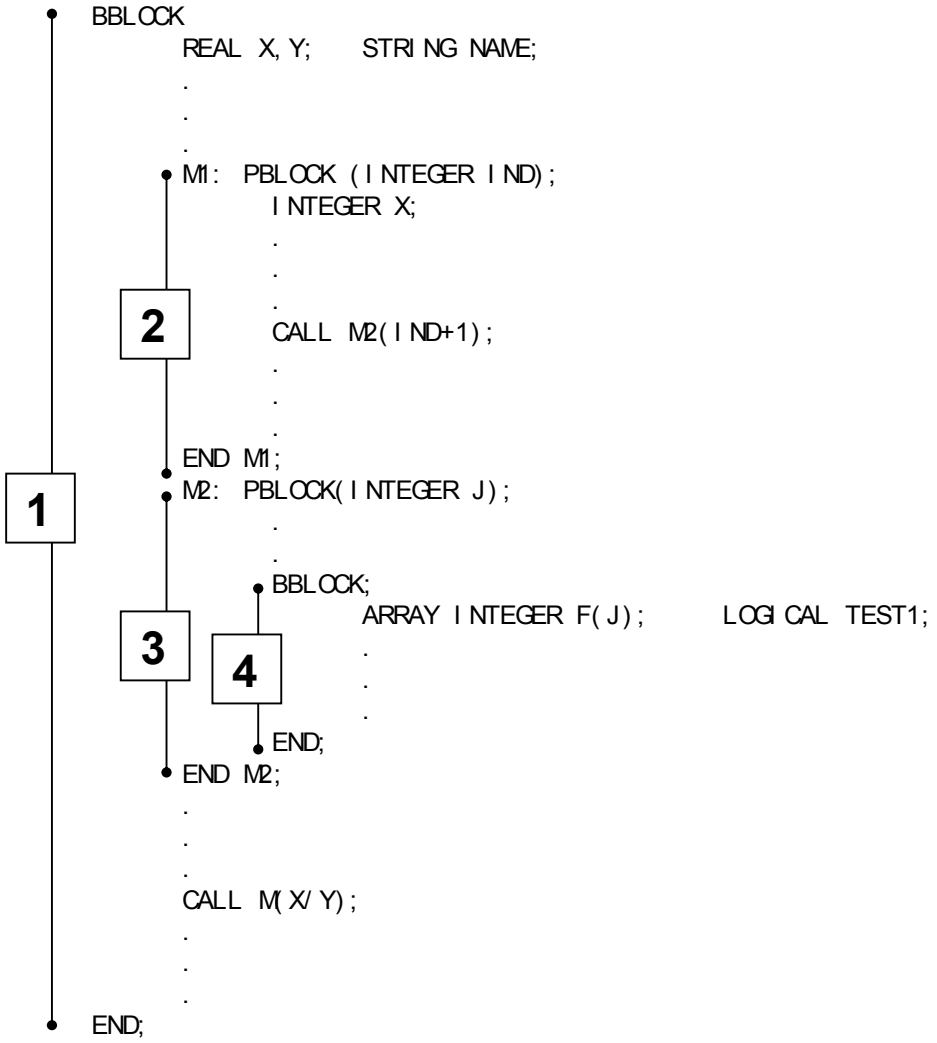
Un identificador declarado dentro de un módulo A es accesible dentro del módulo a no ser que el mismo nombre del identificador se redefina dentro del submódulo A. La redefinición de una variable es válida exclusivamente dentro del ámbito ("*scope*") del submódulo.

Por ejemplo en FORTRAN el ámbito de una variable es una subrutina. En general este concepto se aplicará más a los lenguajes tipo ALGOL, PASCAL, C e incluso a los propios lenguajes orientados a objetos.

4.2.1 Otros conceptos relacionados

Acabamos de definir lenguaje estructurado en bloques, pero este concepto necesita otras definiciones elementales para comprenderlo en toda su amplitud.

En el apartado 3.2 se introdujeron las operaciones *set* y *reset*. En este apartado se estudiarán con más detalle dichas operaciones. Para explicarlas se usará el mismo programa ejemplo que se vio entonces y que se reproduce a continuación con números que identifican lo bloques.



En el ejemplo anterior se denomina con BBLOCK los bloques que se ejecutan de forma secuencial y con PBLOCK los que son bloques de procedimientos y a los cuales se accede por medio de una llamada al procedimiento y una vez ejecutado éste se devuelve el control del programa al bloque principal.

En tiempo de ejecución estos dos tipos de bloque se comportan de forma diferente, pero durante la compilación ambos tipos usan los mismos procedimientos en lo que respecta a las operaciones de *set* y *reset*.

Cuando en la compilación de un texto fuente el traductor se encuentra un bloque la operación *set* crea una subtabla para todas las nuevas variables declaradas.

El nacimiento de una nueva subtabla no implica obligatoriamente una necesidad suplementaria de espacio en la tabla de símbolos, pues los atributos de las nuevas

variables se almacenan en la subtabla durante la compilación del bloque y se borran[4] después de que se lee el END del bloque. La operación de borrado se realiza por medio de *reset*. Esta operación también debe volver a activar la parte de la tabla de símbolos que contiene las variables presentes en el programa antes de la entrada en el bloque.

En el ejemplo anterior, antes de que el bloque 2 sea analizado por el traductor e inmediatamente después de ser analizado, solamente las variables X, Y, NAME y M1 están activas en la tabla de símbolos. La tabla siguiente muestra las variables activas en la tabla de símbolos, así como las inactivas, en el instante anterior a la ejecución de las sentencias *set* y *reset* para el ejemplo anterior.

Operación	Contenidos de la tabla de símbolos (sólo nombre)	
	Activas	Inactivas
Set BLK1		
Set BLK2	M1, NAME, X, Y	
Reset BLK2	X , IND, M1, NAME, X, Y	
Set BLK3	M2, M1, NAME, X, Y	X, IND
Set BLK4	J, M2, M1, NAME, X, Y	X, IND
Reset BLK4	TEST1, F, J, M2, M1, NAME, X, Y	X, IND
Reset BLK3	J, M2, M1, NAME, X, Y	TEST1, F, X, IND
Reset BLK1		J, M2, M1, NAME, X, Y, TEST1, F, X, IND

Inicialmente, tanto la parte activa como la inactiva de la tabla de símbolos están vacías. Antes de comenzar el segundo bloque, las variables M1, NAME, X e Y están localizadas en la zona activa de la tabla de símbolos. Los registros X e IND se añaden se añaden durante la compilación del bloque 2. La operación *reset* que se ejecuta en el END del bloque 2 coloca a X e IND en la zona inactiva de la tabla de símbolos.

Este proceso se realiza para todos los bloques del texto fuente y, una vez finalizada la lectura de éste, todos los registros de la tabla de símbolos están inactivos.

En el ejemplo, las declaraciones de los procedimientos (denotados por PBLOCK) se realizan cuando el nombre del procedimiento se encuentra por primera vez. Por esta razón, el símbolo M2 (en este caso procedimiento) continúa activo una vez que el bloque ha sido compilado. Nótese que la referencia hecha a M2 en el bloque 1 puede tener problemas al no estar declarada todavía. Es el caso de la sentencia FORWARD del PASCAL o la compilación en varias pasadas. Este problema se ignorará en el estudio de las tablas de símbolos y se estudia en la fase de análisis semántico (ver [ORTIN04]), por tanto la declaración de un procedimiento se manejará como cualquier otra declaración.

[4] Realmente no es necesario borrarlos, simplemente es necesario desafilarlos y dejar nuevo espacio disponible para apilar nuevos símbolos.

Si observamos detenidamente la tabla anterior, se ve que las variables se almacenan en la tabla como si ésta fuera una pila LIFO (*last input first output*). Esto no es una coincidencia, pues es una consecuencia del uso de bloques anidados.

Otra propiedad de los lenguajes estructurados en bloques que se debe considerar en el diseño de tablas de símbolos es la duplicidad de los nombres de las variables. Por ejemplo en el programa dentro del bloque 2, cualquier referencia a X indica una variable X entera, pues las operaciones de búsqueda en este tipo de lenguajes se realizan de tal forma que siempre localizan la última variable introducida.

4.2.2 Tablas de símbolos en pila

La organización más simple, desde el punto de vista conceptual, de una tabla de símbolos de un lenguaje estructurado en bloques es la pila. En este tipo de organización los registros que contienen los atributos de los identificadores se van colocando unos encima de otros según se van encontrando las declaraciones de las variables del texto fuente. Una vez que se lee el fin del bloque, todos los registros de los identificadores declarados en el bloque se sacan de la pila, pues dichos identificadores no tienen validez fuera del bloque.

En la Figura 4-16 se muestra una tabla de símbolos en pila para el programa anterior en el momento de finalizar la compilación del bloque 2.

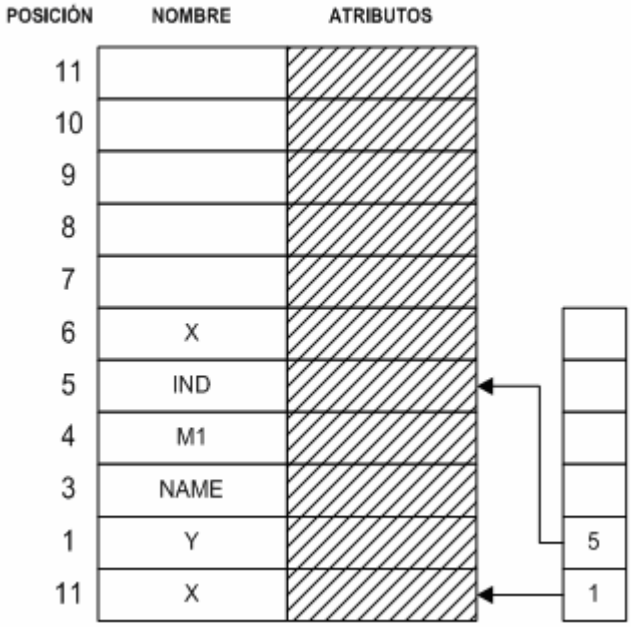


Figura 4-16. Tabla de símbolos en el momento de terminar la compilación del bloque 2

En la Figura 4-17 se muestra la tabla de símbolos cuando se finaliza la compilación del bloque 4.

La operación de inserción es muy simple en una tabla de símbolos en pila. Los nuevos registros se colocan en la cima ("*TOP*") de la pila. Las declaraciones de variables con le mismo nombre dentro de un mismo bloque se detectan antes de la inserción por medio de una búsqueda lineal entre los registros existentes.

La operación de búsqueda requiere una investigación lineal desde la cima ("*TOP*") hasta la base ("*BOTTOM*"). La búsqueda se realiza por orden (desde la cima de la pila) para garantizar que la última aparición de la variable con le nombre buscado se la primera encontrada.

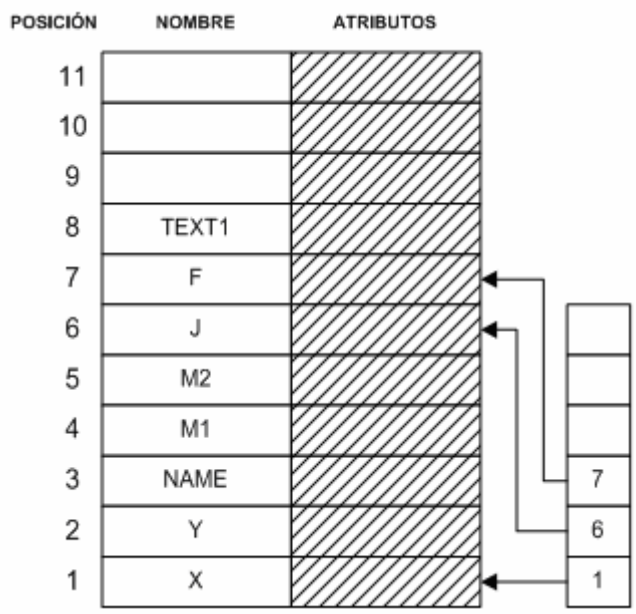


Figura 4-17. Tabla de símbolos en el momento de terminar la compilación del bloque 4

Por ejemplo en la Figura 4-16, la variable X (es decir la variable INTEGER:X) se encuentra la primera.

Nótese que la longitud media de investigación de una tabla de símbolos en pila puede ser menor que la correspondiente a una tabla de símbolos no ordenada, pues la operación *reset* deja solamente las variables del bloque en curso.

La operación *set* en una tabla de símbolos en pila genera un índice del bloque ("*BLOCK INDEX*") que representa el lugar donde está almacenado el primer registro de una variable del bloque y que en el momento de la operación *set* corresponde a la cima ("*TOP*") de la pila.

La operación *reset* saca de la pila todos los registros de las variables que se acaban de compilar en el bloque finalizado, para lo cual utiliza el índice del bloque finalizado, determinando con él hasta qué registro ha de sacar desde la cima de la pila ("*TOP*"). Los registros sacados de la tabla de símbolos pueden guardarse en una tabla inactiva.

La operación reset también elimina el índice del bloque finalizado de la pila auxiliar.

El parecido entre una tabla de símbolos no ordenados y una tabla de símbolos en pila es obvio, pero desgraciadamente esta similitud también se observa en las malas características, por lo que sólo se usan en casos muy especiales.

4.2.3 Tablas de símbolos con estructura de árbol implementadas en pilas

En este apartado se estudiará cómo las tablas con estructura de árbol se pueden utilizar en los lenguajes de bloques anidados. Las tablas de símbolos con estructura de árbol se pueden organizar de dos formas:

a) La primera ya se estudió en el apartado 4.1.3 y la única diferencia está en el borrado o eliminación de los registros al finalizar cada bloque. Esto es un gran problema pues todos los registros están mezclados en un mismo árbol y los pasos necesarios para borrar un registro son:
    - Localizar la posición del registro en la tabla.

- Sacar el registro y cambiar los punteros de las ramas del árbol necesarias.
- Reequilibrar el árbol.

Tal como se vio anteriormente esto necesita del orden de $\log_2 n$ operaciones y hay que hacerlo para cada bloque que se termina. Es evidente que no es una buena organización para un lenguaje con estructura de bloques.

b) La segunda, se podría llamar bosque, en ella cada bloque se organiza como una tabla estructurada en árbol y se elimina dicho árbol cuando se finaliza el bloque.

A continuación se muestran las figuras representativas de este tipo de organización para el ejemplo de programa de los apartados anteriores. Se muestra el estado de la tabla en el momento de finalizar la compilación de los bloques 2 (Figura 4-18) y 4 (Figura 4-19).

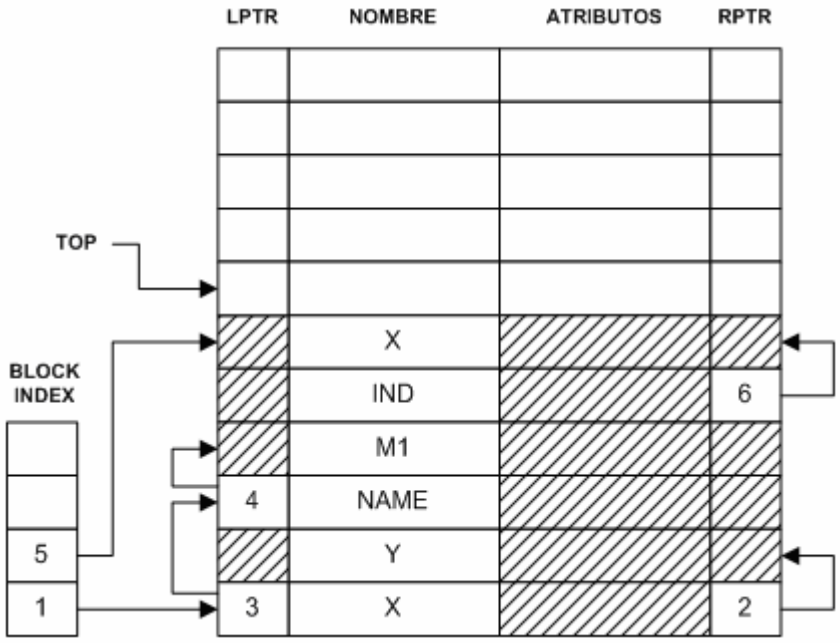


Figura 4-18. Tabla de símbolos con estructura de árbol implementada en pila después de la compilación del bloque 2

En este tipo de organización, los elementos del campo de índices de bloque ("*block index*") señalan los nodos raíz de cada uno de los árboles de la tabla. El puntero izquierdo (LPTR) y el puntero derecha (RPTR) indican los encadenamientos de los árboles AVL.

Nótese que en el proceso de compilación del bloque 3 se encuentra la declaración del procedimiento M2 y además M2 es accesible desde el bloque 1. La inserción de M2 en el árbol del bloque 1, obliga a reequilibrar dicho bloque utilizando la doble rotación, mostrándose el resultado en la Figura 4-19.

Se puede observar que la tabla de símbolos se maneja como una pila. Cuando se detecta una variable nueva en un bloque se almacena en la cima de la pila y el campo de los punteros izquierdo y derecho se ajusta a la estructura de árbol de su bloque.

La operación de búsqueda dentro de un bloque se realiza aprovechando la estructura de árbol y se garantiza la no confusión con las variables del mismo nombre al haber un árbol para cada bloque.

La operación *set* genera el nuevo árbol cuya raíz se localiza con el índice del bloque.

La operación *reset* saca el árbol en curso hacia un área inactiva. Una alternativa a esto es dejar el árbol donde está, marcando esta zona como inactiva y modificando el puntero que indica la cima de bloque.

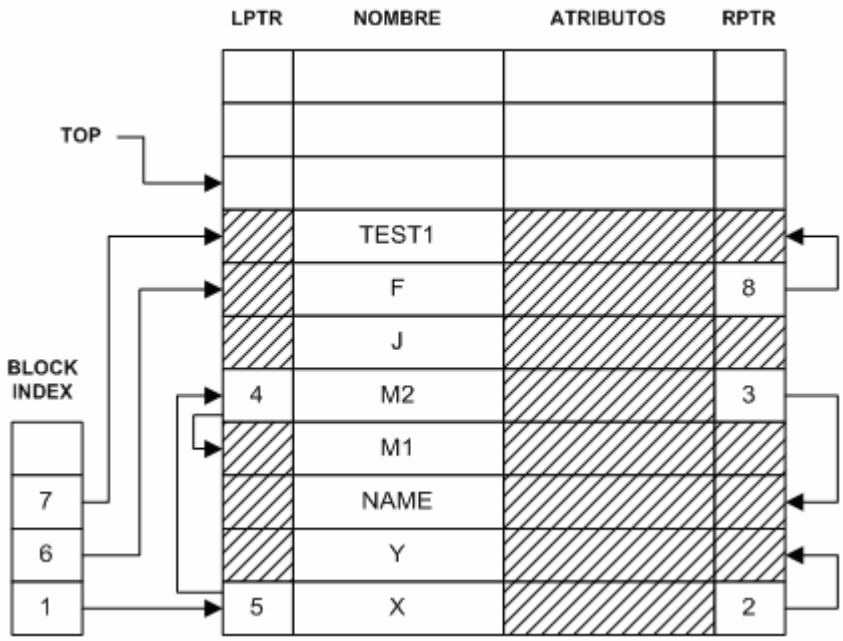


Figura 4-19. Tabla de símbolos con estructura de árbol implementada en pila después de la compilación del bloque 4

Este tipo de organización de la tabla de símbolos se utiliza cuando la memoria de que se dispone para realizarla es pequeña.

4.2.4 Tablas de símbolos con estructura hash implementadas en pilas

A primera vista parece imposible utilizar transformaciones de claves para lenguajes estructurados en bloques, pues las variables de un bloque deben de estar agrupadas y las funciones hash no preservan ningún orden.

En este apartado se utilizarán funciones hash que usan la técnica de hash abierto para resolver los problemas de colisiones, por ser en general más ventajoso tal como se estudió anteriormente. Utilizaremos el mismo ejemplo que en casos anteriores, suponiendo que la función hash realiza las siguientes transformaciones:

Nombre del Identificador	Dirección
X	1
M1	1
NAME	3
IND	5
J	5
TEST1	6
F	8
Y	8
M2	11

Se utiliza un array de punteros de tamaño 11 como se puede ver en la Figura 4-20.

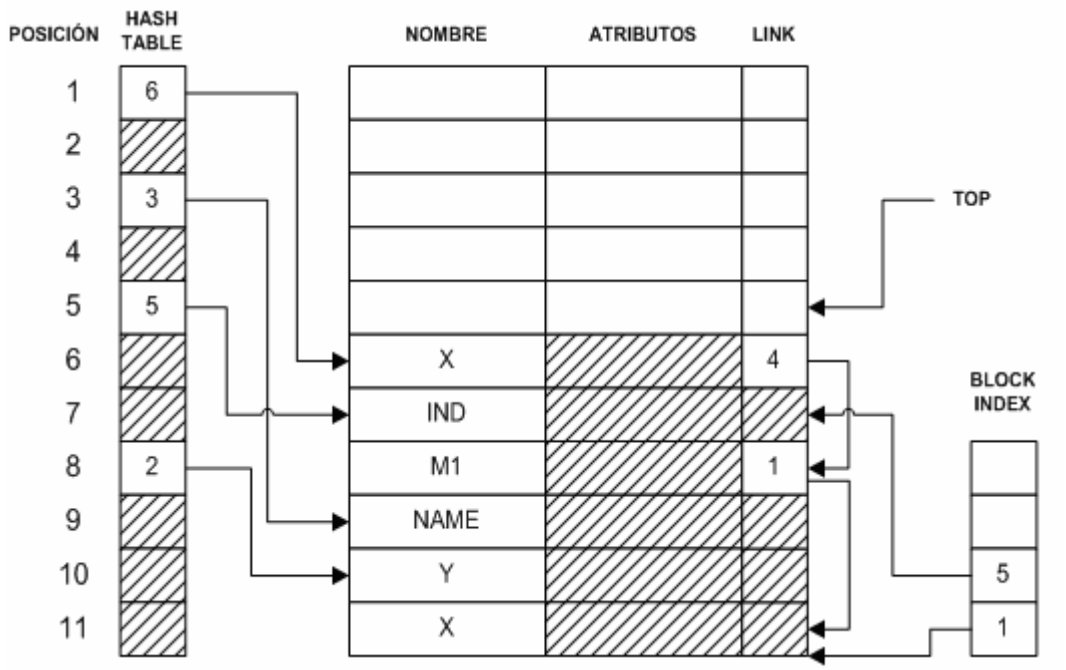


Figura 4-20. Tabla de símbolos con estructura hash implementada en pila después de la compilación del bloque 2

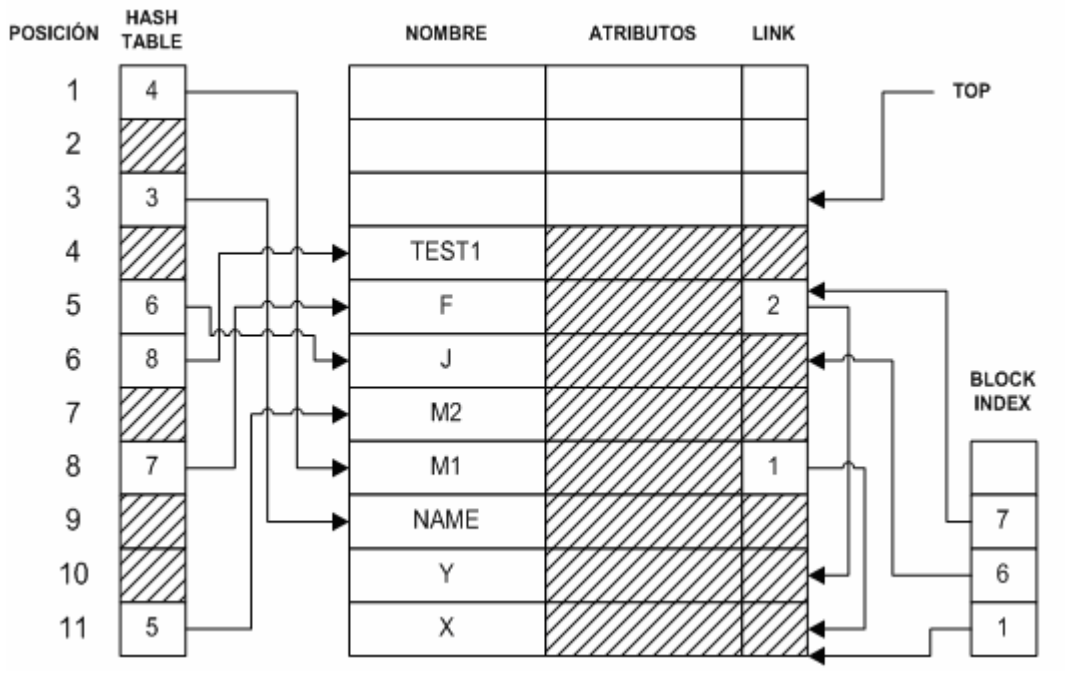


Figura 4-21. Tabla de símbolos con estructura hash implementada en pila en el momento anterior a la terminación de la compilación del bloque 4

La Figura 4-20 muestra la tabla de símbolos en el instante anterior a que se complete el análisis del bloque 2. Otra vez los nuevos registros de las nuevas variables se colocan en la cima de la pila y el campo de índices de bloques ("*BLOCK INDEX*") señala el registro donde comienzan los bloques actualmente activos.

En este ejemplo se puede observar que ocurre cuando se duplica el nombre de una variable en bloques diferentes. Es el caso de la variable X cuando ésta aparece en el bloque 2. El registro del identificador M1 se encadena al registro de la variable X

(declarada en el bloque 1) y la situación de M1 está señalada por el puntero de la X recién introducida. La tabla hash señala a la última X introducida.

Cuando se produce una colisión su resolución de lleva a cabo de esta manera.

La Figura 4-21 muestra la tabla de símbolos en el instante anterior a que se complete el análisis del bloque 4.

Las operaciones de inserción y búsqueda se realizan de forma similar a como se vieron en el apartado 4.1.4, excepto en las particularidades que se producen cuando un nombre de identificador se duplica.

La operación *set* se realiza como en los apartados precedentes. La operación *reset* es la más complicada, tiene bastantes problemas, pues al sacar un bloque se puede eliminar algún campo de encadenamiento utilizado para resolver alguna colisión.

Es preferible, a veces, desactivar esa zona pero manteniendo los valores de los punteros de encadenamiento.

Un esquema general de pilas de tablas hash abiertas se muestra en la Figura 4-22. Este tipo de esquema tiene una ocupación de memoria mayor, pero es más sencillo de implementar.

Hoy día la mayor parte de los lenguajes orientados a objetos (Java, C++, etc.) poseen librerías de contenedores que, aunque ocupen más memoria, facilitan la creación de este tipo de estructuras enormemente, es el caso de los vectores, listas, tablas hash, árboles, etc., que con diferentes nombre se incluyen para las bibliotecas de contenedores de estos lenguajes (*map*, *vector*, *list*, *HashMap*, *ArrayList*, etc.).

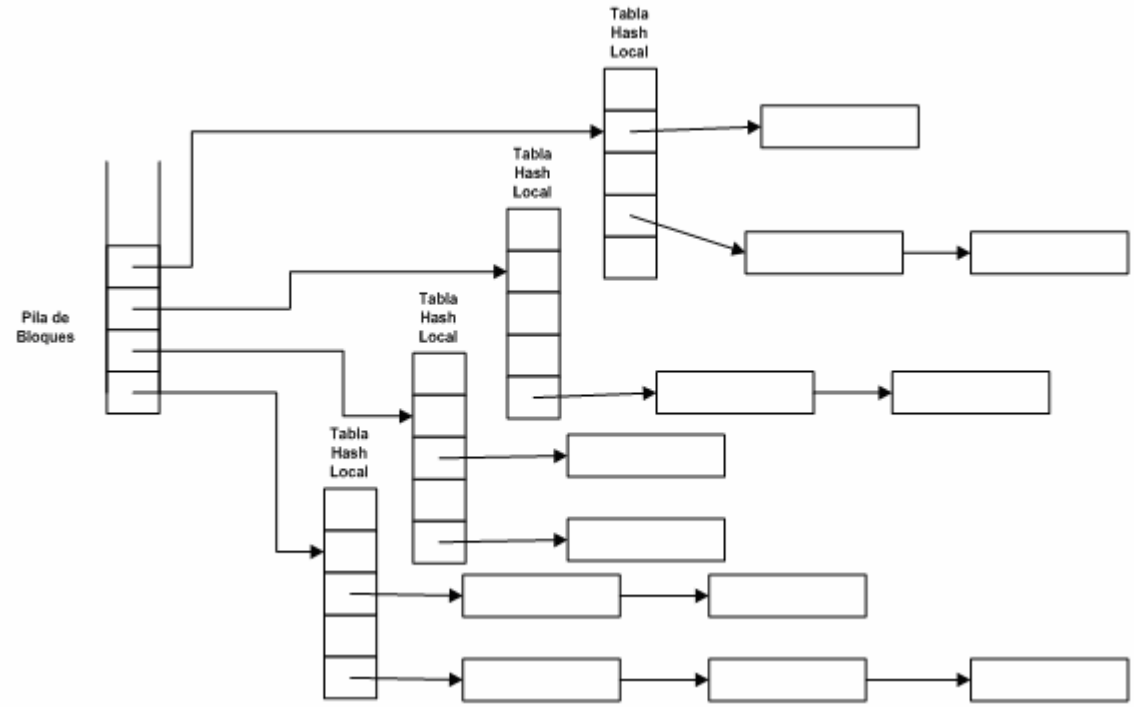


Figura 4-22. Esquema general de pilas de tablas hash abiertas

4.3 Representación OO de símbolos y tipos en compiladores de una pasada

En los sistemas orientados a objetos aparecen nuevos problemas a la hora de implementar la tabla de símbolos que se estudian en los siguientes apartados.

En estos momentos hay dos lenguajes implicados que pueden ser orientados a objetos: el lenguaje a compilar y el lenguaje de implementación del compilador, en este caso el lenguaje de implementación de la propia tabla de símbolos (ver Figura 4-23).

En los siguientes párrafos se considera que el lenguaje de implementación es orientado a objetos, esto proporciona beneficios de implementación de la TS. Todo lo estudiado en este apartado es aplicable para el caso de lenguajes fuente estructurados (también orientados a objetos). Queda fuera del alcance de este documento las TS de lenguajes OO implementados en lenguajes estructurados convencionales.

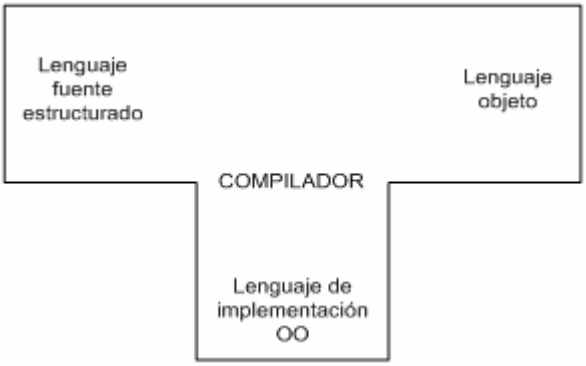


Figura 4-23. Esquema del compilador (T) en que se muestra que el lenguaje fuente y el de implementación son orientados a objetos

4.3.1 La doble jerarquía símbolo-tipo

Cuando estamos compilando un lenguaje orientado a objetos (OO), el esquema de tabla de símbolos debe representar en cierto modo el modelo del lenguaje fuente. En un esquema orientado a objetos tenemos una doble jerarquía:

- Símbolos, que son todos aquellos identificadores que representan variables locales, atributos, campos de un registro, parámetros de un método, etc.
- Tipos, que son todos los identificadores que representan tipos del lenguaje, entendidos aquí los tipos en un sentido amplio, esto es comprendiendo clases cuando se habla de lenguajes OO. (Ver [ORTIN04])

No es objetivo definir aquí lo que es un tipo y simplemente nos quedamos con el concepto de que en OO un tipo determina el tipo de mensajes que puede admitir un símbolo. Cuando dos símbolos son el mismo tipo se entiende que admite exactamente los mismos mensajes.

Así pues el esquema de la tabla de símbolos consta de dos tablas:

- **Tabla de símbolos**, que sirve para guardar una referencia a todos los símbolos que pueden ser accedidos desde el entorno actual. Esta tabla tiene un comportamiento muy similar a todas las tablas vistas hasta ahora, de hecho alguna de aquellas implementaciones podría funcionar perfectamente (por ejemplo una tabla de acceso *hash* implementada en pila).

  Como se puede comprobar en la Figura 4-24, las operaciones de la clase `SymbolTable` son las mismas que en las otras implementaciones de tabla de símbolos vistas hasta ahora.

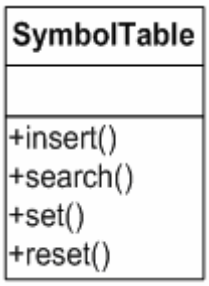


Figura 4-24. Clase que implementa la Tabla de Símbolos

- **Tabla de tipos**, que es la que guarda la estructura de tipos definida en el programa fuente: las clases, las interfaces, los tipos básicos, etc.

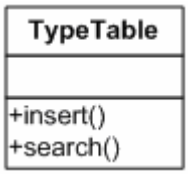


Figura 4-25. Clase que implementa la Tabla de Tipos

NOTA: Se puede discutir aquí si merece la pena o no poner una clase (facade) que unifique el método de acceso a la tabla de símbolos, la propuesta de este documento es que no es necesario y sólo sirve para proponer una clase con un interfaz demasiado grande y con métodos que están poco relacionados ente ellos (muy baja cohesión). Por otro lado, se demuestra que en la compilación de lenguajes OO convencionales (Java, C++, etc.) nunca hay confusión en la búsqueda de identificadores cuando se trata de un tipo y de un símbolo.

4.3.2 La jerarquía de símbolos

La jerarquía de símbolos describe todos los posibles símbolos que pueden aparecer en el código fuente.

A partir de la tabla de símbolos se accede a estos símbolos (representado aquí por una agregación, pero que en general será una acceso algo más complejo). Cada símbolo tiene tipo y debe poderse acceder a dicho tipo a través del símbolo.

El método de acceso puede ser representado como en este caso por una agregación o también podría estar representado por una relación cualificada por una cadena que identificaría el nombre del símbolo.

Este tipo de representación puede ser implementada por medio de una tabla de acceso hash, de manera que, como en un diccionario, se localizase la entrada del símbolo por su nombre.

Este tipo de implementación está recogida en la mayor parte de las bibliotecas de contenedores de los lenguajes orientados a objetos, map, hashmap, hashtable, etc. son algunos de los nombres habituales que recibe este tipo de contenedor dependiendo de los leguajes de implementación.

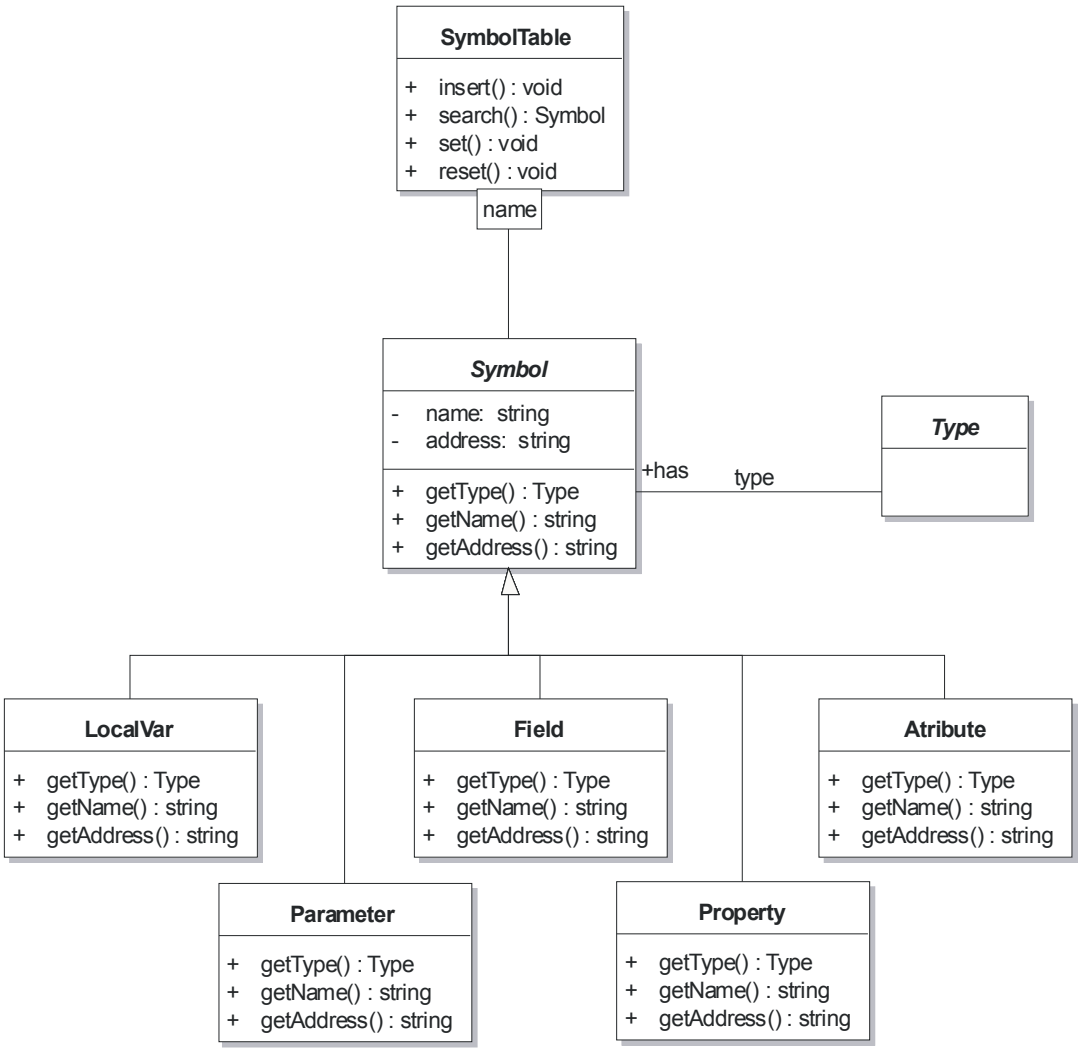


Figura 4-26. Jerarquía de símbolos

Obsérvese que en la Figura 4-26 se ha hecho un esquema muy genérico de tabla de símbolos que prevé que cada uno de los posibles símbolos tenga un comportamiento y unas responsabilidades diferentes.

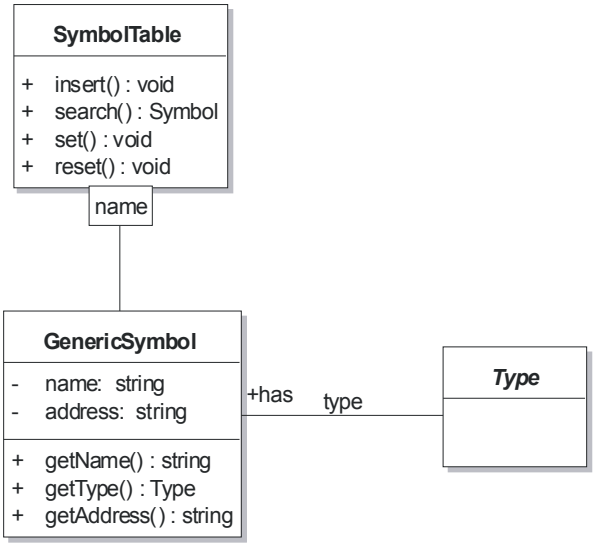


Figura 4-27. Versión simplificada de la jerarquía de símbolos

En general esto no es así, el comportamiento puede quedar perfectamente recogido en los tipos y los símbolos no tienen casi información particular, simplemente sirven para determinar la visibilidad desde un punto concreto del código.

En estos casos se puede hacer una simplificación del esquema anterior como puede apreciarse en la Figura 4-27, por ejemplo se puede guardar en la dirección (que únicamente se usa para la generación de código) un *string* con la dirección a generar directamente y no un valor entero que indique un desplazamiento y que hay que convertir en *string*, en cualquier caso, al generar código.

4.3.3 La jerarquía de tipos

En paralelo a la jerarquía de símbolos tenemos una jerarquía de tipos donde pueden estar todos los tipos básicos del lenguaje y los tipos definidos por el usuario (con *typedef*, clases, etc.).

Un esquema bastante general se puede ver en la Figura 4-28.

En este caso se tiene una tabla de tipos (*TypeTable*) que tiene guardados todos los tipos que aparecen durante la compilación del código fuente.

Al comenzar dicha compilación estarán únicamente los tipos básicos del sistema (los *BuiltinType*). En el momento en que el código comienza a describir clases, arrays, métodos, interfaces, etc. Va creando los nuevos tipos y metiéndolos en la tabla.

Hay que hacer notar que la clase *BuiltinType* representa a cada uno de los tipos básicos del lenguaje, esto es, no existirá nunca una clase *BuiltinType* pero si existirán las clases *IntType*, *DoubleType*, *BooloeanType*, *VoidType*, etc.

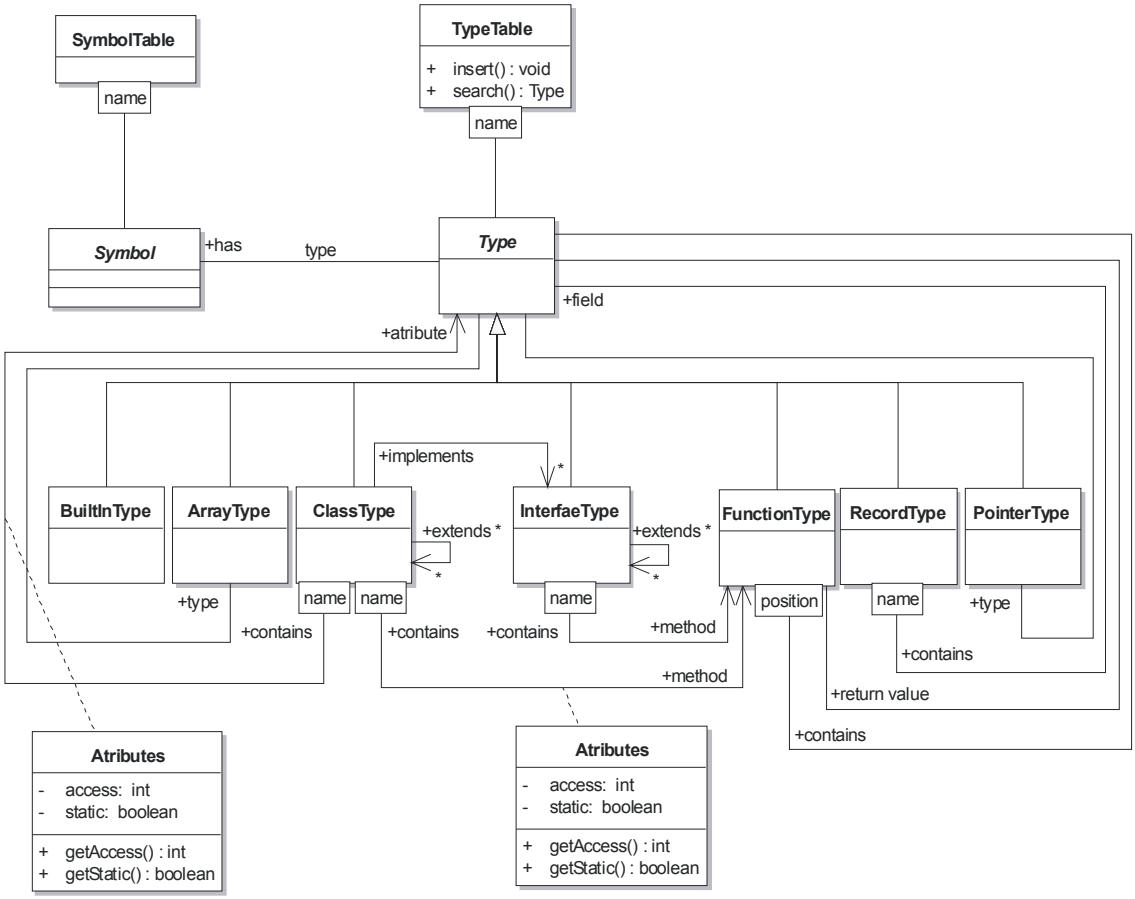


Figura 4-28. Jerarquía y tabla de tipos

En esta tabla se guarda la estructura completa de las clases (sus atributos, sus propiedades, sus métodos, etc.) con todas sus etiquetas de accedo (*private*, *public* y *protected*) o su pertenencia (*static* o pertenencia a la clase y pertenencia a los objetos).

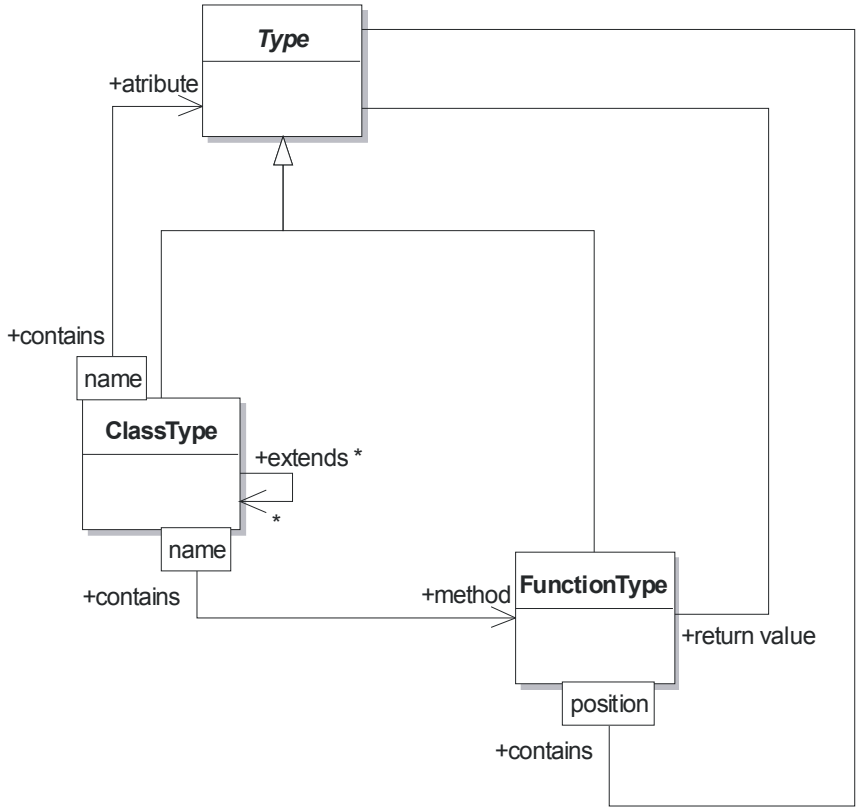


Figura 4-29. La clase contiene métodos y atributos

Cuando hay que guardar información adicional en el atributo o método (*static*, *private*, etc.) se crea una asociación atribuida con la información necesaria

Dependiendo de la cantidad de información y de la estructura compleja que el lenguaje permita a las clases, la clase *TypeTable* tendrá más o menos métodos especializados de inserción y de búsqueda.

Desde cada símbolo en la *SymbolTable* se apuntará a uno de los tipos de esta tabla (que definirá su tipo).

La definición de tipos compuestos de otros tipos (por ejemplo una clase que contiene atributos y métodos) se ha definido mediante una asociación cualificada por el nombre del atributo, definiendo que su acceso se hará a través de una clave que será el nombre del atributo. En el caso de la sobrecarga y siendo un método, esta etiqueta no tiene por qué ser única.

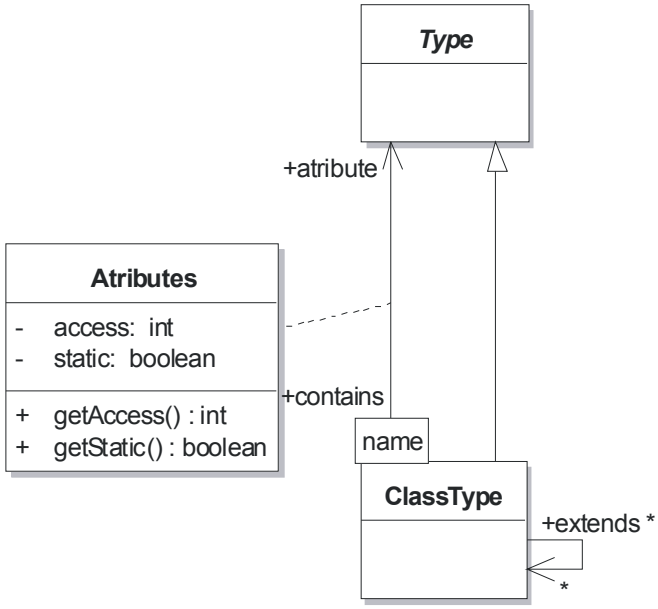


Figura 4-30. Relación cualificada y atribuida para determinar el componente de la clase y los atributos.

Cuando hay que guardar información adicional en el atributo o método (*static*, *private*, etc.) se crea una asociación atribuida con la información necesaria

Por otro lado cada función tiene una relación de tipos que son sus parámetros y una relación directa a su tipo de valor de retorno (que puede ser *void*).

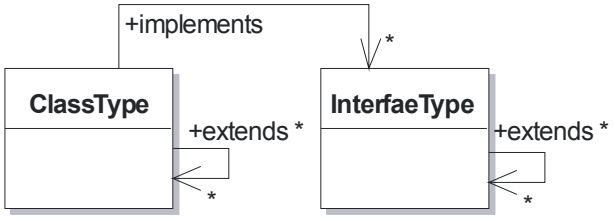


Figura 4-31. Herencia

La herencia se implementa mediante interfaces y clases de manera que las interfaces pueden heredar entre ellas y las clases pueden heredar de otras clases o implementar interfaces. Obsérvese, además, que la cardinalidad de las relaciones permitiría implementar herencia múltiple (Ver Figura 4-31).

4.3.4 Esquema general de la tabla de símbolos

Juntado todos los esquemas anteriores obtenemos el Diagrama de Clases de la Figura 4-32 en el que puede verse el aspecto general de una tabla de símbolos.

Se ha usado el esquema de tabla de símbolos más general, ver párrafos anteriores.

Este esquema puede utilizarse para obtener una simplificación adecuada y adaptada a un compilador concreto.

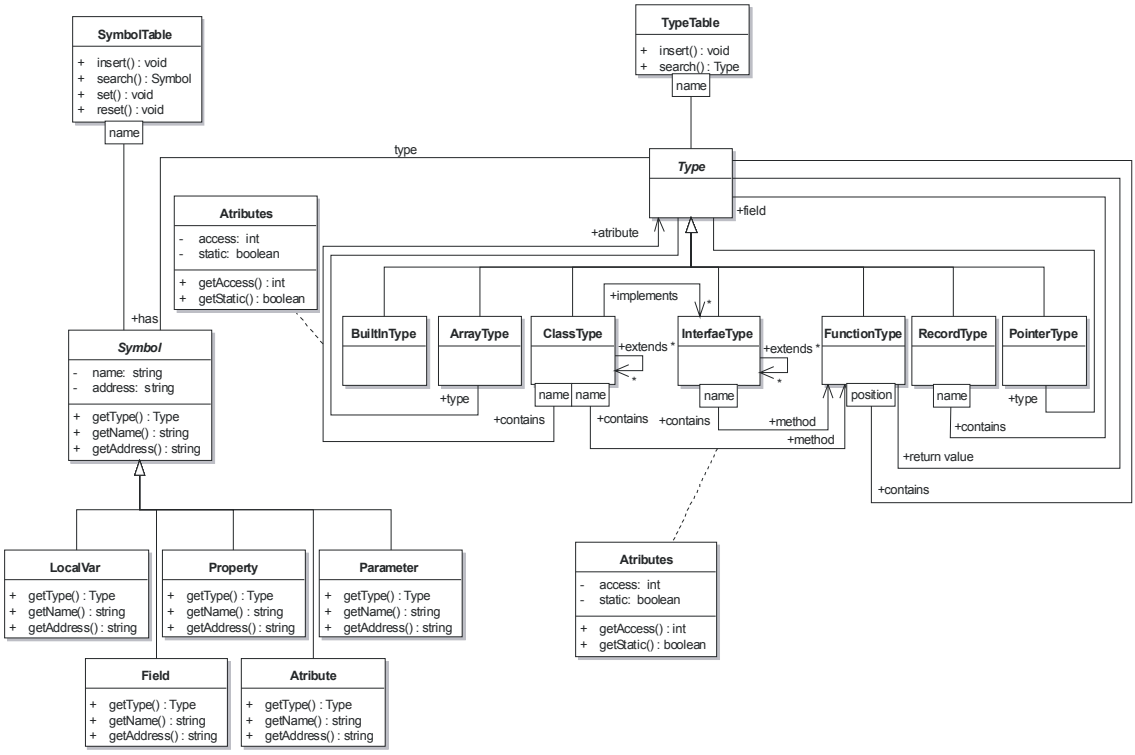


Figura 4-32. Es quema general de una tabla de símbolos con las dos jerarquías diferenciadas. (Ver Anexo 1)

Las dos relaciones de agregación: `SymbolTable` a `Symbol` y `TypeTable` a `Type`, se pueden definir también como cualificadas a través de una *string* que sería el nombre del identificador en el contendor correspondiente. No se ha puesto así por claridad del gráfico.

4.3.5 Funcionamiento y acceso a la tabla de símbolos

Sea un lenguaje que tiene los tipos básicos *void*, *int* y *double* y una estructura en clases similar al Java sin interfaces y con herencia simple.

Se trata de hacer un compilador de este tipo de código y, en este caso, hacer la tabla de símbolos de dicho compilador y demostrar que es suficiente para todo el proceso de compilación.

A partir de un código se irá siguiendo toda la formación de la tabla de símbolos y se podrá comprobar cómo funciona el modelo.

En los siguientes párrafos se muestra el trozo de código que se desea compilar para ver como funciona su tabla de símbolos.

```
1:    forward class TabSimb
2:       {
3:       public int procesa (int k);
4:       public static void main ();
5:       }
6:
7:    class TSdos
8:       {
9:       private static int r;
10:
11:      public static void m (TabSimb ts, int g)
12:         {
```

```
13:          ts.procesa(g);
14:          }
15:       }
16:
17:    public class TabSimb
18:       {
19:       double p;
20:
21:       public int procesa (int k)
22:          {
23:          int m;
24:          double d = k * m;
25:          p = 2 * d;
26:          return d;
27:          }
28:
29:       public static void main ()
30:          {
31:          TabSimb t = new TabSimb();
32:          TSdos.m(ts,7);
33:          }
34:       }
```

El esquema que se necesita de tabla de símbolos está simplificado con respecto al general que se ha visto anteriormente.

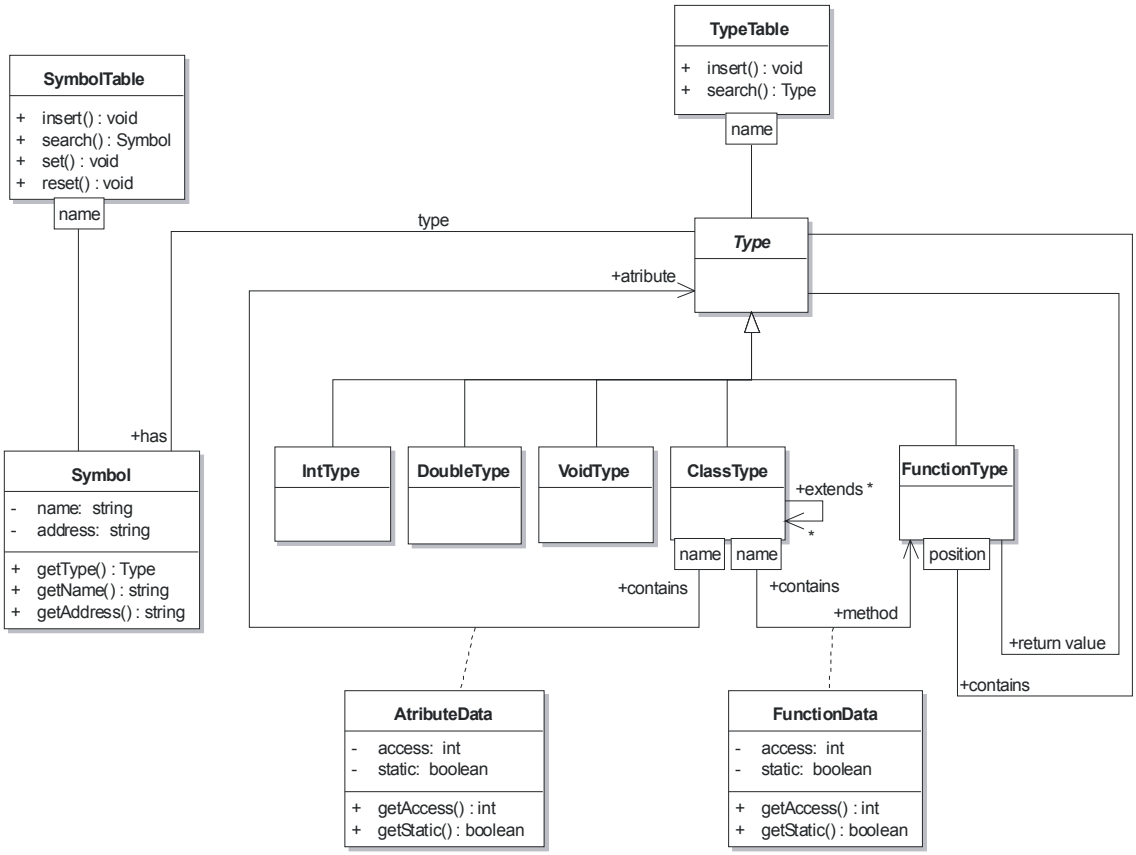


Figura 4-33. Tabla de símbolos simplificada para el ejemplo. (Ver Anexo 2)

Por ejemplo, se ha quitado el tipo *BuiltinType* y ha sido sustituido por los tipos *IntType*, *DoubleType* y *VoidType* que son los tipos básicos del sistema.

No hay interfaces y la herencia es simple, por tanto sólo cabe una relación de una clase a muchas, ya que una clase sólo puede heredar como máximo de otra, pero cada clase puede ser heredada por múltiples clases. El nuevo esquema puede verse en la Figura 4-33.

Antes de comenzar la compilación se crean dos objetos: *mySymbolTable:SymbolTable* y *myTypeTable:TypeTable*.

Inmediatamente después se crean los tipos básicos: *IntType*, *DoubleType* y *VoidType* y se insertan en la tabla de tipos, por tanto al principio de la compilación el esquema es como se puede ver en la Figura 4-34.

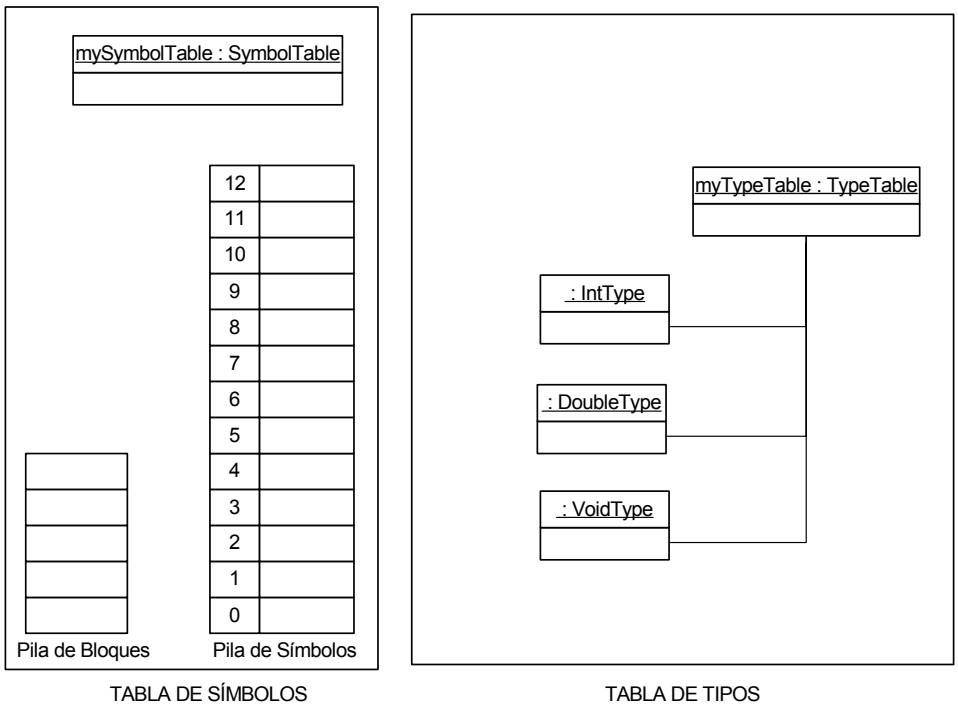


Figura 4-34. Situación inicial de la tabla de símbolos

Comienza la compilación y se encuentra una definición (*forward*) de la clase `TabSimb` que se implementará con posterioridad.

Esta definición completa dará de alta un nuevo tipo en la tabla de símbolos, como se puede ver en la Figura 4-35.

Como se puede apreciar en dicha figura, se han dado de alta dos métodos (en el *forward* sólo se declaran los elementos públicos) y se han creado los objetos de las relaciones atribuidas donde se han colocado los datos referentes a dichos métodos.

Los atributos de esta clase `TabSimb`, se crearán en la tabla de símbolos durante la compilación de dicha clase.

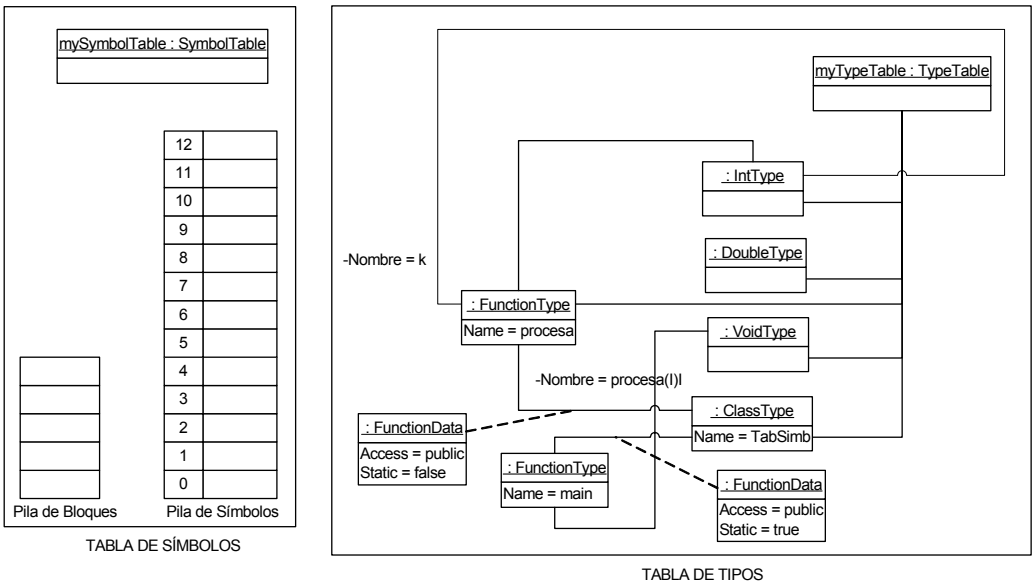


Figura 4-35. Situación compilando en la línea 4

Al compilar la línea 7 se identifica el token *class* y se ejecuta *set* en la tabla de símbolos, seguidamente el nombre de la clase `TSdos` y se crea un nuevo objeto de la clase *ClassType* para contener este tipo y se da de alta como tipo en la tabla de tipos.

La ejecución del comando *set* provoca que se cargue en la *Pila de Bloques* la dirección del puntero al tope de la *Pila de Símbolos* lo que marca que a partir de este momento los símbolos que aparezcan pertenecerán a este ámbito.

La implementación de la tabla de símbolos de la figura podría ser una tabla con direccionamiento hash implementada en pila.

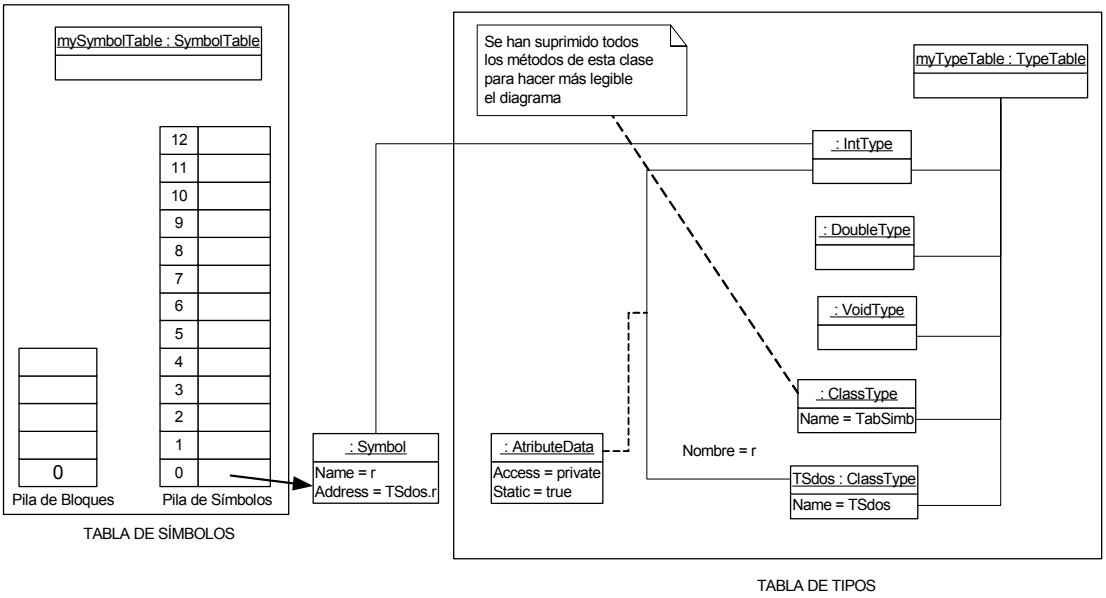


Figura 4-36. Situación después de compilar la línea 9

Seguidamente se identifica el `private static int r` (línea 9) y se da de alta un nuevo símbolo en la tabla de símbolos, con su tipo apuntando hacia el tipo `int` de la tabla de tipos. En la Figura 4-36 se puede ver la situación de la tabla de símbolos después de la compilación de la línea 9.

Al mismo tiempo se da de alta un atributo en la tabla de tipos, dentro del símbolo que identifica la clase `TSdos`.

A continuación se compila la línea 11 y se encuentra un método que debe ser dado de alta en la tabla de tipos como parte de la clase `TSdos`.

Al mismo tiempo, en la tabla de símbolos se ejecuta un nuevo *set*, esta vez por ser el comienzo de un nuevo método y se crean dentro los dos parámetros.

Estos parámetros se dan de alta, además, en la tabla de tipos como parte del método `TSdos.m`. En la Figura 4-37 se puede ver la situación de la tabla de símbolos después de compilada la línea 11.

En la línea 13 se encuentra "`ts.`" y al buscarlo en la tabla de símbolos se encuentra que es un objeto de la clase `TabSimb`. Después, al encontrar la llamada al método `procesa()`, se busca este a través de la relación tipo de *:Symbol* que apunta a la entrada `TabSimb` en la tabla de tipos.

Al llegar la compilación a la línea 14 se ejecuta el *reset* y se desapila el último bloque anidado que corresponde al ámbito del método `m` de la clase `TSdos`. Esto elimina los parámetros `ts` y `g` de la tabla de símbolos, pero el símbolo `r` y el nombre del método `m` siguen estando activos.

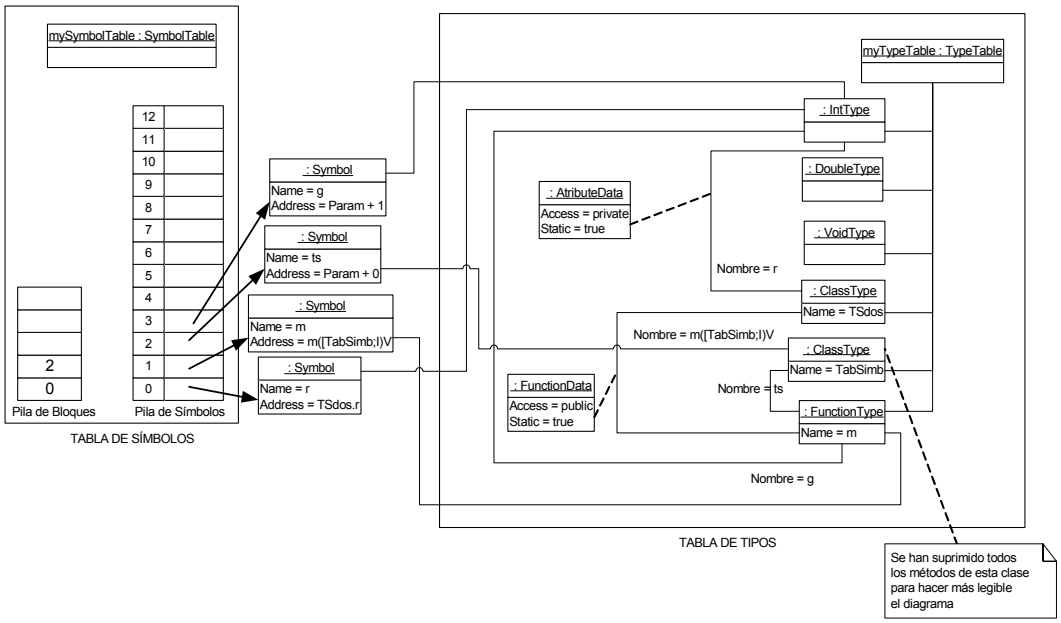


Figura 4-37. Situación de la Tabla de Símbolos después de compilada la línea 11

Por último, en la línea 15 se cierra el ámbito de la clase `TSdos` y se ejecuta de nuevo el reset, de manera que de nuevo se elimina un bloque y en este caso se eliminan el atributo `r` y el método `m`, aunque siguen accesibles a través de la clase en la tabla de tipos.

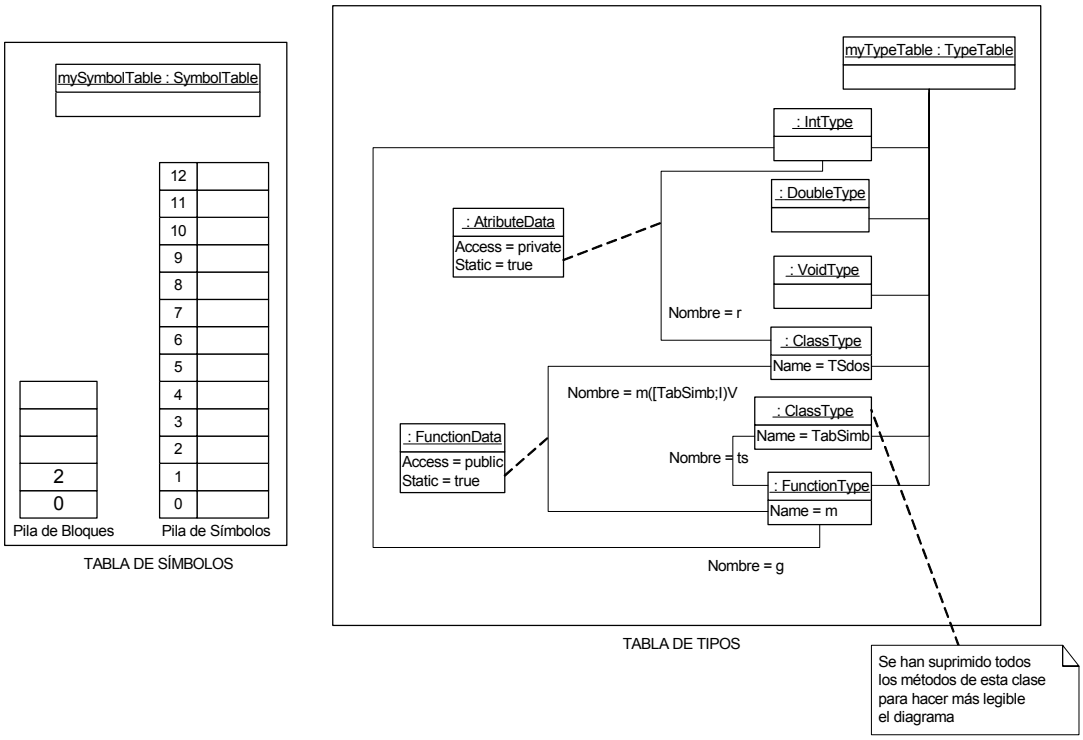


Figura 4-38. Estado de la tabla de símbolos después de compilada la línea 15

Ahora se comienza con la compilación de la siguiente clase (`TabSimb`), esta clase había sido introducida en la tabla de tipos previamente mediante una instrucción `forward`. Este hecho debería quedar marcado para no confundir cuando se ha hecho una declaración de este tipo y cuando se intenta redeclarar la clase.

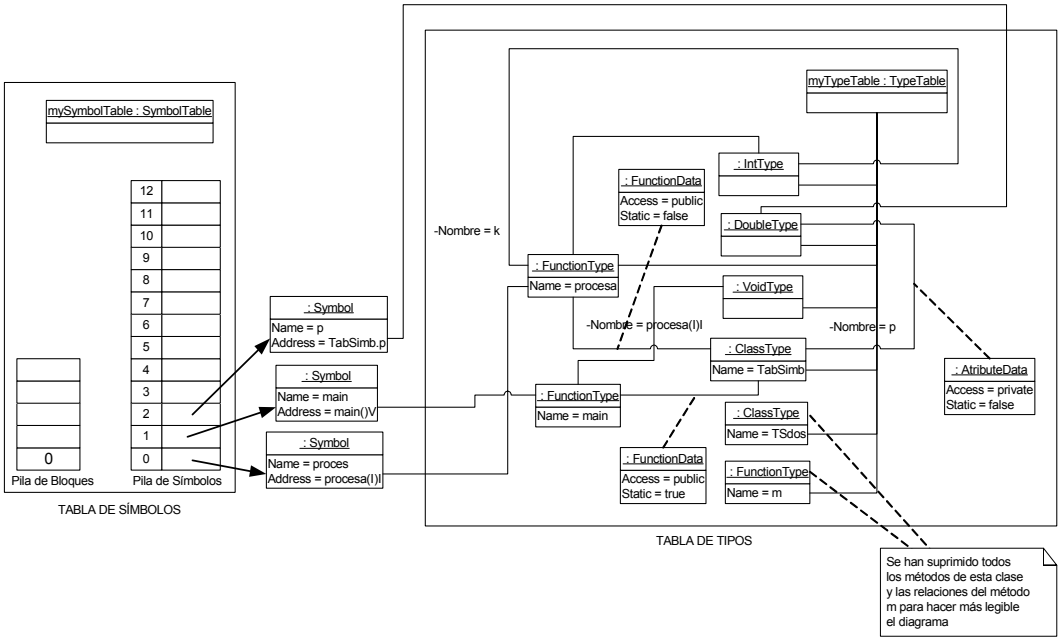


Figura 4-39. Estado de la tabla de símbolos después de compilar la línea 19

Esto produce una nueva llama a *set* y se crea un ámbito para la clase. Además se introducen en la tabla de símbolos los nombres de los métodos ya declarados en el `forward` (`procesa` y `main`).

En la línea 19 se introduce un nuevo atributo, esta vez los parámetros no son explícitos y se deberá decidir cuáles son los que queremos poner (por ejemplo `private` y no

`static`). Así pues se introduce este atributo en la tabla de símbolos y en la de tipos con estos modos de acceso.

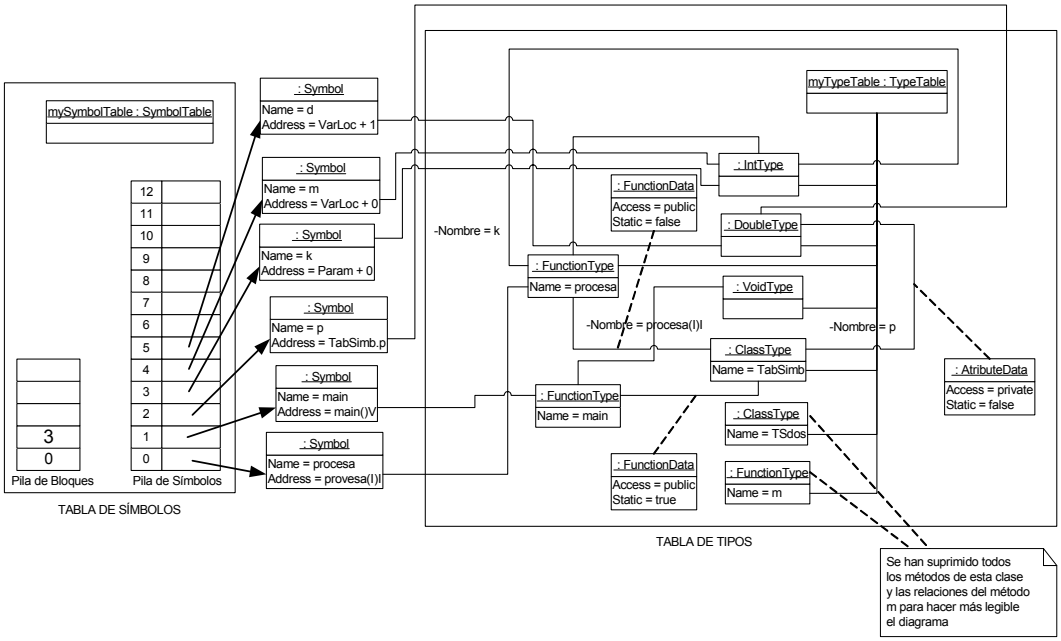


Figura 4-40. Estado de la tabla de símbolos al compilar la línea 25

En la línea 21 se encuentra el método procesa (ya dado de alta en el `forward`) y ahora se ejecuta el *set* para abrir un nuevo ámbito.

Dentro de este ámbito, estarán sus parámetros (`k` en la línea 21) y sus variables locales (`m` en la línea 23 y `d` en la línea 24).

En la Figura 4-40 se puede ver el aspecto de la tabla de símbolos cuando se está compilando la línea 25.

Al legar a la llave de la línea 27 se ejecuta el *reset* y se eliminan el parámetro y las variables locales del método `procesa`, aunque este nombre permanece en el ámbito de la clase.

En la línea 29 comienza el método `main`, se vuelve a ejecutar el *set* para crear un nuevo ámbito en la tabla de símbolos. En este caso, en la línea 31 se introduce el símbolo `t`.

En la Figura 4-41 se puede ver el aspecto de la tabla de símbolos después de haber sido compilada la línea 31.

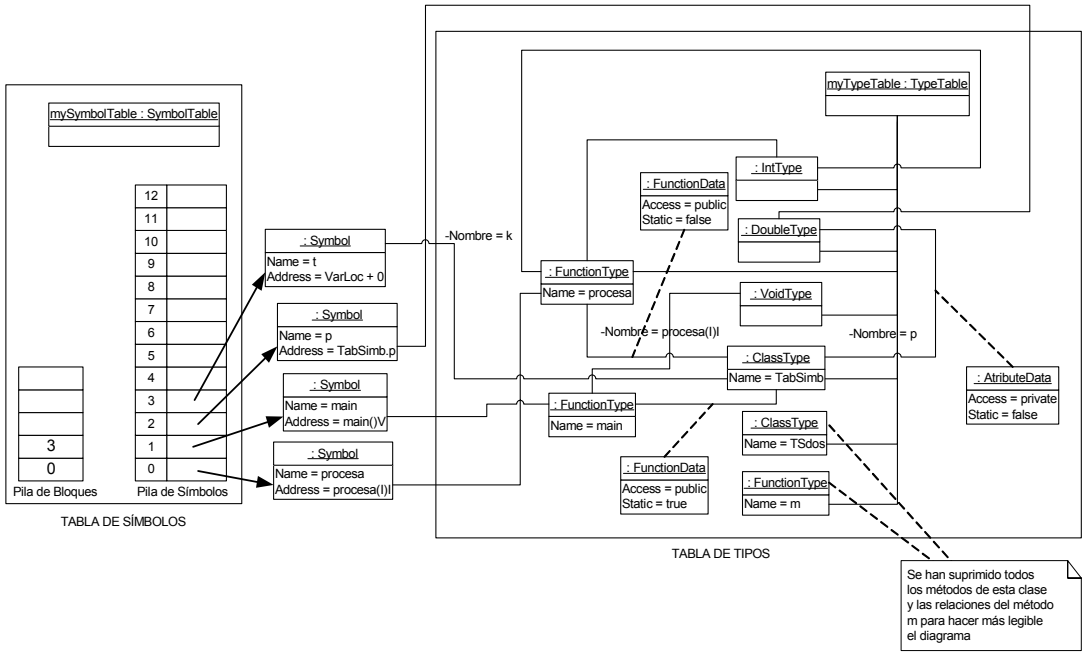


Figura 4-41. Estado de la tabla de símbolos después de compilada la línea 31

En la línea 33 se cierra este ámbito y desaparece la variable local t. Después en la línea 34 se cierra la llave de la clase y se cierran todos los ámbitos. Por último se termina el fichero y por tanto la compilación.

Toda la compilación anterior se ha realizado usando como modelo un compilador de una pasada. Este tipo de compilación obliga a la declaración adelanta (*forward*) de la clase `TabSimb`. Se hubiera podido utilizar un modelo de varias pasadas, en este caso la primera pasada introduciría en la tabla de símbolos el nivel 0 de cada clase, de manera que en las sucesivas pasadas ya estarían declaradas las clases y métodos necesarios para la compilación.

4.4 Tablas de símbolos OO en compiladores de varias pasadas

Cuando se utilizan compiladores de varias pasadas sobre un árbol AST, la TS puede ser utilizada sólo en algunas pasadas (las primeras sobre el AST) y su utilización se reduce a la de guardar los contextos hasta que éstos son resueltos directamente en el AST.

Una vez el AST ha sido decorado con esta información (y quizás alguna referencia a algún objeto externo) la TS puede eliminarse y su información puede recuperarse desde el propio árbol generado y decorado.

Las jerarquías de tipos y símbolos en este caso son similares a las del apartado 4.4 y las mismas que están referencias en los anexos: Anexo 1. Diagrama General de Clases y Anexo 2. Tabla de símbolos simplificada para el ejemplo.

4.4.1 Modelos de descripción sobre el AST

La información de la tabla de símbolos es, fundamentalmente la de símbolos y tipos. Y esa información debe quedar reflejada en el AST correspondiente. Hay dos maneras de tratar al símbolo: manteniendo un objeto símbolo o integrando su información en propio nodo del AST.

Sea el siguiente código de ejemplo de un lenguaje sencillo que sólo permite dos tipos básicos (`int` y `double`) y un código principal en que se pueden hacer lecturas, escrituras y expresiones:

```
1:    int i;
2:    float f;
3:    main ()
4:       {
5:       int f;
6:       read f;
7:       i = f;
8:       write I + 2;
9:       }
```

Una vez generado el árbol AST (usando la descripción de nodo que se puede encontrar en el Anexo 3. Descripción de los nodos del AST (Ejemplo primero de varias pasadas)) quedaría:

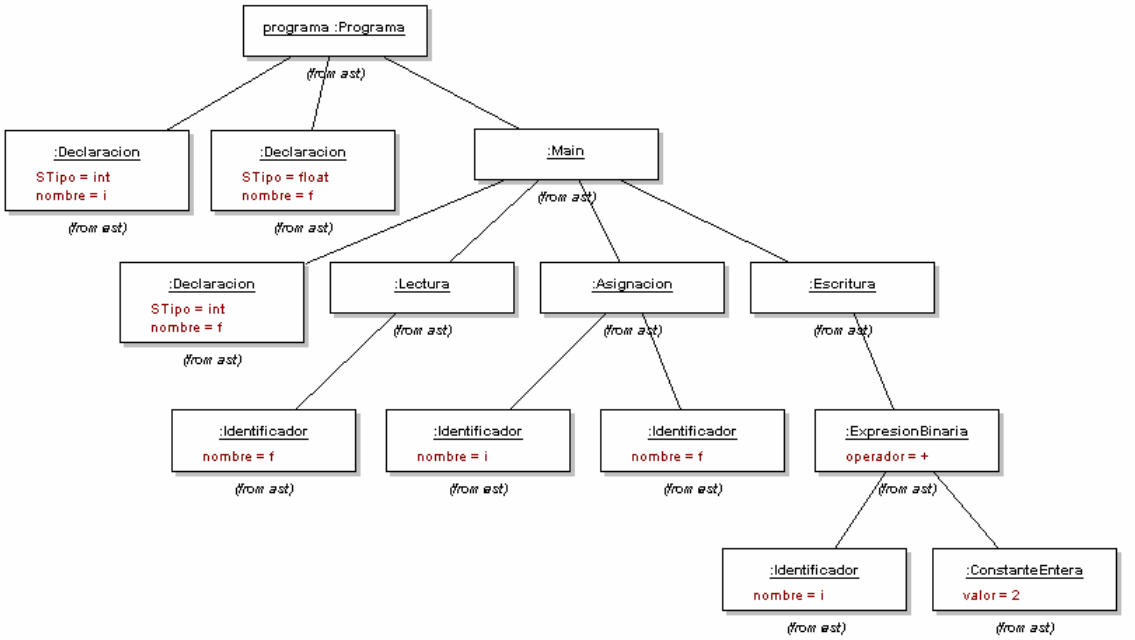


Figura 4-42. AST del código de ejemplo

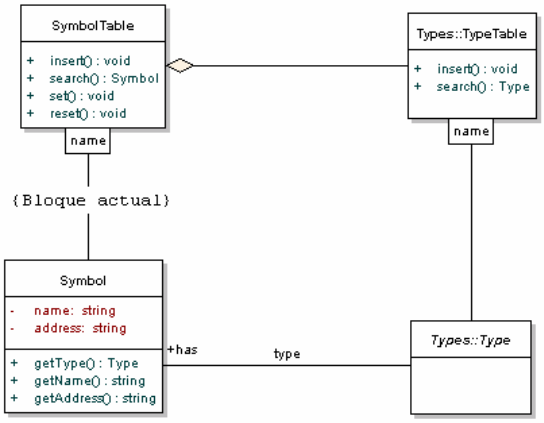


Figura 4-43. Estructura de clases de la TS

Ahora se lanzaría el objeto *visitor* `Identificador` que crearía las referencias y objetos necesarios para poder obtener el contexto en posteriores pasadas (Segunda pasada si

consideramos al análisis sintáctico como la primera pasada y primera pasada con un *visitor*).

Para ello se crearía una tabla de símbolos que contendría una tabla de tipos y una de símbolos (propiamente dicha).

La tabla de símbolos tiene una estructura de clases formada por dos tablas de acceso hash en una de las cuales se guardan los símbolos del bloque actual y en la otra se guardan los tipos (ver Figura 4-43). De una forma esquemática se puede entender la TS como en la siguiente ilustración:

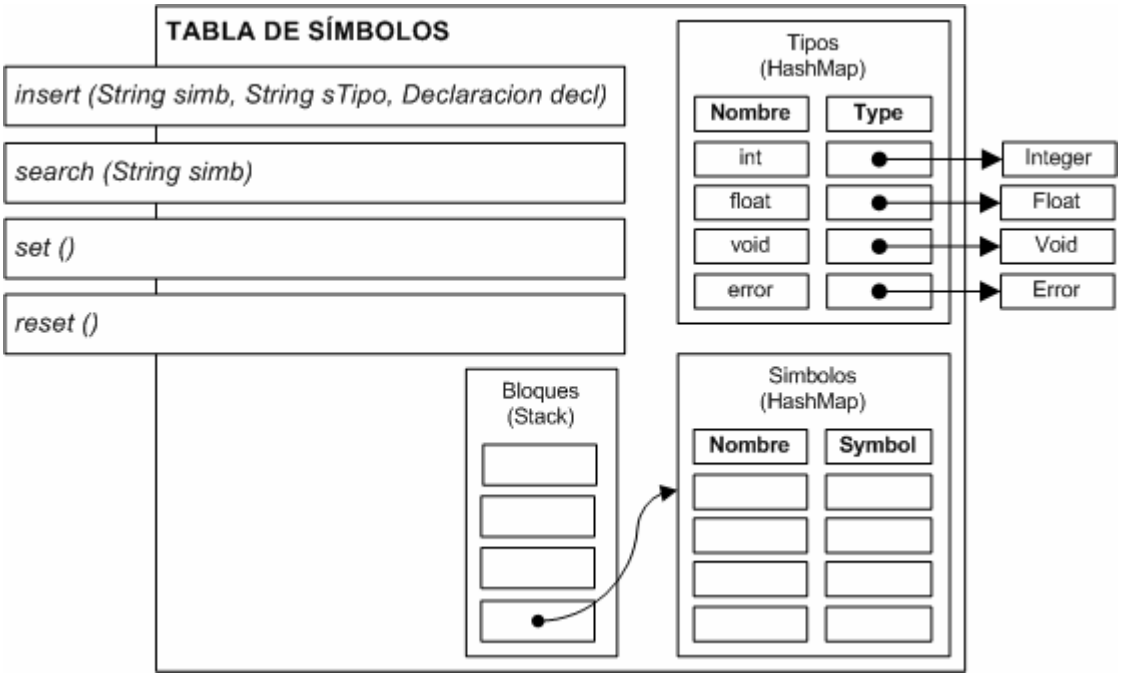


Figura 4-44. Aspecto de bloques de la TS

Como se puede ver por la ilustración se han introducido los tipos básicos en la tabla de tipos. También existe la posibilidad de tratar los tipos básicos como elementos aparte y usarlos siguiendo un patrón de diseño "*Singleton*".

En el diseño realizado la tabla de bloques crea una subtabla de símbolos de acceso hash cada vez que se entra en un bloque (ejecución de `set()`). Al ejecutarse el `reset()` se elimina esa subtabla.

Cuando se inserta un elemento (al visitar el nodo `Declaracion` del AST), se ejecuta la operación `insert()` de la TS.

Al insertar este nodo se hacen varias operaciones:

1. Se Crea un nuevo objeto `Symbol` para guardar en él la información correspondiente al símbolo que se trata de insertar. En la mayor parte de los casos será su `offset`. Cabe decir en este momento que este objeto puede ser eliminado totalmente si esta información se guarda en el propio nodo `Declaracion` del AST.
2. Desde este nodo `Symbol` se referencia al nodo `Declaracion` del AST y al tipo correspondiente en la tabla de tipos (previamente se hará la búsqueda del tipo de su nombre).

3. Ahora se debe modificar el nodo `Declaracion` para añadirle una referencia al nodo `Symbol` recién creado.
4. Por último se inserta el objeto `Symbol` en la `HashMap` actual de símbolos.

Un vez terminada la inserción del símbolo, el AST quedaría como se puede ver en la Figura 4-45. La doble referencia `Declaracion<->Symbol` para permitir la navegación en ambos sentidos (según el momento) hace pensar en una simplificación si se introduce toda la información de `Symbol` en el propio nodo `Declaracion`.

La modificación del nodo `Declaracion` es posible porque este nodo deberá tener sus atributos `public` para poder ser visitado, esto permitirá añadirlo información. Debe recordarse (del Semántico) que los nodos del árbol AST son objetos de datos más que objetos con un comportamiento complejo.

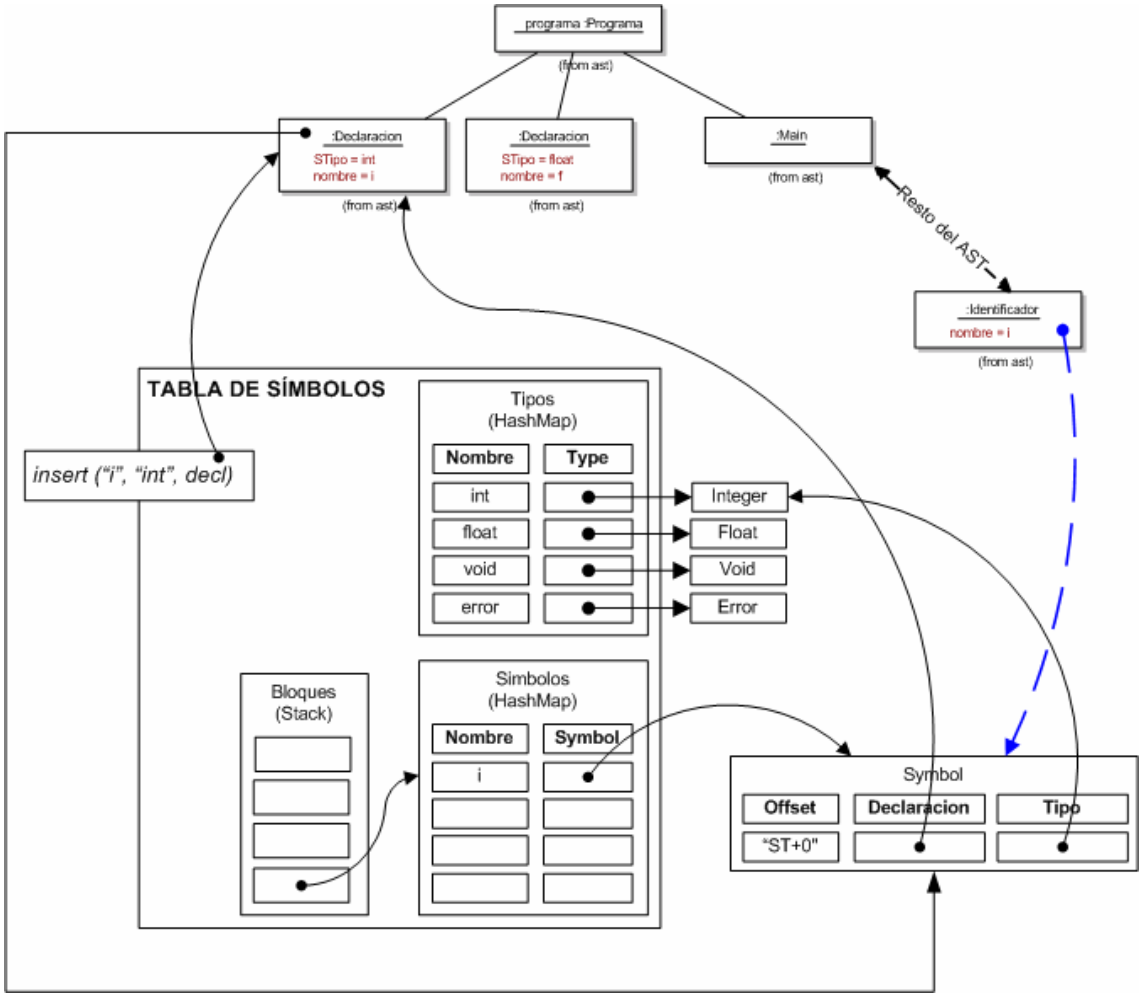


Figura 4-45. Insertar una declaración y referencias un identificador

El `visitor` Identificador seguirá visitando el árbol y al llegar al nodo `Identificador` (donde se usa esta variable "`i`") precisará hacer las siguientes operaciones:

1. Se busca el identificador "`i`" en la TS.
2. Se encuentra el nodo `Symbol` correspondiente y se retorna esta referencia.
3. El nodo `Identificador` añade esta referencia (línea a trazos) para posteriores visitas.

En este momento se debe comentar que la lógica de inserción y de búsqueda ha quedado oculta en los dos escenarios contemplados anteriormente, esto es, no se ha considera ni la doble declaración de un identificador en el mismo ámbito ni la utilización de un identificador sin su previa declaración.

La lógica en este casos se puede entender la habitual, esto es, si al insertar y símbolo ya existe otro con el mismo nombre en el mismo ámbito, se debe producir un error.

Del mismo modo, si al buscar un símbolo este no aparece en el ámbito actual, se buscaría en todos los ámbitos anteriores contenidos en la pila de bloques. Si al llegar al final de la pila todas las búsquedas resultan infructuosas se debe concluir que el símbolo no ha sido adecuadamente declarado y se debe producir el correspondiente error.

Al terminar de visitar todo el árbol, en este caso, la TS no es necesaria para posteriores visitas, por lo que la información queda referenciada a través de los objetos actualmente creados.

Esta estructura resultante (simplificada para el caso del identificador "`i`" se puede ver en la Figura 4-46. Ahora, para posteriores visitas, desde la `Declaración` es posible navegar hasta el `Symbol` y desde éste hasta el `Tipo`.

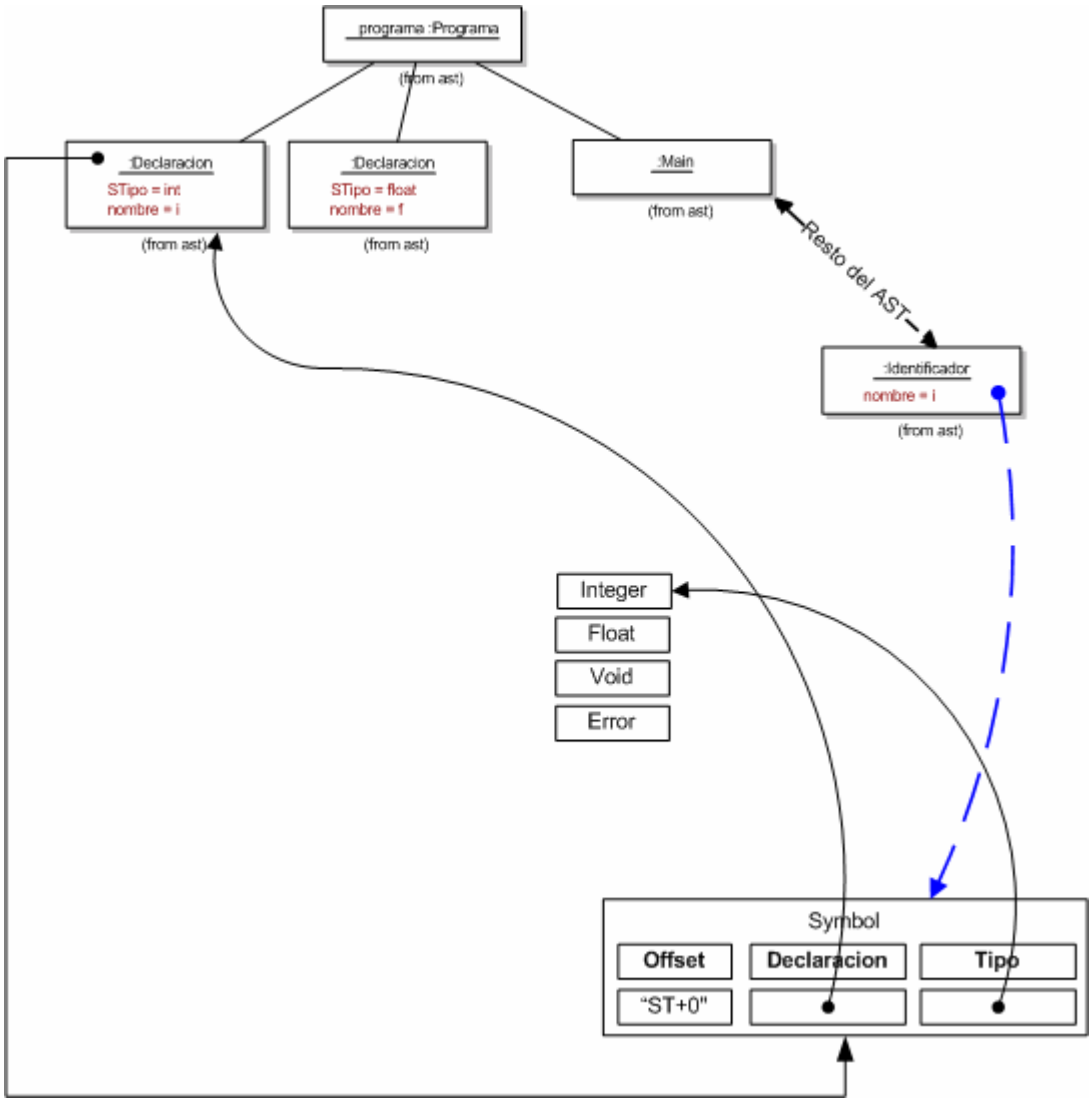


Figura 4-46. Estructura AST-TS resultante una vez terminada la visita de identificación

Del mismo modo, al llegar al `Identificador` es posible navegar al `Symbol` y desde éste al `Tipo` o al nodo `Declaracion` del AST.

Como se ha comentado en varios sitios en este apartado es posible una simplificación si se elimina el nodo `Symbol` y esta información se introduce en el propio nodo `Declaracion` del AST.

Cuando se inserta la declaración en la TS no se creará ningún nodo `Symbol` y se introducirá en el `HashMap` de símbolos el propio nodo `Declaracion` con la información correspondiente.

Una vez terminada la visita, la referencia bidireccional `Declaracion<->Symbol` no existe y el AST decorado queda bastante más simplificado con respecto al caso anterior, como se puede ver en la Figura 4-47.

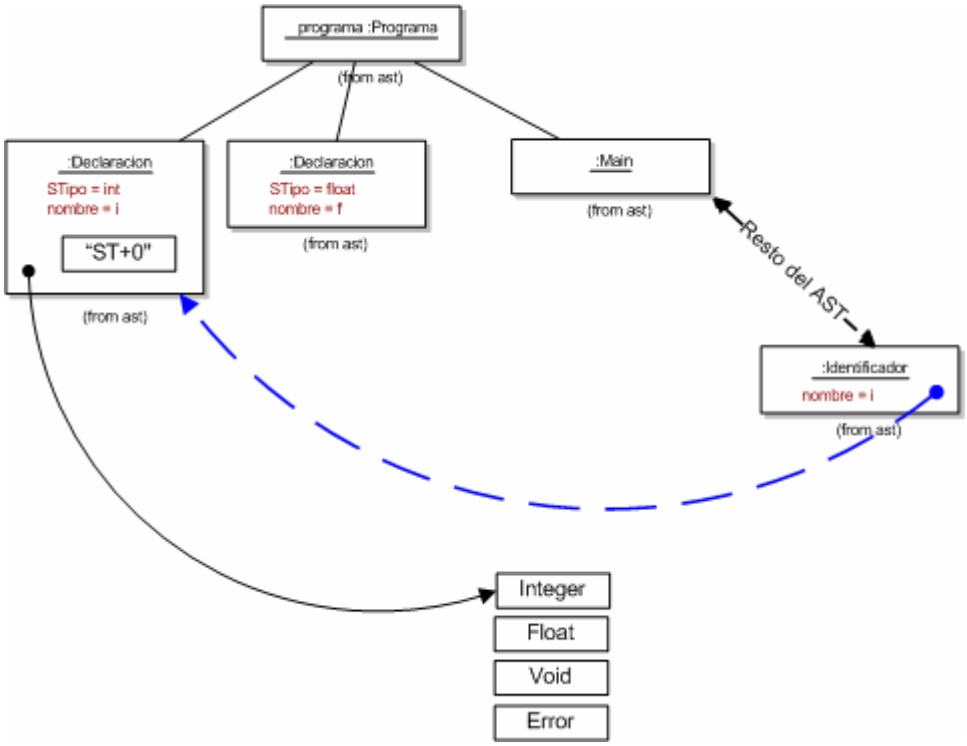


Figura 4-47. Estructura simplificada AST-TS si se elimina el objeto Symbol y se introduce toda la información en el nodo Declaracion del árbol AST

Una vez visitado el árbol completo, se puede ver en la Figura 4-48 el aspecto que tendría dicho árbol final (En este caso la clase `Integer` se ha nombrado como `IntType`).

Las referencias quedan como se representan en dicha figura. Las líneas que unen los nodos `Identificacion` con los nodos `Declaracion` no pasan por los objetos `Symbol`, en este caso se ha asumido que toda la información del objeto `Symbol` se guarda en el propio nodo `Declaracion`.

Los objetos `SymbolTable` y `TypeTable` están en este esquema sólo a modo informativo, aunque en realidad, al terminar la visita, no se tendría su referencia.

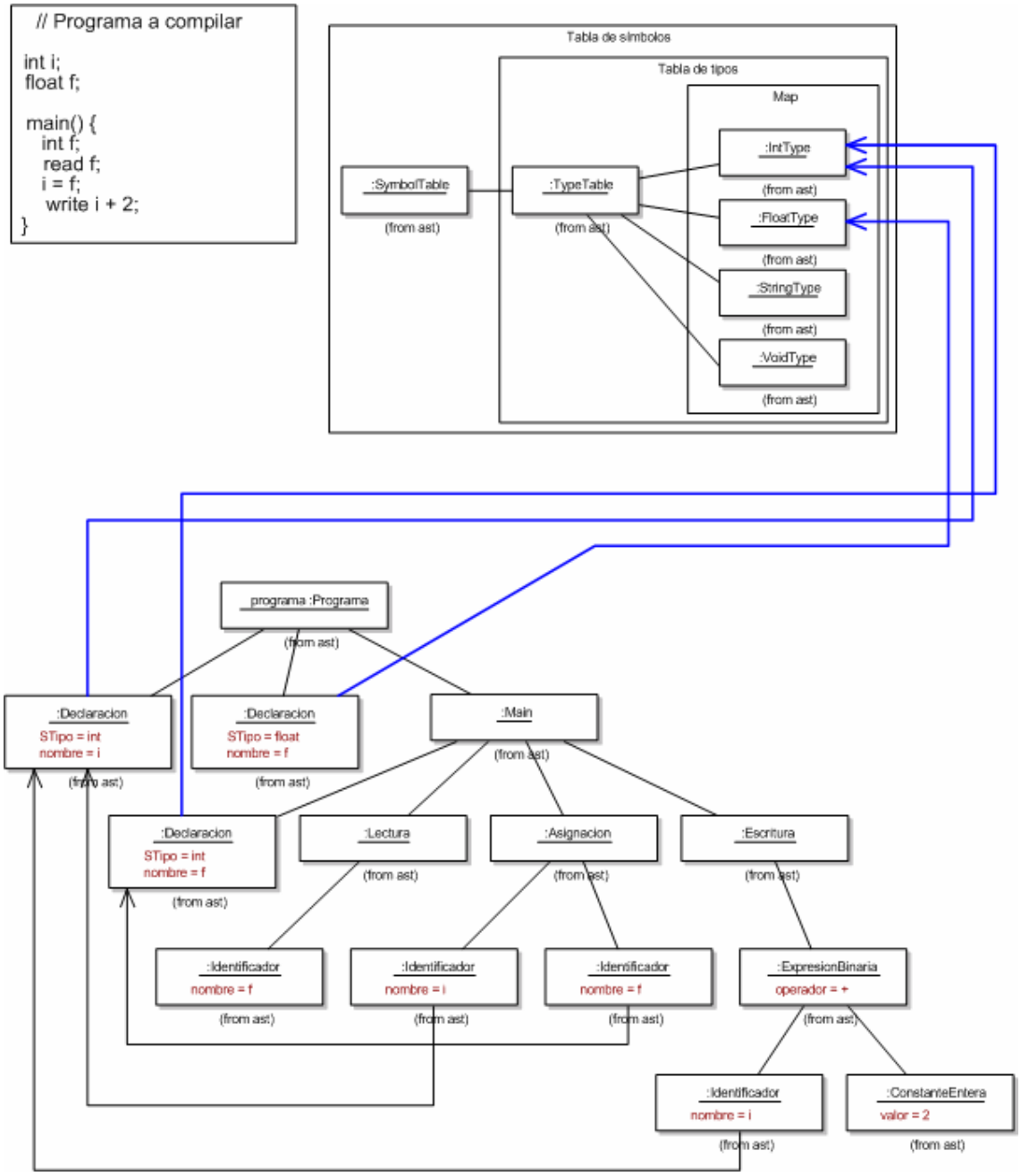


Figura 4-48. Árbol AST decorado resultante de la primera pasada del visitor

4.4.2 Tipos de usuario

Se puede explorar el tipo de usuario `struct` como ejemplo del modo en que se tratarían estos tipos.

El primer paso a considerar es que el nodo del AST donde se declare la `struct` definirá al mismo tiempo la estructura del tipo, por lo que se partirá del hecho de que este nodo debe implementar ambas interfaces (o jerarquías de herencia): `Type` y `NodoAST`.

Ahora hay tres niveles de inserción:

1. Insertar un tipo de usuario en la TS y que referencie al Nodo AST correspondiente.
2. Declarar un símbolo con el tipo de usuario definido.

3. Usar dicho tipo en algún lugar y referenciar su nodo `Identificador` adecuadamente.

Para los casos segundo y tercero el procedimiento es similar al visto en los casos anteriores, sólo que en este caso el objeto `Type` correspondiente no será un tipo básico sino uno que se ha creado previamente.

Para insertar el tipo se ha previsto que el propio nodo `Struct` implemente la interfaz de tipo, como puede verse en la Figura 4-49. `Struct`, además de ser un `NodoAST` es también un tipo (`Struct`) y eso le va a permitir formar parte del árbol y del `HashMap` de tipos.

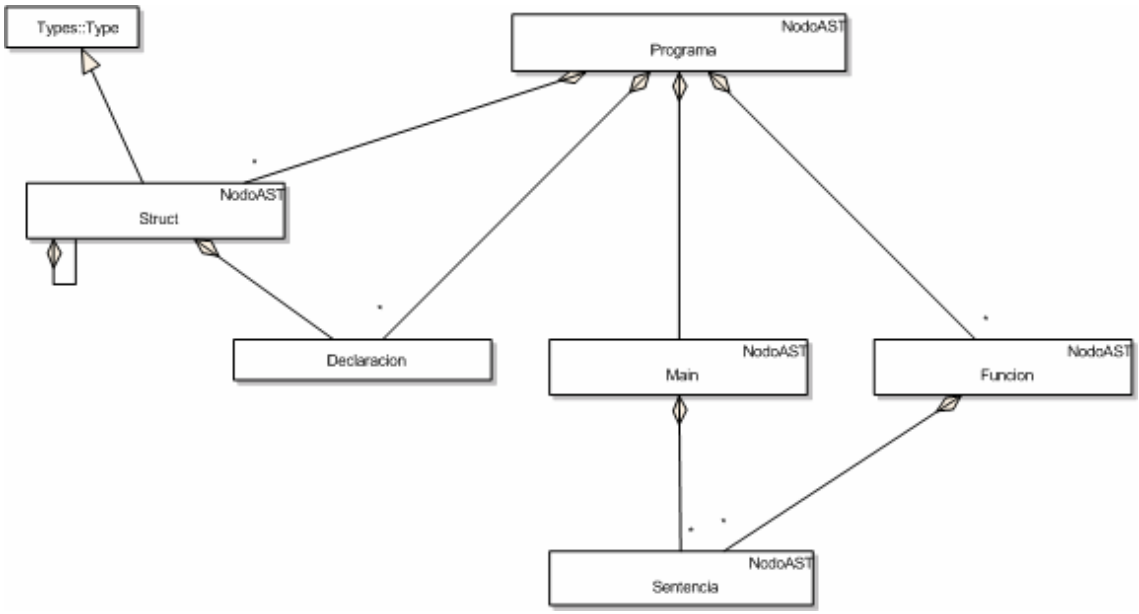


Figura 4-49. Agregación prevista entre los nodos del Árbol AST. Véase la derivación que Struct hace de Type.

Sea el siguiente código:

```
1:    struct Concepto {
2:       int cantidad;
3:       String descrip;
4:       float precio;
5:    };
6:
7:    main() {
8:       int i;
9:       i = 2;
10:      Concepto con;
11:      con.cantidad = 1;
12:      con.descrip = "teléfono";
13:      con.precio =240.00;
14:      int a [5];
15:      a[i]=7;
16:   }
```

Al pasar el analizador sintáctico generará el árbol AST del apartado 8: Anexo 4. Árbol AST del ejemplo 2 de varias pasadas.

Observando este árbol se puede ver que toda la estructura del nuevo tipo (`Struct`) está representada en el propio árbol como nodos AST.

Al visitar el visitor `Identificador` el nodo `Struct` se genera una llamada a la TS para insertar éste en la tabla de tipos.

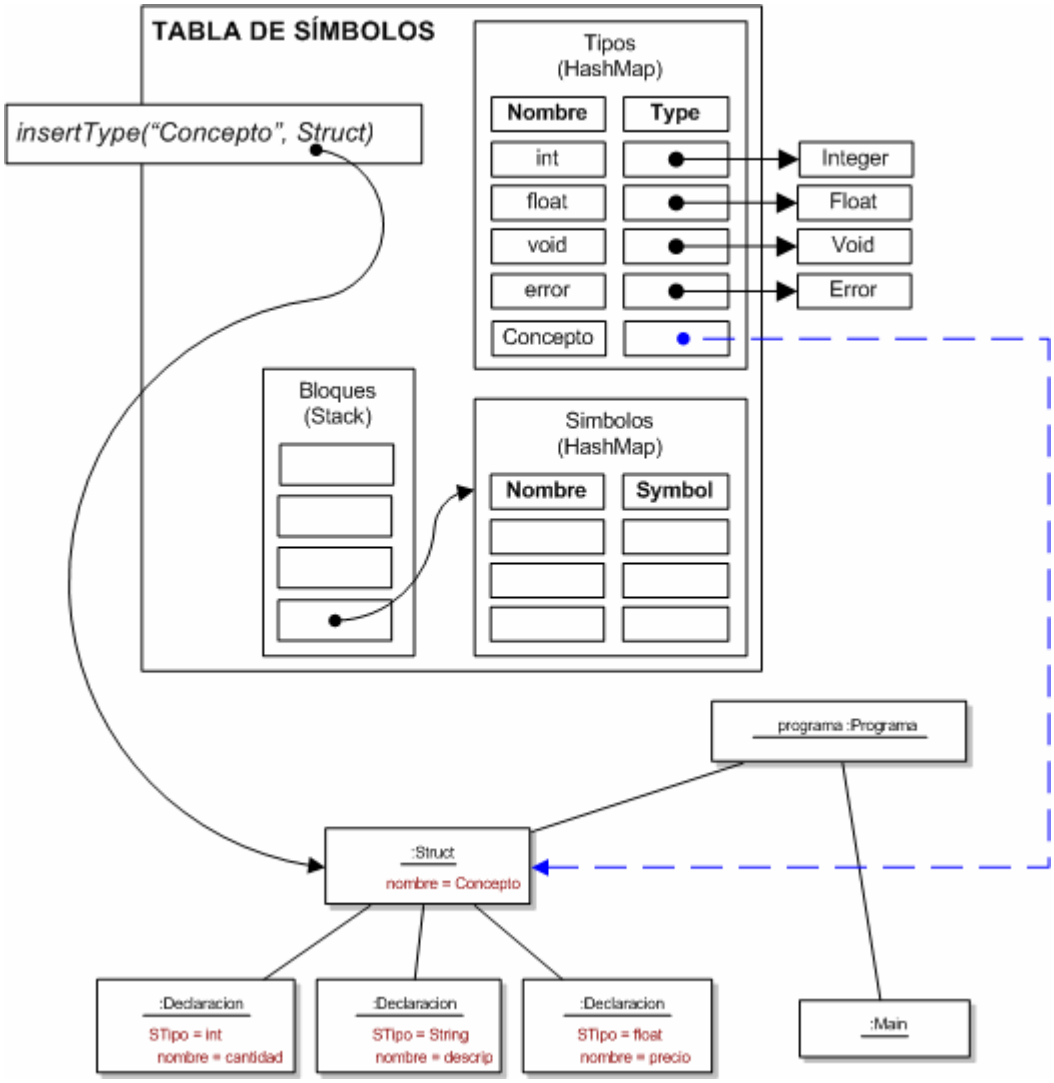


Figura 4-50. Insertar el tipo Struct

Una vez insertado, al navegar por el nodo `Declaracion` de la `Struct con`, se produce el mismo código de pasos anteriores, sólo que ahora el tipo estará en el propio árbol y por tanto la referencia al tipo será un nueva referencia a otro nodo del AST (En los casos anteriores los tipos eran los básicos del lenguaje y se creaban al crear la TS).

Por último en el lugar en que se utiliza el `Identificador con`, se hará una referencia al nodo `Declaración` correspondiente. El árbol resultante puede verse en la Figura 4-51.

Compárese esta figura con la Figura 4-47 y se podrá ver la similitud de estas declaraciones y estos identificadores con los representados en esta figura con los representados en la otra.

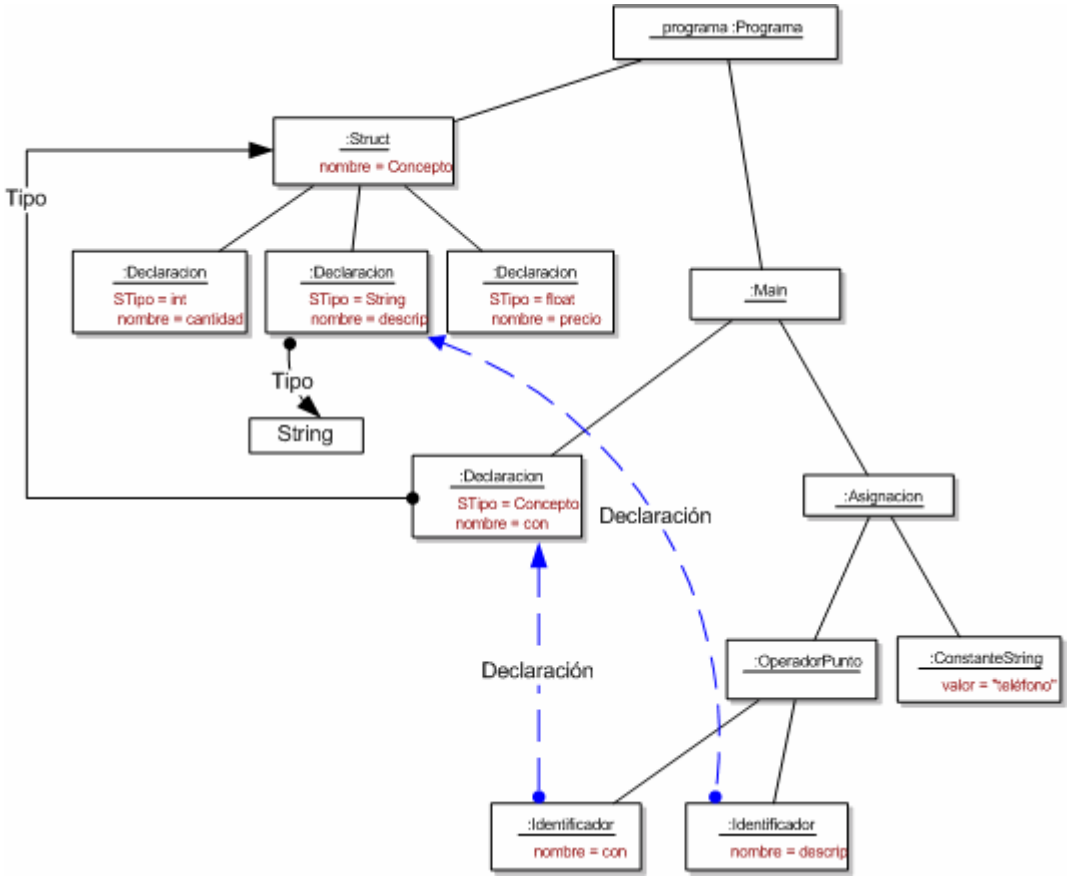


Figura 4-51. Referencias de declaración y tipo en el árbol AST para el tipo Struct

4.4.2.1 Declaración de arrays

Los `arrays` no son nuevos tipos sino que se pueden considerar variantes de los tipos que ya existen, por tanto no definirán nuevos nodos en el árbol que implementen la interfaz `Type`.

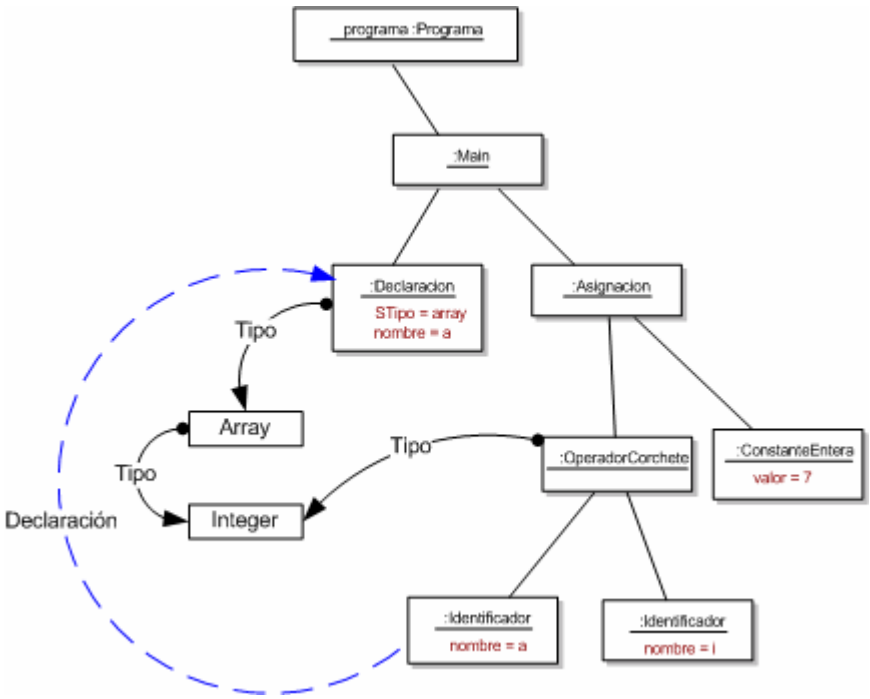


Figura 4-52. Declaración de Arrays

Nótese que el tipo del `Identificador` es `Array` en la tabla de tipos y el subtipo de este tipo es `Integer`.

Por otro lado el tipo resultante de aplicar el operador corchete es `Integer`.

4.4.3 Declaraciones forward

Hay dos maneras de hacer una declaración `forward` en un lenguaje a compilar en varias pasadas:

1. Declaración de prototipos
2. Pasada previa para reconocer prototipos

4.4.3.1 Declaración de prototipos

El primer caso no tiene ninguna diferencia con la solución aportada para el caso de una sola pasada. El prototipo debe estar antes de la utilización del identificador de manera que se pueda insertar en la TS y cuando se utilice se le hará referencia a lo definido en el prototipo.

Cuando se defina el tipo realmente, se comprobará que ha sido definido previamente como un prototipo y que la definición de tipo definitiva no contradice lo que estuviera ya definido en el prototipo y contenido en la TS.

4.4.3.2 Pasada previa para reconocer prototipos

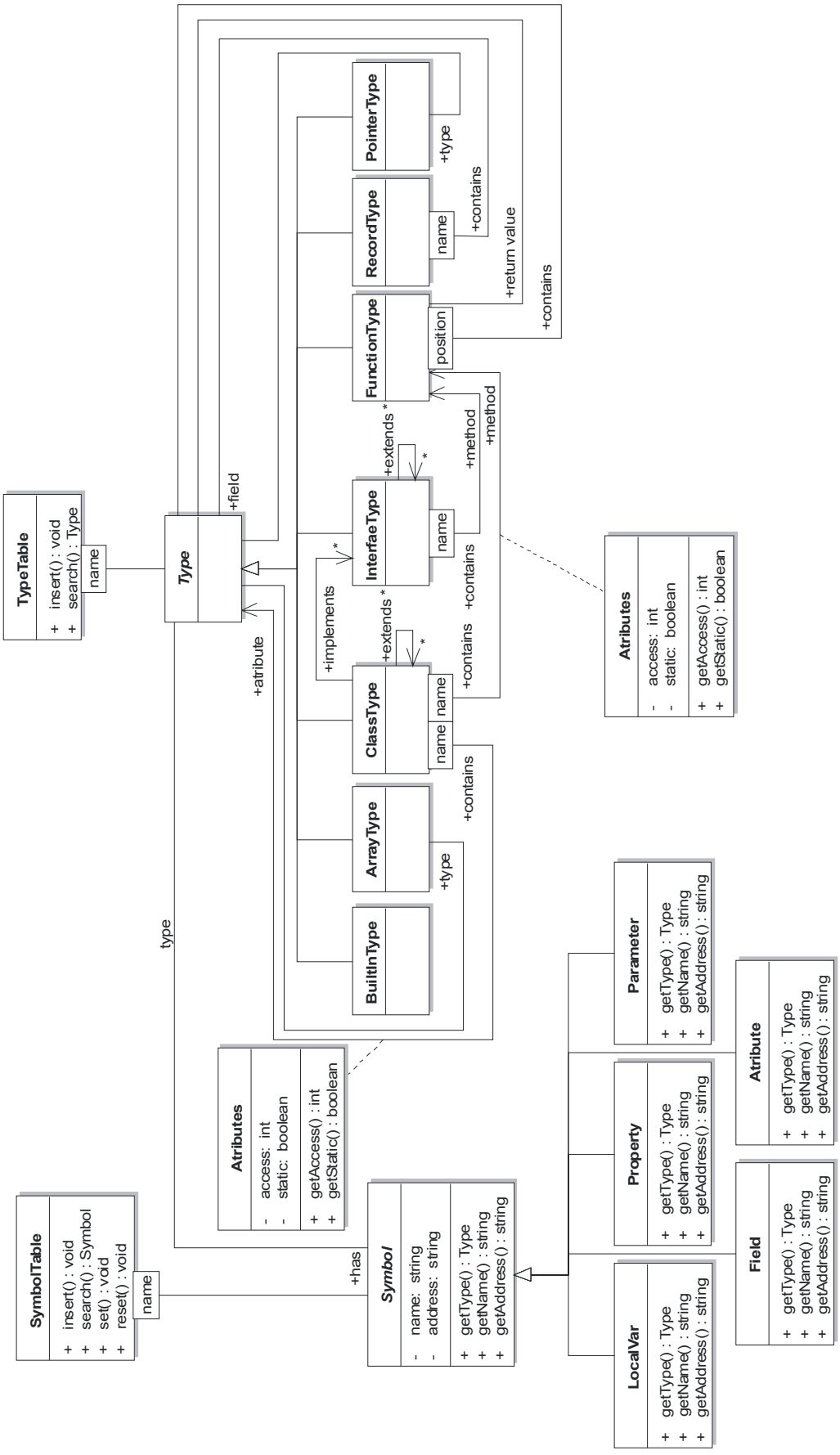
El segundo caso provoca que la visita para reconocer los identificadores se parta en dos:

1. El visitor primero recorre el árbol buscando elementos que puedan definirse como prototipos, por ejemplo estructuras, clases, funciones, métodos, etc. y los inserta en la TS.
2. El segundo Vuelve a recorrer el árbol introduciendo el resto de los elementos en la TS. Ahora ya no hay problemas puesto que todos los tipos de usuraos y las funciones/métodos existirán en la TS.

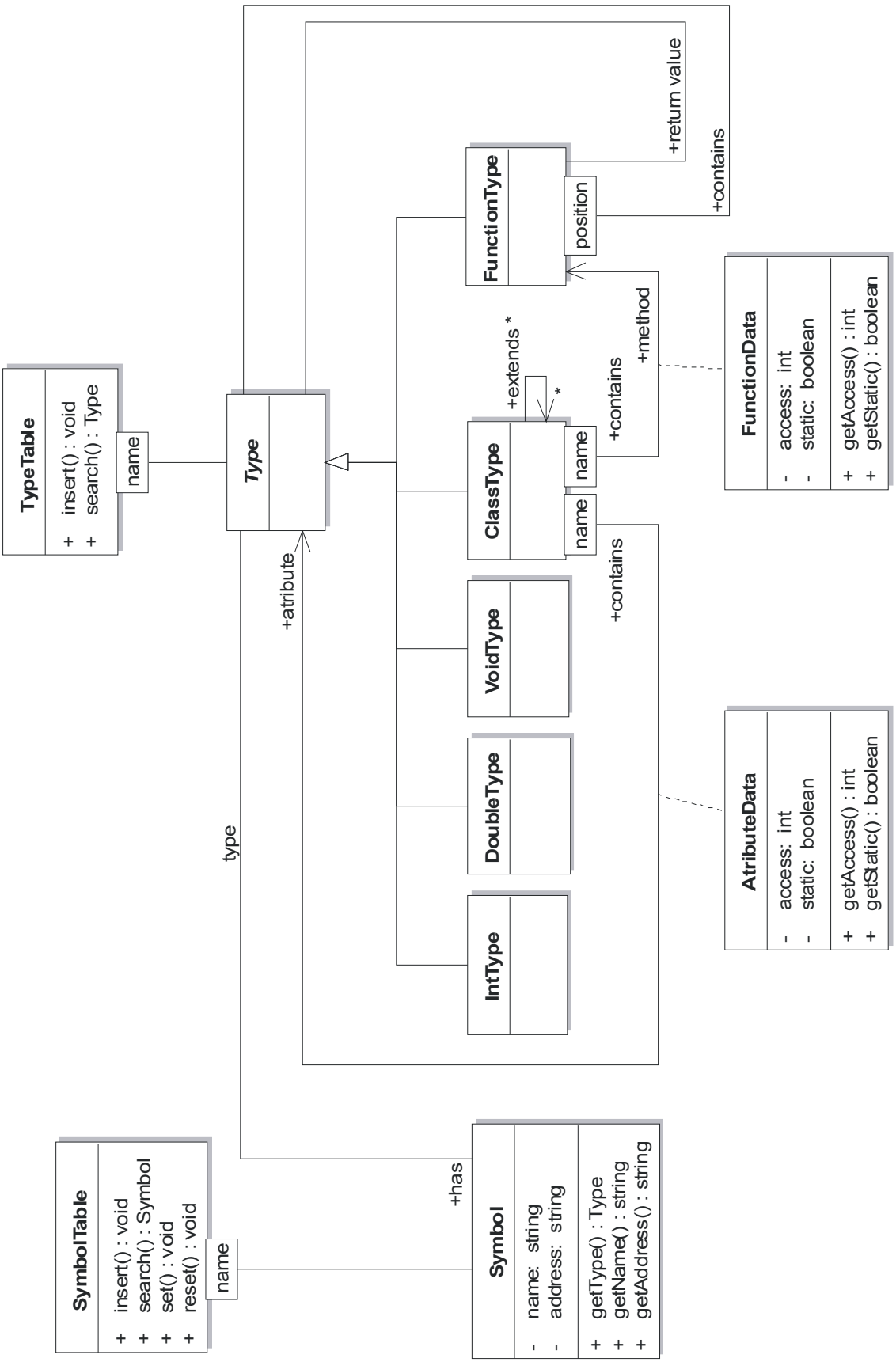
En este segundo caso se deben tener en cuenta una serie de restricciones sobre la TS:

1. La TS debe ser compartida por dos visitor diferentes, por tanto debe ser declarada en un ámbito global a ambos (También podría ser un *singleton*).
2. No se pueden eliminar los ámbitos en la primera pasada, puesto que pueden ser necesarios en la segunda y por tanto hay que reestructurar el comportamiento de la TS para que el `reset()` no elimine información.

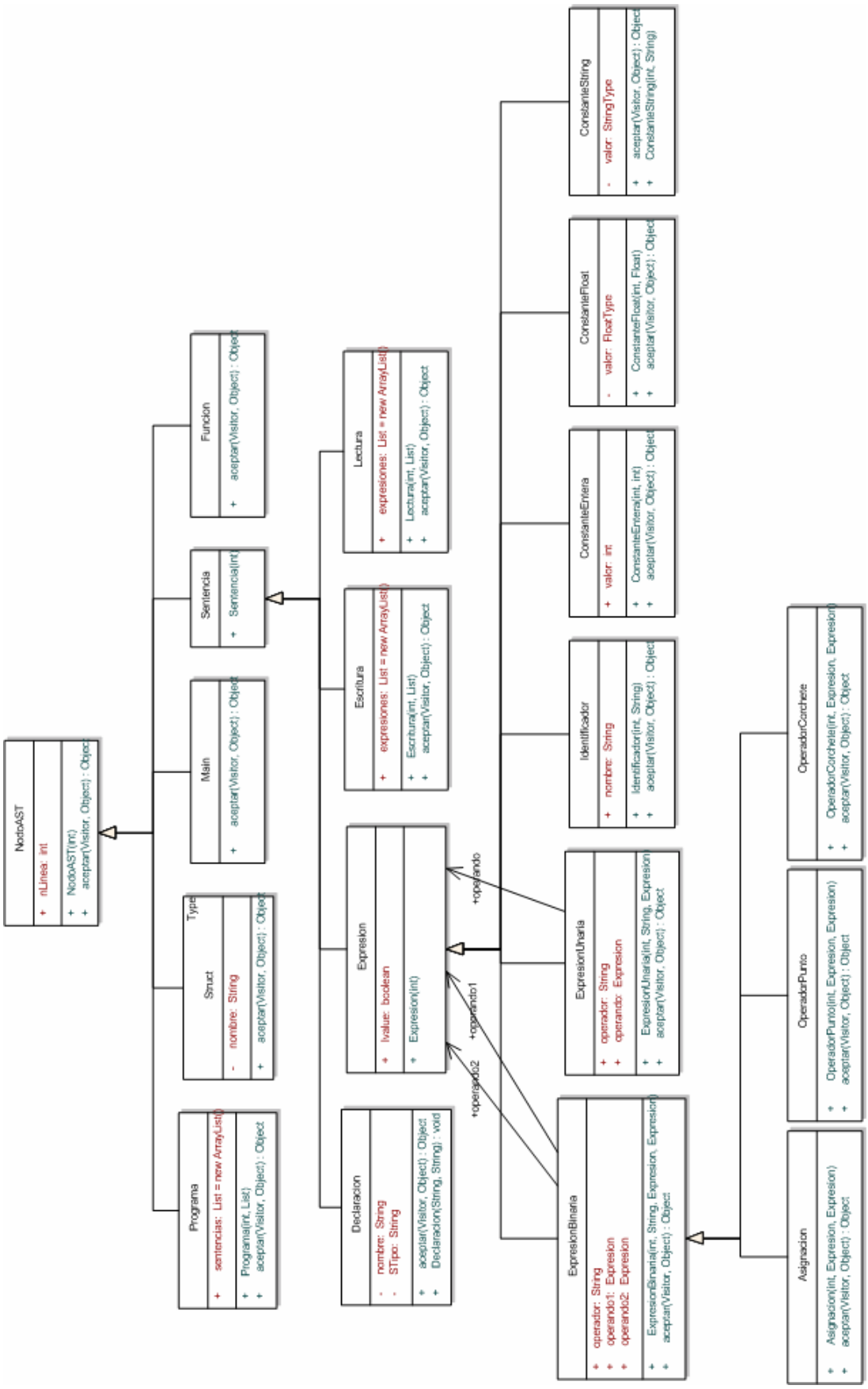
5 Anexo 1. Diagrama General de Clases



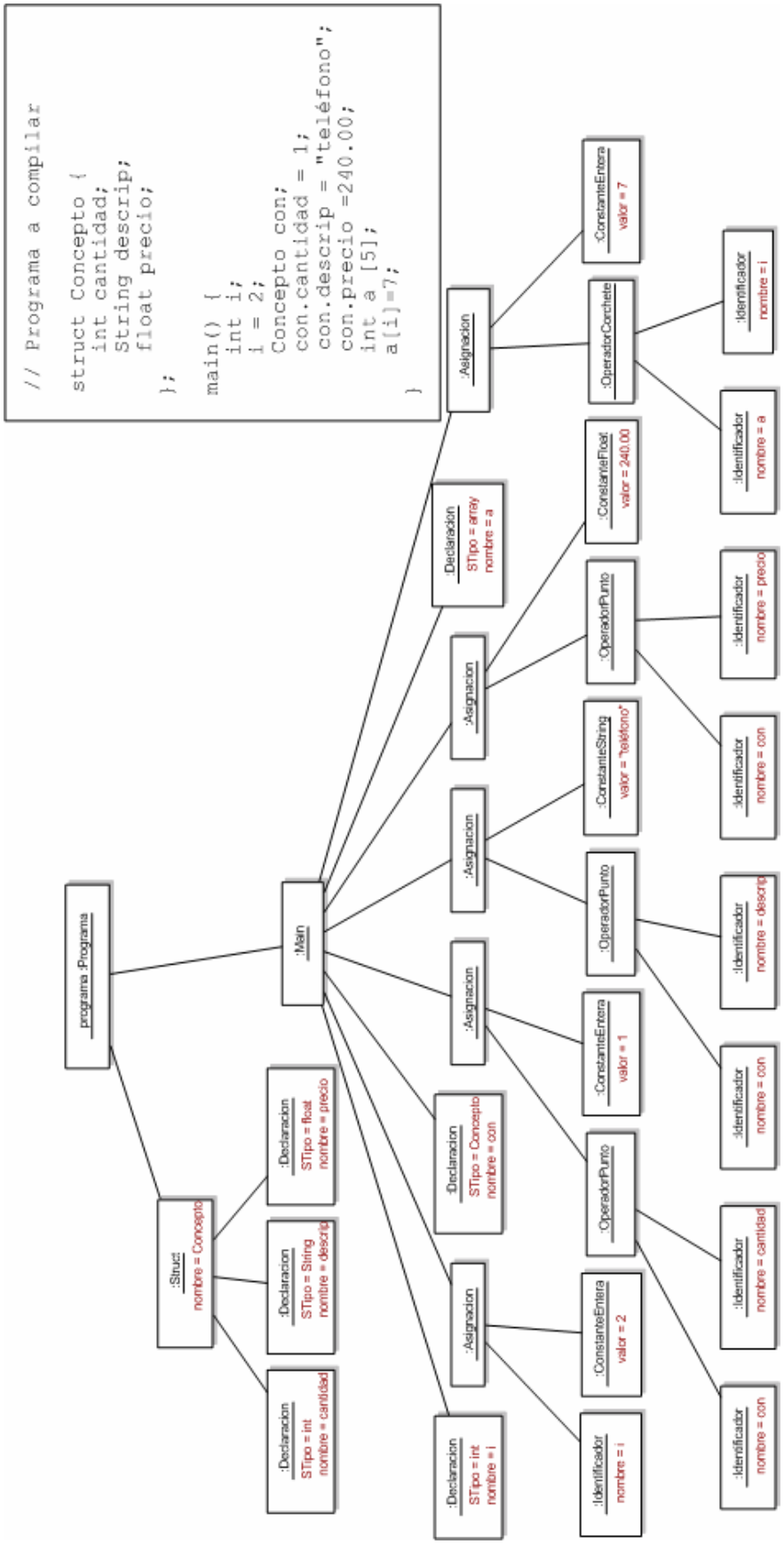
6 Anexo 2. Tabla de símbolos simplificada para el ejemplo



7 Anexo 3. Descripción de los nodos del AST (Ejemplo primero de varias pasadas)



8 Anexo 4. Árbol AST del ejemplo 2 de varias pasadas



9 Bibliografía y Referencias

AHO86 A. Aho, R. SEIT y J. D. Ullman. *Compiladores: principios, técnicas y herramientas*. Addison-Wesley Interamericana (1990).

AVL62 G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. Soviet Math Doklady, 3:1259--1263, 1962

BUCH63 Werner Buchholz: *File Organization and Addressing*. IBM Systems Journal 2(2): 86-111 (1963)

CUEV91 Cueva Lovelle, J. M. *Lenguajes, Gramáticas y autómatas*. Cuaderno didáctico nº 36, Depto. De Matemáticas, Universidad de Oviedo, 1991.

CUEV95c Cueva Lovelle, J. M. *Organización de la memoria en tiempo de ejecución en Procesadores de Lenguaje*. Cuaderno didáctico nº 55, Depto. De Matemáticas, Universidad de Oviedo, 1995.

CUEV95d Cueva Lovelle, J. M. *Tablas de Símbolos en Procesadores de Lenguaje*. Cuaderno didáctico nº 54, Depto. De Matemáticas, Universidad de Oviedo, 1995.

CUEV03 J.M. Cueva, R. Izquierdo, A.A. Juan, M.C. Luengo, F. Ortín, J.E. Labra. Lenguajes, Gramáticas y Autómatas en Procesadores de Lenguaje. Servitec. 2003.

HIBB62 Thomas N. Hibbard, *Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting*, Journal of the ACM (JACM), v.9 n.1, p.13-28, Jan. 1962

HOLUB90 Allen I. Holub. *Compiler design in C*. Prentice-Hall (1990). (http://www.holub.com)

HOPCR02 J.E. Hopcroft, R. Motwani, J.D. Ulman. Introducción a la Teoría de autómatas, lenguajes y computación. Pearson Educación. 2002.

LOUDE97 Kenneth C. Louden. Compiler Construction: Principles and Practice. Brooks Cole. 1997.

LUM71 Vincent Y. Lum, P. S. T. Yuen, M. Dodd: *Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files*. CACM 14(4): 228-239 (1971)

C. P. Wang, Vincent Y. Lum: *Quantitative Evaluation of Design Tradeoffs in File Systems*. SIGIR 1971: 155-162

KNUTH73 Knuth, D.: *The Art of Computer Programming, Vol. 3, Sorting and Searching,* Addison Wesley, Reading, Mass. (1973).

Knuth D. *El arte de programar ordenadores, Volumen III: Clasificación y búsqueda*. Ed. Reverté, 1987.

ORTIN04 F. Ortín, J.M. Cueva, J.E. Labra, A.A. Juan, M.C. Luengo, R. Izquierdo. Análisis Semántico en Procesadores de Lenguajes. Servitec. 2004. ISBN: 84-688-6208-8

SEVE74 Dennis G. Severance: *Identifier Search Mechanisms: A Survey and Generalized Model.* ACM Computing Surveys 6(3): 175-194(1974)