ANALISIS SEMANTICO

Los programas que se compilan no son solamente cadenas de símbolos sin ningún significado que pueden ser aceptadas como correctas o no por una maquina abstracta. El lenguaje no es más que el vehiculo por el cual se intenta transmitir una serie de instrucciones a un procesador para que éste las ejecute produciendo unos resultados. Por ello, la tarea del compilador requiere la extracción del contenido semántico incluido en las distintas sentencias del programa.

Simultaneamente existen ciertos aspectos relativos a la corrección de un programa que no se pueden expresar claramente mediante el lenguaje de programación, y otros para los que resultaria muy engorroso hacerlo.

Por todo ello, se hace necesario dotar al compilador de una serie de rutinas auxiliares que permitan captar todo aquello que no se ha expresado mediante la sintaxis del lenguaje y todo aquello que hace descender a nuestro lenguaje de programación de las alturas de una maquina abstracta hasta el nivel de un computador real. A todas estas rutinas auxiliares se las denomina genéricamente análisis semántico.

El análisis semántico, a diferencia de otras fases, no se realiza claramente diferenciado del resto de las tareas que lleva a cabo el compilador, mas bien podria decirse que el análisis semántico completa las dos fases anteriores de análisis lexicografico y sintáctico incorporando ciertas comprobaciones que no pueden asimilarse al mero reconocimiento de una cadena dentro de un lenguaje.

Desde el punto de vista del compilador, las rutinas de análisis semántico pueden ser de dos tipos:

(1) Estáticas. Son aquellas que se realizan durante la compilación del programa

(2) Dinámicas. Son aquellas que el compilador incorpora al programa traducido. Para ello, el compilador debe añadir en el código objeto ciertas instrucciones que se ejecutarán simultáneamente a la ejecución del programa.

Entre las comprobaciones de tipo estático podemos destacar las comprobaciones de tipos, que se estudian a continuación con mayor detenimiento. Con ellas se pretende garantizar que los dominios de todas las variables que se emplean en un programa son los adecuados en cada caso o de no serlo, realizar las oportunas modificaciones para garantizar la compatibilidad.

Otras comprobaciones estáticas son las de unicidad. Muchos lenguajes de programación exigen que los identificadores que utilizan sean declarados previamente de forma univoca. Sin embargo, aunque parece trivial, no está incluido en la sintaxis de contexto libre de un lenguaje el que una variable determinada se defina solamente una vez. Por tanto se debe incluir esta comprobación entre las rutinas semanticas. Por lo general esta tarea se realiza conjuntamente a la gestión de la tabla de simbolos que se estudia en el siguiente capítulo.

Otros tipos de comprobaciones llevadas a cabo durante la fase de compilación serían las referentes al control de flujo en determinados lenguajes, asi por ejemplo en BASIC debemos asegurarnos de que las sentencias FOR y NEXT están debidamente anidadas. Téngase en cuenta que este lenguaje no tiene estructura de bloques y que sintácticamente podrian quedar sentencias FOR sin su correspondiente sentencia NEXT o viceversa.

Entre las comprobaciones dinámicas, las más evidentes son aquellas que dependen del estado de la máquina en el momento de la ejecución del programa, como por ejemplo, los fallos en la apertura de ficheros, errores de dispositivos, etc...; pero también hay otras que dependen del propio programa como son los errores numéricos, divisiones por cero, desbordamiento númerico (overflow),etc., o los errores de direccionamiento de memoria como puedieran ser el desbordamiento de la pila (stack) o del montón (heap) y los errores por exceso o defecto en el acceso a una variable indexada. En algunos casos podria pensarse que la comprobación puede realizarse en tiempo de compilación y de hecho, en algunos casos así es. Sin embargo en el caso general hay ciertas comprobaciones deben hacerse necesariamente durante la ejecución del programa. Por ejemplo, si el indice de una matriz no se corresponde con una constante, sino con una variable o una expresión, será necesario evaluar esta expresión para saber si ha habido desbordamiento. Esta evaluación sólo se efectúa en tiempo de ejecución ya que el valor que tomarán las variables es desconocido durante la compilación.

Sistemas de tipos

1. Expresiones de tipos.

Se entiende por *sistema de tipos* de un lenguaje de programación todas aquellas reglas que permiten asignar tipos a las distintas partes de un programa y verificar su corrección. En concreto formarán el *sitema de tipos* las definiciones y reglas que permiten comprobar cual es el dominio asignado a una variable, y en que contextos puede ser usada.

Cada lenguaje tiene un *sistema de tipos* propio, aunque a veces puede variar de una a otra implementación. Un sistema de tipos incluye tanto comprobaciones estáticas como dinámicas, según que se realicen en tiempo de compilación o en tiempo de ejecición. Como es lógico, mientras más comprobaciones puedan realizarse en la fase de compilación, menos tendran que realizarse durante la ejecución, con la consiguiente mejora en la eficiencia del programa objeto. En la práctica ningún lenguaje es tan fuertemente estructurado que permita una completa comprobación estática de tipos.

Para abordar el sistema de tipos de un lenguaje de programación imperativo del estilo de C, PASCAL, MODULA, ADA etc... se define una estructura asociada a cada segmento de código de un lenguaje a la que se denomina "*expresión de tipo*".

Cada lenguaje de programación requerirá unas expresiones de tipos adecuadas a sus características.A continuación, a modo de ejemplo, se definen las expresiones de tipos más comunes usadas en un lenguaje imperativo similar a C o PASCAL:

(1) Tipos simples: Son *expresiones de tipos* los tipos simples del lenguaje, y algunos tipos especiales:

integer
real
char
boolean
void
error

Los cuatro primeros son los tipos de datos simples mas comunes en los lenguajes de programación, los dos últimos son tipos simples especiales que usaremos para su atribución a diversas partes de un programa a fin de homogeneizar el tratamiento de todo el conjunto mediante el metodo de las expresiones de tipos.

Tomando el lenguaje C como ejemplo, el segmento de código al que estan asociada la expresion de tipos integer es aquella en que aparece la palabra reservada "`int`", etc..

(2) Identificadores: Son expresiones de tipos los identificadores de tipos definidos en el propio programa, es decir los identificadores correspondientes a los tipos definidos por el usuario.

Por ejemplo supongamos que en PASCAL definimos el tipo :

```
type caracter = char;
```

entonces, se puede considerar que el identificador *`caracter`* es una expresion de tipos, al igual que lo son los tipos simples definidos en el punto anterior. En principio, no todos los sistemeas de tipos incluyen identificadores de tipos, como se verá esto depende de si existen o no tipos definidos por el usuario y del criterio de equivalencia de tipos que se fije en el lenguaje.

(3) Constructores de tipos: Permiten formar tipos complejos a partir de otros más simples. La semántica de cada lenguaje tiene asociada unos constructores de tipos propios. En general, en los lenguajes de programación se definen los siguientes constructores:

(a) Matrices: Si *T* es una expresión de tipos, entonces *array(R,T)* es también una expresión de tipos que representa a una matriz de rango *R* de elementos de tipo *T*.

Ejemplos: Sea el segmento de código C :

```
char a[10]
```

o el segmento de código PASCAL :

```
a:array[0..9] of char
```

A ambos segmentos de código debe asignarseles la expresión de tipos:

array(0-9,char)

(b) Productos: Sea T_1 y T_2 expresiones de tipos, $T_1 \times T_2$ es una expresión de tipos que representa al producto cartesiano de los tipos T_1 y T_2. A fin de simplificar consideraremos que el constructor u operador de tipos $\times$ es asociativo por la izquierda. Mas adelante se verán ejemplos de esta definición.

(c) Registros: Sea un registro formado por los campos u_1, u_2 ... u_N, siendo cada uno de ellos de los tipos T_1,T_2 ... T_N, entonces, la expresión de tipos asociada al conjunto es :

$$record\ (\ (u_1{:}T_1) \times (u_2{:}T_2) \times \ldots \times (u_N{:}T_N)\ )$$

Ejemplos: Sea el segmento de código C :

```
struct
{
      char nombre[30]; float dni;
}
```

O el segmento de código PASCAL :

```
record
      nombre:array[0.29] of char; dni:real;
end;
```

en ambos casos se asigna a estos segmentos de código la expresión de tipos:

$$record\ (\ (nombre{:}array(0\text{-}9,char)) \times (dni{:}real)\ )$$

(d) Punteros.Si T es una expresión de tipos, entonces $pointer(T)$ es una expresión de tipos que representa a un puntero a una posición de memória ocupada por un dato de tipo T.

Ejemplo: Sean los segmentos de código C y PASCAL

```
char *p;                    p:^char
```

en ambos casos se les asigna la expresión de tipos:

$$pointer(char)$$

(e) Funciones: Sean T_1,T_2 ... T_N, las expresiones de tipos asociadas a los segmentos de código correspondientes a los argumentos de una función, y sea T el tipo devuelto por la función. Entonces, la expresión de tipos asociada a la función es :

$$((T_1 \times T_2 \times \ldots \times T_N) \rightarrow T)$$

Por simplicidad, supondremos en lo sucesivo que el constructor u operador de tipos $\times$ tiene mayor prioridad que el constructor $\rightarrow$

La anterior lista de constructores de tipos no es exhaustiva, según el lenguaje se definen los constructores de tipos necesarios. Por ejemplo, pudiera ser que un lenguaje incluyera un constructor de tipo para definir pilas (de elementos de tipo *T*), o ficheros (de registros de tipo *T*), etc.

(4) Variables: En algunos casos, podriamos considerar también la existencia de variables dentro de las expresiones de tipos, es decir expresiones de tipos que funcionan como variables que toman un valor dentro del dominio de las expresiones de tipos definidas en los puntos (1)-(3). Esto es de utilidad al tratar el polimorfismo.

2. Equivalencia de tipos.

Una caracteristica importante del sistema de tipos de un lenguaje de programación es sin duda el conjunto de reglas que permiten decir que dos tipos son iguales. Es ésta una caracteristica que queda en muchos casos sin definir en las especificaciones del lenguaje, dando por ello lugar a diferentes interpretaciones por parte de los realizadores del compilador. Esto se traduce en la practica en diferencias que afectan a la compatibilidad entre diferentes dialectos de un mismo lenguaje.

Exsiten principalmente tres modalidades de equivalencia entre tipos:

(1) Equivalencia estructural: Dos tipos son estructuralmente equivalentes sí y solo sí son el mismo tipo básico, o están formados por la aplicación de un mismo constructor a tipos estructuralmente equivalentes

Ejemplo: Supongamos que en un programa PASCAL tenemos las siguientes definiciones:

```
type
     tipovector = array [0..9] of integer;
     tipomatriz = array [0..9] of array [0..9] of integer;
var
     vector   : array [0..9] of integer;
     arreglo  : array [0..9,0..9] of integer;
     matriz   : tipomatriz;
     vecvec1  : array [0..9] of tipovector;
     vecvec2  : array [0..9] of tipovector;
```

Los tipos asociados a las variables `arreglo`, `matriz`, `vecvec1` y `vecvec2` son estructuralmente equivalentes ya que su expresión de tipos es en todos los casos:

array(0-9,array(0-9,int))

Sin embargo la variable `vector`, no es estructuralmente equivalente a las anteriores porque la expresión de tipos que le corresponde es:

array(0-9,int)

(2) Equivalencia nominal. Se dice que dos tipos son nominalmente equivalentes cuando son estructuralmente equivalentes considerando como tipos básicos e indivisibles a los identificadores de tipos. Es decir, la equivalencia nominal funciona igual que la equivalencia estructural siempre y cuando no existan tipos definidos por el programador, o bien, si interpretamos que en este caso los tipos del usuario pasan a formar parte del repertorio de tipos básicos

Ejemplo : En el ejemplo anterior los tipos correspondientes a las variables `vecvec1` y `vecvec2` son nominalmente equivalentes, porque en ambos casos podemos considerar la expresión de tipos:

array(0-9,tipovector)

Sin embargo, la expresión de tipos de la variable `arreglo` es:

array(0-9,array(0-9,int))

y por tanto no puede considerarse nominalmente equivalente a las anteriores.

(3) Equivalencia funcional. Dos tipos se consideran equivalentes funcionalmente cuando pueden emplearse indistintamente en un mismo contexto.

Ejemplo: Supongamos el programa en lenguaje C :

```
int mat_chica[10], mat_grande[40];

void ordena (mat, n)
int mat[],n;
{
.....
}

ordena (mat_chica,10);
ordena (mat_grande,40);
```

En este contexto los tipos de las variables `mat_chica` y `mat_grande` son funcionalmente equivalentes. El sistema de tipos del lenguaje C no es tan estricto como el de PASCAL o MODULA ello se debe en parte a la utilización de la equivalencia funcional y a las conversiones automáticas de tipos que emplea. No debe confundirse la equivalencia funcional con la compatibilidad de tipos. Existe una diferencia sutil entre dos tipos compatibles y dos tipos funcionalmente equivalentes. En el primer caso, el compilador realiza sobre uno de ellos una transformación interna para que ambos se transformen en tipos equivalentes; en el caso de la equivalencia funcional no se realiza ninguna modificación interna, sino que se relaja el criterio de equivalencia.

3. Representación de una expresión de tipos

Si queremos analizar los diferentes tipos que intervienen dentro de un programa, será necesario que el compilador cuente con alguna estructura interna que le permita manejar comodamente las expresiones de tipos.

La representación interna que el compilador tiene de la expresión de tipos asociada a un determinado trozo de código debe ser fácilmente manipulable, ya que su creación se realizará conforme se realiza la lectura del programa fuente. Por otra parte debe permitir comparar fácilmente las expresiones asignadas a distintos trozos de código, especialmente a los identificadores de variables, esta comparación dependerá de la modalidad de equivalencia que se aplique, por lo que la representación que se elija también debe ser acorde a esta caracteristica. Las representaciones internas más habituales son las siguientes:

(1) Codificadas.Se usan cuado las expresiones de tipos son sencillas. Consisten en asignar un número natural a cada una de las expresiones de tipos que puedan construirse en el lenguaje. Esta asignación debe hacerse mediante una función inyectiva a fin de que sirva a los efectos de comprobación de equivalencia.

Por ejemplo, supongamos un lenguaje en el que se han definido los tipos simples: *char*, *integer*, y *real*, no se consideran identificadores de tipos y los únicos constructores de tipos que se utilizan son:

pointer(T)	puntero a una variable de tipo *T*.
array(T)	matriz uni o pluridimensional a elementos de tipo *T*.
function(T)	función que devuelve un valor de tipo *T*.

Se asignan los siguientes códigos:

char	1
integer	2
real	3
pointer	1
array	2
function	3

Todas las expresiones de tipos pueden codificarse mediante el siguiente algoritmo recursivo:

(a) Sí una expresión de tipos corresponde a un tipo simple se le asigna el código correspondiente el tipo simple.

(b) Sí una expresión de tipos es compuesta, estará formada por un constructor aplicado a un tipo *T*.Este tipo *T* estará representado por un número natural *N*. El número natural que se asigna a la expresión de tipos compuesta se halla mediante la fórmula:

$$4*N + \textit{número asignado al constructor de tipos}$$

Es fácil demostrar que la función así construida es inyectiva, es decir a cada expresión de tipos le asigna un único número natural, y por consiguiente para saber si dos expresiones de tipos son iguales basta comparar los códigos correspondientes.

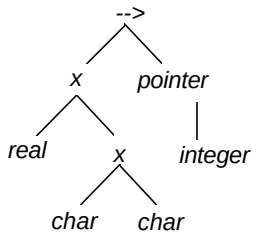
(2) Árboles. Internamente el compilador puede emplear estructuras de datos de tipo arborescente para representar las estructuras de tipos. En general, basta con emplear

árboles binarios, pero esto depende del sistema de tipos que se haya definido en el lenguaje. En las representaciones mediante árboles los tipos compuestos dan lugar a la aparición de un nodo en el que se sitúa el constructor de tipos, (es decir el operando), y una o más ramas que apuntan al árbol que contiene las estructuras de tipos de los operadores de la expresión.

Ejemplo: Para el sistema de tipos genérico definido en el primer apartado, la expresión de tipos:

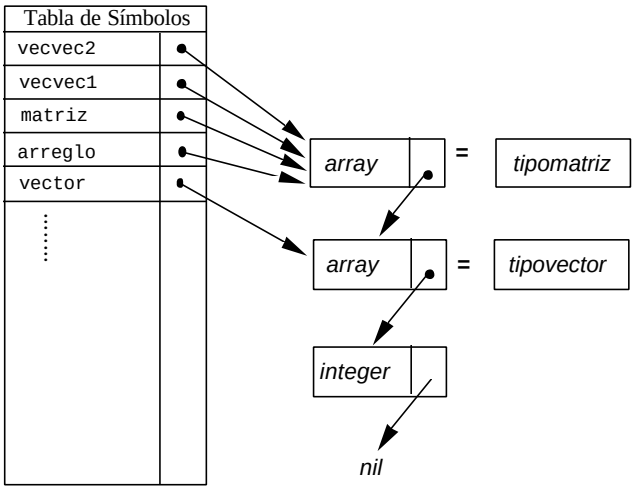
(real x (char x char) --> pointer(integer))

puede representarse mediante el árbol:

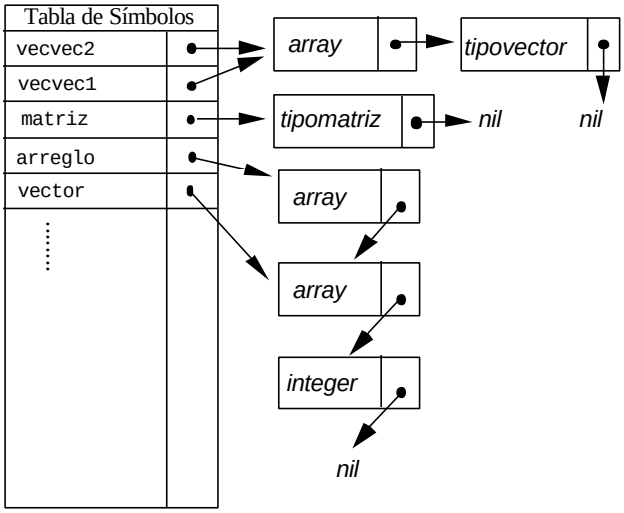


Al diseñar el compilador debe tenerse en cuenta el modo de equivalencia de tipos que se desea, asi como la estructura de datos que debe contener la expresión de tipos. Una opción consiste en almacenar en un campo de la tabla de símbolos una estructura de datos distinta para cada elemento del programa. Otra alternativa consiste en almacenar una macro-estructura de forma arborescente común a todos los símbolos del programa y a la que apunta el campo correspondiente a cada uno de los símbolos de la tabla de símbolos del compilador.

Ejemplo: Para el programa PASCAL del ejemplo del epígrafe anterior, supueto que se define un sistema de tipos con equivalencia estructural y sin comprobación de rangos de las matrices se obtiene la siguiente represenatación compacta de las estructuras de tipos de las variables que intervienen en el programa:



El mismo ejemplo en el caso de que el sistema requiera equivalencia estructural de tipos puede utilizar la siguiente estructura de datos:



En cualquier caso para comprobar que dos tipos son iguales basta verificar que es igual la dirección a la que apuntan los punteros del campo de la tabla de símbolos correspondiente al tipo.

(3) DAGS. Son una forma abreviada de representar los árboles. Se obtienen cuando varias ramas de un árbol señalan a la misma subestructura. Todo lo dicho para representaciones arbóreas es valido para representaciones mediante DAGS. La principal ventaja que presentan es que son representaciones más compactas, por tanto, ocupan menos memoria y se la comprobación de equivalencia se efectua con mayor rapidez. Por contra, su construcción es algo mas complicada.

Ejemplo: La expresión de tipos del ejemplo del apartado anterior puede representarse mediante el siguiente DAG: