Tema 5
Tabla de Símbolos

También se la llama tabla de nombres o tabla de identificadores y tiene dos funciones principales:

- Efectuar chequeos semánticos.
- Generación de código.

Permanece sólo en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración.

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc. La gestión de la tabla de símbolos es muy importante, ya que consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica. Aunque también sirve para guardar información referente a los tipos creados por el usuario, tipos enumerados y, en general, a cualquier identificador creado por el usuario, nos vamos a centrar principalmente en las variables de usuario. Respecto a cada una de ellas podemos guardar:

- *Almacenamiento del nombre.*
  Se puede hacer con o sin límite. Si lo hacemos con límite, emplearemos una longitud fija para cada variable, lo cual aumenta la velocidad de creación, pero limita la longitud en unos casos, y desperdicia espacio en la mayoría. Otro método es habilitar la memoria que necesitemos en cada caso para guardar el nombre. En C esto es fácil con los char *. Si hacemos el compilador en MODULA-2, por ejemplo, habría que usar el tipo ADDRESS.

- *El tipo* también se almacena en la tabla, como veremos en un apartado dedicado a ello.

- *Dirección de memoria en que se guardará.*
  Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber donde encontrar el valor de esa variable en tiempo de ejecución, también cuando se trata de variables globales. En lenguajes que no permiten recursividad, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del bloque de datos de ese bloque, (función o procedimiento) en concreto.

- *El número de dimensiones de una variable array, o el de parámetros de una función o procedimiento* junto con el tipo de cada uno de ellos es útil para el chequeo semántico. Aunque esta información puede extraerse de la estructura de tipos, para un control más eficiente, se puede indicar explícitamente.

- También podemos guardar información de *los números de línea en los que se ha usado un identificador, y de la línea en que se declaró.*

✯ Consideraciones sobre la Tabla de Símbolos.

La tabla de símbolos puede iniciarse con cierta información útil, tal como:

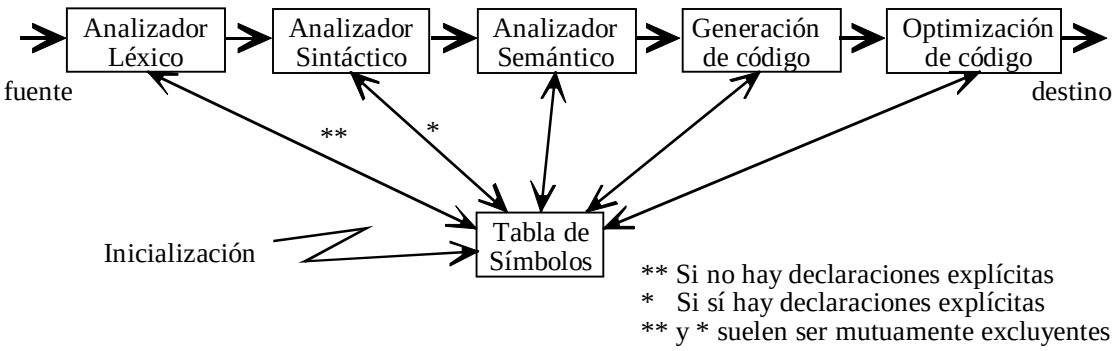
- Constantes: PI, E, etc.

- Funciones de librería: EXP, LOG, etc.

- Palabras reservadas. Esto facilita el trabajo al lexicográfico, que tras reconocer un identificador lo busca en la tabla de símbolos, y si es palabra reservada devuelve un token asociado. Bien estructurado puede ser una alternativa más eficiente al lex tal y como lo hemos visto (hash perfecto).

Conforme van apareciendo nuevas declaraciones de identificadores, el analizador léxico, o el analizador sintáctico según la estrategia que sigamos, insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas.

El analizador semántico efectúa las comprobaciones sensibles al contexto gracias a la tabla de símbolos, y el generador de código intermedio usa las direcciones de memoria asociadas a cada identificador en la tabla de símbolos, al igual que el generador de código. El optimizador de código no necesita hacer uso de ella



La tabla de símbolos contiene información útil para poder compilar, por tanto existe en tiempo de compilación, y no de ejecución.

Sin embargo, en un intérprete, dado que la compilación y ejecución se producen a la vez, la tabla de símbolos permanece todo el tiempo.

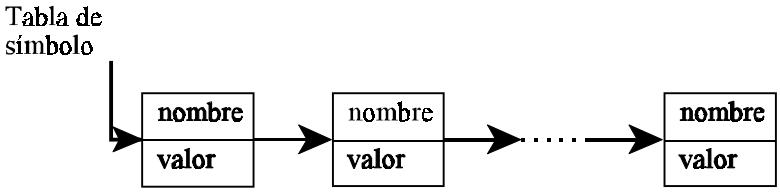
Ejemplo

Vamos a hacer un intérprete. Recordar que en un intérprete la entrada es un programa y la salida es la ejecución de ese programa.

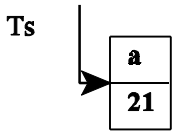
Suponemos que queremos hacer las siguientes operaciones:

a = 7 * 3
b = 3 * a

En la segunda instrucción necesitamos saber cuanto vale 'a'; es decir el valor de 'a' debe estar guardado en algún sitio. Para ello utilizaremos una lista de pares:

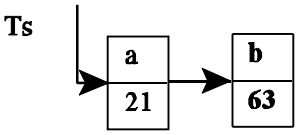


de forma que cuando nos encontremos con la instrucción a = 7 * 3, miremos en la tabla, si no está 'a' en la tabla, creamos un nodo para introducirla.

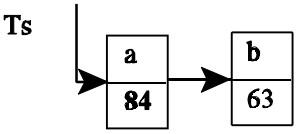


A continuación nos encontramos con b = 3 * a. ¿Qué es 'a'? Busco en la tabla de símbolos y vemos que el valor de 'a' es 21.

b = 3 * a Ahora buscamos b en la tabla de símbolos y como no está lo creamos.



Si ejecutáramos ahora la instrucción: a = a + b
Tendríamos a 'a' y 'b' en la tabla de símbolos con lo cual solo tendríamos que modificar el valor de 'a'.



Como hemos dicho una tabla de símbolos es una estructura que almacena información relativa a los identificadores de usuario. Pero, ¿que información exactamente?, según el propósito que tengamos. En nuestro caso el proposito es hacer un intérprete y nos interesa mantener información, entre otras cosas, del valor de cada variable.

Ejemplo

- En la implementación del ejemplo completo que veremos más adelante, para trabajar con la tabla de símbolos nos creamos un TAD lista. (Nosotros utilizaremos una lista por simplicidad. Lo mejor es usar otra estructura más eficiente. Por ejemplo un árbol AVL). La tabla de símbolos consta de una estructura llamada símbolo. Las operaciones que puede realizar son:
  - *crear* : Crea una tabla vacía.
  - *insertar*: Parte de una tabla de símbolo y de un nodo, lo que hace es añadir ese nodo a la cabeza de la tabla.
  - *buscar*: Busca el nodo que contiene el nombre que le paso por parámetro.
  - *imprimir*: Devuelve una lista con los valores que tiene los identificadores de usuario, es decir recorre la tabla de símbolos. Este procedimiento no es necesario pero se añade por claridad, y a efectos de resumen y depuración

- El programa lex :
  Si se encuentra un número lo convierte a *int* y devuelve el token NUMERO.
  Si se encuentra un identificador de usuario:
  Primero lo busca en la tabla de símbolos,
  Si lo encuentra, entonces devuelve su valor
  Si no lo encuentra, lo inserta
  ¿Porqué no ignoro el retorno de carro?- Porque forma parte de la gramática (Observar el fichero ejem2y.yac). Lex no puede ignorar el retorno de carro

- El programa Yacc:

```
%union {
        int numero;
        simbolo * ptr_simbolo;
         }
```

  Esto es un registro con parte variante, en el que todo es parte variante. Si ponemos %union ya no hay que poner YYSTYPE.
  Cosas interesantes de la gramática:

```
prog    : prog asig
        ...
        |       (ε forma parte de un prog).
        ;
asig    : ID ASIG expr
        | ID ASIG asig
```

  Como podemos observar esta regla es recursiva a la derecha. Se permiten cosas como
  a := b := c := 3 * 4
  En cuanto a la regla *expr* produce ambigüedad, ésta se soluciona introduciendo las instrucciones

```
% left '+'
% left '*'
```

  El atributo del identificador de usuario 'ID', es un puntero a un símbolo.

Fichero TabSimb2.c

```
#include <stdlib.h>
#include <stdio.h>

typedef struct nulo
        {
                struct nulo * sig;
                char nombre [20];
                int valor;
        } simbolo;

simbolo * crear()
{
        return NULL;
};

void insertar(p_t,s)
simbolo **p_t;
simbolo * s;
{
        s->sig = (*p_t);
        (*p_t) = s;
};

simbolo * buscar(t,nombre)
simbolo * t;
char nombre[20];
{
        while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
                t = t->sig;
        return (t);
};

void imprimir(t)
simbolo * t;
{
        while (t != NULL)
                {
                        printf("%s\n", t->nombre);
                        t = t->sig;
                }
};
```

Fichero Ejem2l.lex

```
%%
[0-9]+              {
                       yylval.numero = atoi(yytext);
                       return NUMERO;
                    }

":="                {   return ASIG; }

[a-zA-Z][a-zA-Z0-9]*{
                       yylval.ptr_simbolo = buscar(t,yytext);
                       if (yylval.ptr_simbolo == NULL)
                        {
                            yylval.ptr_simbolo=(simbolo *) malloc(sizeof(simbolo));
                            strcpy(yylval.ptr_simbolo->nombre, yytext);
                            yylval.ptr_simbolo->valor=0;
                            insertar(&t, yylval.ptr_simbolo);
                         }
                       return ID;
                    }

[ \t]+              {;}
.|\n                {return yytext[0];}
```

Fichero Ejem2y.yac

```
%{
#include "tabsimb2.c"
simbolo * t;
%}
%union {
        int numero;
        simbolo * ptr_simbolo;
         }
%token <numero> NUMERO
%token <ptr_simbolo> ID
%token  ASIG
%type <numero> expr asig prog
%start prog

%left '+'
%left '*'
%%
prog    :       prog asig  '\n'  { printf("Asignaciones efectuadas\n");}
        |       prog expr  '\n'  { printf("%d\n",$2);}
        |       prog error '\n'  { yyerrok;}
        |
        ;
asig    :       ID ASIG expr {
                                $$ = $3;
                                $1->valor = $3;
                             }
         |      ID ASIG asig  {
                                $$ = $3;
                                $1->valor = $3;
                             }
         ;
expr    :       expr '+' expr   {$$ = $1 + $3;}
        |       expr '*' expr   {$$ = $1 * $3;}
        |       ID              {$$ = $1->valor; }
        |       NUMERO          {$$ = $1;}
        ;
%%
#include "ejem2l.c"
#include "errorlib.c"
void main()
{       t = crear();
        yyparse ();
        imprimir(t);
 }
```