



**INSTITUTO POLITÉCNICO NACIONAL**  
**ESCUELA SUPERIOR DE CÓMPUTO**



COMPILADORES

3CM7

---

# PRÁCTICA 03

---

**Alumno:**

Díaz Medina Jesús Kaimorts

**Boleta:**

2016350140

**Profesor:**

Tecla Parra Roberto

Calculadora de Vectores

Implementación de Tabla de Símbolos

09 de octubre 2018

## Objetivo

Completar un programa que con ayuda de YACC y el HOC3 (Calculadora científica con variables), la cuál trabajará sobre ella los 4 tipos de operaciones básicas y también si es posible se agregará bltin.

## Introducción

### Tabla de símbolos.

- También llamada *Tabla de nombres* o *Tabla de identificadores*, tiene dos funciones principales.
  1. Ejecutar chequeos semánticos.
  2. Generación de código.

Permanece sólo en tiempo de compilación y 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 de creadores 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 se pueden guardar:

#### a) Almacena el nombre.

Se puede hacer con o sin límite. Si se hace con límite, se emplea una longitud fija para cada variable, lo cuál 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 se necesita en cada paso para guardar el nombre. En lenguaje C es fácil con los **char \***. Si hacemos el compilador MODULA-2, por ejemplo, habría que usar el tipo ADDRESS.

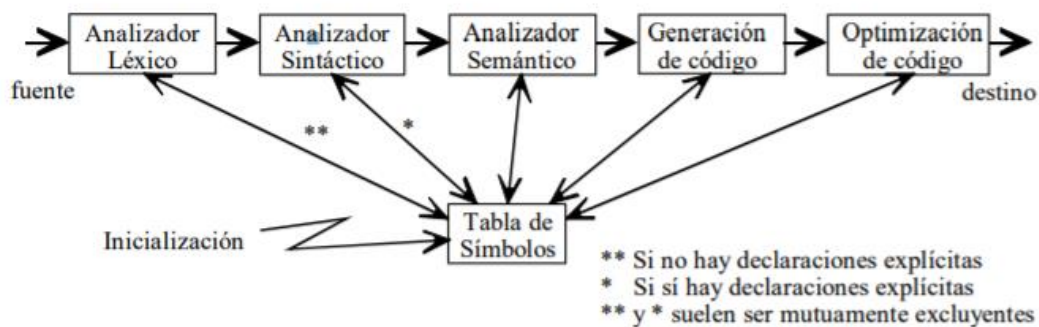
- b) El tipo también se almacena en la tabla, tal y como veremos en un apartado dedicado a ellos.

#### c) Dirección de memoria.

Aquí es donde se guardará. Esta dirección es necesario por que las instrucciones que referencian a una variable deben saber dónde encontrar el valor de esas variables 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 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.

- d) El número de dimensiones de una variable, array, o el de parámetro 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 control más eficiente, se puede indicar explícitamente.



## Desarrollo

Agregar una tabla de símbolos para permitir el nombre de variables de más de una letra y agregar a la gramática la producción para el operador de asignación.

Usar una lista simplemente ligada (en lenguaje C o Java) o una tabla hash (de las librerías de Java).

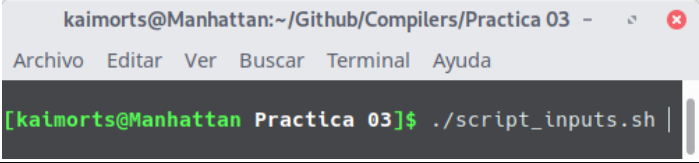
### Calculadora para vectores.

Las variables servirán para guardar los vectores.

### Entrada.

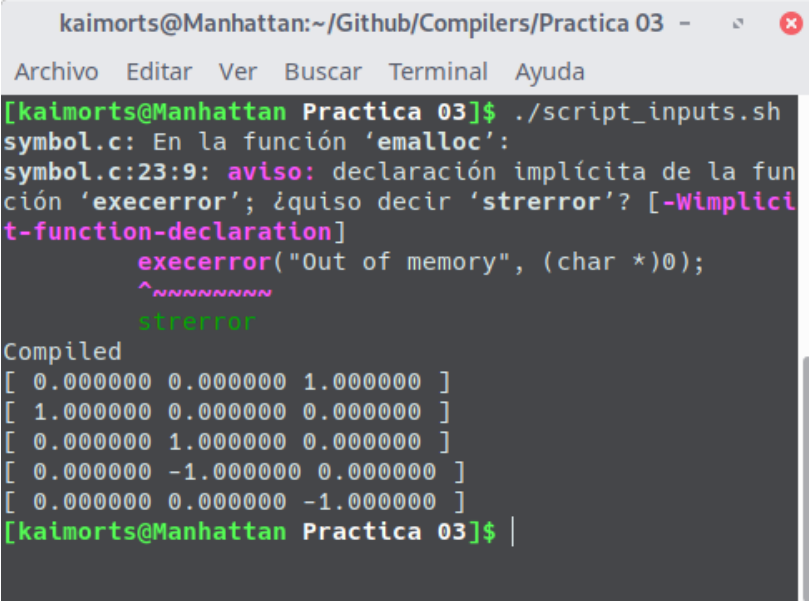
- Se creó un archivo llamado *inputs.txt* el cuál servirá como entrada estándar de las operaciones pertinentes de las operaciones previamente implementadas en la Práctica 01, como lo son: Producto cruza, Producto Punto, Suma, Resta, etc. Además, en este archivo se realizará la asignación de variables que guardarán los vectores.

Asimismo, se creó un script el cuál ejecutará el programa considerando las entradas del archivo *.txt* previamente mencionado.

<pre>i=[1 0 0] j=[0 1 0] k=[0 0 1] i#j j#k k#i i#k j#i</pre>	<pre>#!/bin/bash make ./vec &lt; inputs.txt</pre> 
inputs.txt	Figura 1. script_inputs.sh

### Salida.

La salida de las respectivas operaciones realizadas se muestra a continuación.



```
kaimorts@Manhattan:~/Github/Compilers/Practica 03 -
Archivo Editar Ver Buscar Terminal Ayuda
[kaimorts@Manhattan Practica 03]$ ./script_inputs.sh
symbol.c: En la función 'emalloc':
symbol.c:23:9: aviso: declaración implícita de la función 'execerror'; ¿quiso decir 'strerror'? [-Wimplicit-function-declaration]
      execerror("Out of memory", (char *)0);
      ^~~~~~
      strerror
Compiled
[ 0.000000 0.000000 1.000000 ]
[ 1.000000 0.000000 0.000000 ]
[ 0.000000 1.000000 0.000000 ]
[ 0.000000 -1.000000 0.000000 ]
[ 0.000000 0.000000 -1.000000 ]
[kaimorts@Manhattan Practica 03]$
```

Figura 2. Salidas del programa con la Tabla de Símbolos implementada.

### Gramática.

Se implementaron las declaraciones y la sección de reglas gramaticales necesarias para poder agregar la tabla de símbolos que permitirá usar variables más de una variable. Además, se cambiaron los símbolos en la definición de asociatividad del producto punto y cruz: 'o' y 'x', por los símbolos ':' y '#'. Además, se modificó la pila de YACC, que es de tipo unión, agregando la estructura **Symbol**.

Declaraciones		
%union{ double num; Vector * val; Symbol * sym; }	/* Declaración de YACC*/ %token <num> NUMBER %token <sym> VAR INDEF %type <val> vector %type <val> expr asgn %type <num> number	/* Asociatividad */ %right '=' %left '+' '-' %left '*' %left UNARYMINUS %left ':' '#'

**Sección de reglas.**

```

%%
list:
| list '\n'
| list asgn '\n'
| list expr '\n' { imprimeVector($2); }
| list number '\n' { printf("\t%.8g\n", $2); }
| list error '\n' { yyerror; }
;

asgn: VAR '=' expr { $$ = $1->u.val = $3;
                    $1->type = VAR; }
;

expr: vector { $$ = $1; }
| VAR { if( $1->type == INDEF )
        execerror("Variable no definida", $1->name);
        $$ = $1->u.val;
      }
| asgn
| expr '+' expr { $$ = sumaVector ( $1, $3 ); }
| expr '-' expr { $$ = restaVector( $1, $3 ); }
| NUMBER '*' expr { $$ = escalarVector( $1, $3 ); }
| expr '*' NUMBER { $$ = escalarVector( $3, $1 ); }
| expr '#' expr { $$ = productoCruz( $1, $3 ); }
;

number: NUMBER
| expr ':' expr { $$ = productoPunto( $1, $3 ); }
| '|' expr '|' { $$ = magnitudVector( $2 ); }
;

vector: '[' NUMBER NUMBER NUMBER ']' { $$ = creaVector(3);
                                       $$->vec[0] = $2;
                                       $$->vec[1] = $3;
                                       $$->vec[2] = $4;
                                       }
;
%%

```

**Manejador de Expresiones Regulares**

```

int yylex(){
int c;
while ((c = getchar()) == ' ' || c == '\t')
;
if (c == EOF)
return 0;

if (c == '.' || isdigit(c) ) {
ungetc(c, stdin);
scanf("%lf\n", &yylval.num);
return NUMBER;
}

if (isalpha(c)) {
Symbol * s;
char sbuf[200];

```

```

char * p = sbuf;
do {
    *p++=c;
} while((c = getchar()) != EOF && isalnum(c));

ungetc(c, stdin);
*p = '\0';
if ((s = lookup(sbuf)) == (Symbol *)NULL)
    s = install(sbuf, INDEF, NULL);
yyval.sym = s;

if (s->type == INDEF)
    return VAR;
else{
    //printf("func=(%s) tipo=(%d) \n", s->name, s->type);
    return s->type;
}
}
return c;
}

```

### Tabla de Símbolos.

```

static Symbol * symb_list = 0; /* Tabla de simbolos : Lista ligada */

Symbol * lookup(char * s){ /* Encontrar 's' en la Tabla de simbolos */
    Symbol * sp;
    for (sp = symb_list ; sp != (Symbol *)0; sp = sp->next)
        if ( strcmp( sp->name, s) == 0 )
            return sp;
    return 0; /* 0 ==> No se encontró */
}

char * emalloc(unsigned n){ /*Se revisa el regreso desde malloc*/
    char * p;
    p = malloc(n);
    if (p == 0)
        execerror("Out of memory", (char *)0);
    return p;
}

Symbol * install(char * s, int t, Vector * d){ //Se instala 's'
    Symbol * sp; //en la tabla de simbolos
    char * emalloc();
    sp = (Symbol *)emalloc(sizeof(Symbol));

    sp->name = emalloc( strlen(s) + 1); /* '\0' es +1 */
    strcpy( sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symb_list; /* Se pone al frente de la lista */
    symb_list = sp;

    return sp;
}

```

## Librerías.

En el archivo ***vector\_calc.c*** se implementaron funciones que nos ayudarán a validar ya asignar cada uno de los símbolos la tabla de símbolos.

```
...
#define MAX 102
#define ASCII 26
Vector * cubeta[MAX];
/*Operaciones con vectores*/
...
int obtenID(char var){
    if ( 'A' <= var && var <= 'Z' )
        return ( var - 'A' );
    return ( var - 'a' + ASCII );
}
Vector * obtenValor(char var){
    int ID = obtenID(var);
    Vector * c = cubeta[ ID ];
    return c;
}

void actualizaValor(char var, Vector * a){
    int ID = obtenID(var);
    cubeta[ ID ] = a;
}
```

## Conclusión

---

El HOC 3 nos permite almacenar variables a través del uso de una tabla de símbolos donde se almacena el nombre de las variables, para poder después hacer referencia a ellos y trabajar las variables y las operaciones básicas y no tan básicas, esta ya es la calculadora científica.