



Tarea INFO165

Compiladores: Traductor dirigido por Sintaxis

Integrantes:
Sebastian Duran
Mauricio Romero
Diego Vera

Asignatura: Compiladores INFO165
Profesora: Maria Eliana de la Maza

10/12/2020

Introducción

En el presente trabajo se ha solicitado desarrollar un traductor dirigido por sintaxis que permite ejecutar programas escritos en lenguaje LMA (Lenguaje manejador de arreglos). El traductor deberá aceptar una secuencia de instrucciones del lenguaje manejador de arreglos y efectuar las acciones a medida de que se realiza el análisis sintáctico. Para esto se debe realizar las etapas de análisis léxico y análisis sintáctico del programa a realizar, junto a la traducción dirigida por sintaxis, para ello se necesitará del generador de analizadores léxico Flex, el generador de análisis sintáctico Bison y el lenguaje de programación C.

El lenguaje manejador de arreglos, contiene las siguientes instrucciones:

- PARTIR: primera instrucción de cualquier programa.
- INICIAR(nomarr,e1,e2,e3,...,e8): crea un arreglo llamada nomarr con los elementos e1,e2,e3,...,e8 (números enteros positivos). Si el arreglo tiene menos de 8 elementos el resto se completa con ceros.
- METER(nomarr,x,y): en nomarr, inserta el elemento x en la posición y.
- SACAR(nomarr,y): en nomarr, elimina el elemento que se encuentra en la posición y .
- MIRAR(nomarr): despliega en pantalla los elementos de nomarr.
- DATO(nomarr,x): despliega en pantalla el elemento de nomarr, que se encuentra en la posición x.
- FINALIZAR: última instrucción de cualquier programa.

1. Desarrollo

Para realizar el desarrollo del trabajo fue imprescindible hacer uso de dos herramientas que nos permitieron cumplir con los objetivos solicitados, dichas herramientas se describirán brevemente a continuación:

1.1 Flex

Flex es una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones.

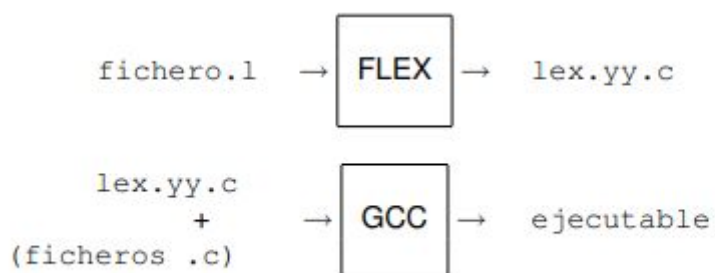


imagen 1.Creación de un analizador léxico con Flex

1.1.1 Pasos de compilación

a) Compilar la especificación del analizador y crea el fichero `yy.lex.c` a través del comando:
`$flex fichero.l`

b) Compilar el analizador C y generar el programa ejecutable, para ello se debe enlazar con la librería de FLEX la cual proporciona implementaciones por defecto `yywrap()` y `main()`. Además se debe compilar y enlazar con `ficheros.c` del usuario mediante el comando:
`$gcc lex.yy.c`, del cual se obtiene como salida un archivo ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

1.1.2 Partes de una especificación en Flex

Un fichero está compuesto de tres secciones, las cuales cumplen un rol específico en el mismo, dichas secciones se encuentran separadas por una línea que contiene el símbolo `"%%"` :

1. Sección de definiciones: La sección de definiciones contiene declaraciones de definiciones de nombres sencillas para simplificar la especificación del escáner, y declaraciones de condiciones de arranque.
2. Sección de reglas y acciones: Contiene una serie de reglas de la forma **patrón {acción}**, en donde los patrones son escritos como una series de expresiones regulares mientras que la acción corresponde a código C que se ejecuta al momento de encontrar una concordancia entre el patrón con el texto de entrada.
3. Sección de rutina de usuario:se copia a 'lex.yy.c' . Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. La presencia de esta sección es opcional

```
< sección de declaraciones >

% %

< sección de reglas y acciones >

% %

< sección de rutinas de usuario >
```

imagen 2.Secciones de fichero Flex

Una vez ejecutado el analizador Léxico Flex se obtendrá como salida un archivo lex.yy.c ,en donde se encuentra la función yylex() la cual contiene tablas que son usadas para emparejar tokens cada vez que la función yylex() es llamada, cuando esto ocurre dicha función se encarga de analizar los tokens desde el fichero de entrada global yyin. La función continua hasta alcanzar el final del fichero o hasta que se ejecute una sentencia "return".

1.2 Bison

Bison es una herramienta que traduce la especificación de una gramática de contexto libre a un programa en C implementa un analizador LALR(1) que reconoce frases de esa gramática. Además de reconocer frases se puede asociar código C a las reglas de la gramática, lo que permite especificar acciones semánticas, que se ejecutarán cada vez que se aplique la regla correspondiente.

1.2.1 Pasos de Compilación

- a) Compila la especificación del analizador y crea el fichero fichero.tab.c con el código y las tablas del analizador LALR(1) mediante: `$bison.fichero.y`
- b) Al ingresar `$bison -d fichero.y` en el terminal se genera fichero.tab.h (mediante la opción -d) con las definiciones de las constantes asociadas a los tokens, además de variables y estructuras de datos necesarias para el analizador léxico

- c) El usuario deberá de proporcionar sus propias funciones `main()`, `yyerror()` y `yylex()`. Dentro del código de usuario se deberá llamar a la función `yyparse()` que a su vez llamará a la función `yylex()` del analizador léxico cada vez que necesite un token.

Finalmente se ejecuta el comando para ejecutar: `$gcc fichero.tab.c`

Otra opción para usar conjuntamente Flex y Bison es mediante el uso de los siguientes tres comandos:

```
$bison -d fichero.y
```

```
$flex fichero.l
```

```
$gcc lex.yy.c fichero.tab.c -o salida -lm
```

Cabe mencionar que para la compilación del presente trabajo se ha decidido utilizar la última opción descrita.

1.2.2 Partes de una especificación en Bison

Tiene tres estructuras separadas por el símbolo “%%”, a continuación se describe cada una de ellas:

1. Sección de declaraciones: Aquí se pueden incluir código C necesario para las acciones semánticas asociadas a las reglas el cual será copiado en `fichero.tab.c`, también se pueden definir tokens de la gramática y no terminales.
2. Sección de reglas: son las producciones de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.
3. Sección de rutinas de usuarios: En esta sección zona se puede escribir código C adicional, bien funciones llamadas desde las acciones de las reglas o, en el caso de programas pequeños, suelen incluirse las funciones `main()`, `yyerror()` e `yylex()` propias del usuario.

```
<sección de declaraciones>
```

```
%%
```

```
<sección de reglas y acciones>
```

```
%%
```

```
<sección de rutinas de usuario>
```

imagen3.Secciones de especificación Bison

1.2.3 Integración con el analizador Léxico FLEX

Los analizadores Bison necesitan una función llamada ‘`yylex()`’ para devolverles el siguiente token de la entrada. Esa función devuelve el tipo del próximo token y además puede poner

cualquier valor asociado en la variable global yyval. Para usar Flex con Bison, normalmente se especifica la opción -d de Bison para que genera el fichero 'y.tab.h' que contiene las definiciones de todos los '%tokens' que aparecen en el fuente Bison. Este fichero de cabecera se incluye después en la fuente de Flex.

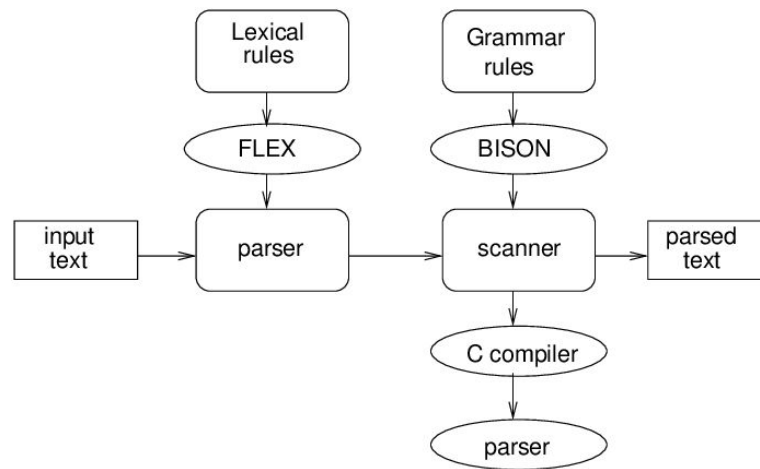


imagen 4. Estructura de analizador desarrollado con Flex y Bison

2 Especificación del programa

Se crearon los ficheros "flex.l" donde definimos el léxico del analizador y "sintactico.y" donde definimos la gramática a utilizar para el analizador sintáctico para la compilación en flex y bison respectivamente.

2.1 Gramática a Utilizar

s : PARTE INST FINALIZA (1)

INST: INST INST| (2)

INICIA |METE | (3)

SACA | (4)

MIRA | (5)

DAT (6)

INICIA:

iniciar PARA arr COMA constante COMA constante COMA constante COMA
constante COMA constante COMA constante COMA constante COMA constante
PARC |(7)

iniciar PARA arr COMA constante COMA constante COMA constante COMA
constante COMA constante COMA constante COMA constante COMA constante PARC |(8)

iniciar PARA arr COMA constante COMA constante COMA constante COMA
constante COMA constante COMA constante PARC |(9)

iniciar PARA arr COMA constante COMA constante COMA constante COMA
constante COMA constante PARC| (10)

iniciar PARA arr COMA constante COMA constante COMA constante COMA
constante PARC |(11)

iniciar PARA arr COMA constante COMA constante COMA constante PARC|(12)

iniciar PARA arr COMA constante COMA constante PARC |(13)

iniciar PARA arr COMA constante PARC (14)

METE: meter PARA NOMARR COMA ENTERO COMA ENTERO PARC (15)

SACA: sacar PARA NOMARR COMA ENTERO PARC (16)

MIRA: mirar PARA NOMARR PARC (17)

DAT: dato PARA NOMARR COMA ENTERO PARC (18)

PARTE: partir (19)

FINALIZA: finalizar (20)

arr: NOMARR (21)

constante: ENTERO (22)

NOMARR, ENTERO e ID fueron definidos previamente en el analizador léxico flex “flex.l” y corresponden a los siguientes valores:

NOMAR "L"{letras}*{digito}+

ENTERO [0-9]+

ID [a-z][a-zA-Z0-9]+

letras [a-zA-Z]

digito [0-9]

Donde NOMARR es un identificador para los arreglos que siempre comienzan con la letra “L”.

ENTERO es un número positivo incluyendo el cero.

2.2 Estructura de datos

Se decidió utilizar utilizar como estructura de datos un arreglo de registros para almacenar las variables necesarias, ya que de esta manera se puede tener acceso a un dato específico a través de su posición de manera sencilla, además de limitar el almacenamiento y poder tener una referencia para los requisitos mínimos del analizador.

1. Estructura **arreglo**: Registro con que almacena una carácter y un arreglo de tamaño 8 en donde se almacenan los números enteros que ingresa el usuario.

```
struct arreglo:
{
    char nombre[50];
    int valor[8];
};
```

2. Estructura arreglo **Buffer**: Arreglo de tamaño 1000 en donde se almacenarán los registros con la Estructura arreglo (nombre del arreglo con sus 8 valores, además de un carácter que será del tipo NOMARR).

```
struct arreglo buffer[1000]
```

2.3 Funciones y Métodos

- **void identificadorArreglo(char id[50]):** Transforma un carácter defectuoso dejando solamente el id.
- **void creaArreglo(char id[50], int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8):** Busca un espacio de memoria vacío en el buffer para almacenar una carácter y 8 números enteros definidos por el usuario los cuales serán contenidos en un arreglo tipo entero de tamaño 8 (va de la posición 1 a la posición 8). Si un usuario decide ingresar menos de 8 números enteros provocará que las posiciones restantes del arreglo sean rellenas con ceros.
- **int buscaPosicionArreglo(char id[50]):** Se encarga de retornar el valor que tiene almacenado un id en la tabla de buffer cuando es referenciado.
- **void meterEnArreglo(char id[50], int x, int y):** Se inserta el valor "x" en la posición "y" del arreglo asociado al carácter, haciendo que todo los elementos que

originalmente se encontraban ubicados entre la posición “y” y el final de la lista se desplacen en una posición hacia la derecha .

- **void mirarArreglo(char id[50]):** Despliega en pantalla los números enteros del arreglo asociados al carácter ingresado por el usuario.
- **void sacarDeArreglo(char id[50],int y):** Se elimina el elemento “y” del arreglo asociado al carácter ingresado por el usuario.
- **void datoDeArreglo(char id[50],int x):** Se despliega en pantalla el valor ubicado en la posición x para el carácter ingresado por el usuario.

2.4 Funcionamiento del programa

Una vez definido el léxico del analizador en el archivo “.l”, usamos la gramática del lenguaje para definir el analizador sintáctico en el archivo “.y”. Esto con sus respectivas funciones o métodos para almacenar los arreglos en el buffer y poder hacer operaciones de la siguiente forma.

La producción **INICIA** hace uso de la función **creaArreglo(char id[50], int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8)**.

La producción **METE** hace uso de la función **meterEnArreglo(char id[50], int x, int y)**.

La producción **SACA** hace uso de la función **sacarDeArreglo(char id[50],int y)**.

La producción **MIRA** hace uso de la función **mirarArreglo(char id[50])**.

La producción **DAT** hace uso de la función **datoDeArreglo(char id[50],int x)**.

Adicionalmente las siguientes funciones se apoyan de la función **buscaPosicionArreglo(char id[50]):**

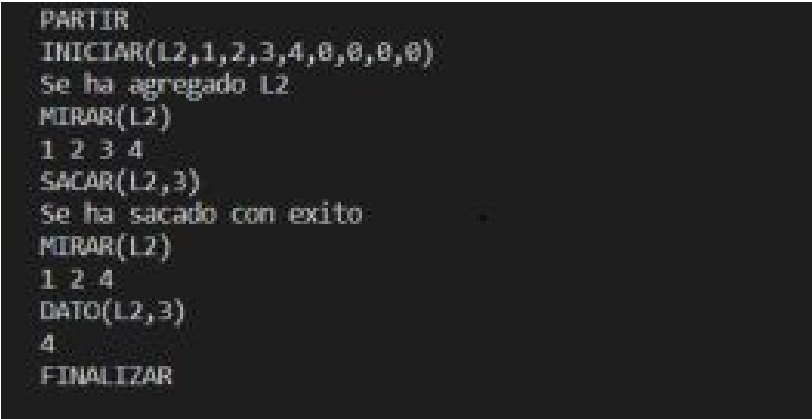
- **meterEnArreglo(char id[50], int x, int y)**
- **mirarArreglo(char id[50])**
- **sacarDeArreglo(char id[50],int y)**
- **datoDeArreglo(char id[50],int x)**

Mientras que todas las producciones usan **identificadorArreglo(char id[50])**.

Por último para poder **ejecutar el programa**. usamos los siguientes comandos en el siguiente orden como ejemplo.

```
$bison sintactico.y
$flex flex.l
$gcc lex.yy.c sintactico.tab.c -o analizador -lm
$./analizador
```

Se ejecutará en la terminal el analizador donde se permitirá ingresar las entradas con el léxico y sintáctico utilizado de la siguiente forma como ejemplo:



```
PARTIR
INICIAR(L2,1,2,3,4,0,0,0,0)
Se ha agregado L2
MIRAR(L2)
1 2 3 4
SACAR(L2,3)
Se ha sacado con éxito
MIRAR(L2)
1 2 4
DATO(L2,3)
4
FINALIZAR
```

** En el terminal no reconoce la tilde

2.5 Requerimientos de Hardware y Software

Requerimiento de software:

- Compilador flex 2.5.35
- Compilador bison (GNU Bison) 2.3

El programa desarrollado fue ejecutado en notebooks con las siguientes características:

Notebook 1:

Sistema operativo: Windows 10

RAM:8GB 3200 Hz

Procesador:ryzen 5 4600 H de 6 núcleos 12 hilo 3 a 4 Ghz

Notebook 2:

Sistema operativo: MacOS Catalina

RAM: 8GB 2133 Hz

Procesador: 2,3 GHz Intel Core i5 de 2 núcleos

Notebook 3:

Sistema operativo:Ubuntu versión 18.04

RAM:8GB 2666 Hz

Procesador: Intel core i5 9300H de 4 núcleos

Requerimientos de Hardware:

- Procesador de 1 GHz (por ejemplo Intel Celeron) o mejor.
- 1 GB de RAM (Memoria del SO utilizado por Windows 10 y Ubuntu 18.04) y 4GB de RAM (Memoria del SO utilizado por MacOS Catalina).
- Acceso a internet para instalar los paquetes de Bison y Flex.

Problemas que surgieron al desarrollar el proyecto

- Problemas al manejar las variables de entrada de las funciones en la declaración de la gramática, ya que no estábamos ingresando los valores correctamente con los signo \$. Sin embargo se pudo solucionar y obtener un correcto funcionamiento de las funciones.
- Problemas de conflicto de avance y reducción en la producción Inst->Inst Inst de nuestra gramática que en la ejecución no hace problema.
- Problemas al ingresar el identificador de un arreglo ya que las comas y paréntesis se estaban agregando al identificador. Ejemplo en la línea de ejecución INICIAR(L1,1,2,3) el identificar pasaba como L1,1. Sin embargo lo solucionamos con la función **identificadorArreglo(char id[50])** que se apoya de un auxiliar para dejar el nombre como corresponde.

Conclusiones

Como grupo, aprendimos el uso de las disciplinas pasadas como estructuras de datos, estructuras discretas, autómatas y finalmente compiladores, para la construcción de un lenguaje, además de la importancia del buen uso del léxico y de la parte sintáctica tanto para el programador como para una máquina.

En el camino, como se habló anteriormente, surgieron diversos problemas, que fueron causadas de la inexperiencia de nosotros los desarrolladores, como problemas en el código que era el caso del identificador del arreglo.

Por otro lado, en el código en general pueden ser mejoradas en varios aspectos, como el uso de una gramática más flexible, en que ya no tenga un límite determinado de elementos que pueda soportar un arreglo, en qué implementar una gramática recursiva (que no sea recursividad izquierda).

Que pueda ser flexible en la cantidad de arreglos que puede almacenar y el tamaño del identificador de los arreglos, en que puede ser implementado por una estructura dinámica

como una lista doblemente enlazadas, para un mejor manejo de las búsquedas de los elementos.

Y por último la opción de implementar la asignación de valores enteros a un identificador, que fue pensada (como se muestra en el léxico y el sintáctico con la estructura de ID) pero por el fin del trabajo no se implementó ya que no está mencionado en los objetivos del proyecto y se ahorraría espacio en la memoria porque se tendría que definir un buffer especial para ese tipo de variables, por otra parte dejamos menos compleja la inicialización de un arreglo ya que está aceptando valores enteros y no variables.

Bibliografía Utilizada

http://webdiis.unizar.es/asignaturas/LGA/docs_externos/flex-es-2.5.pdf

<https://engineering.purdue.edu/~milind/ece573/2011fall/project/bison.html>

Anexo 1

lexico.l

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "sintactico.tab.h"

int lineno = 1; // initialize to 1

//void ret_print(char *tipo_token);

//void yyerror();

%}

%x ML_COMMENT

letras [a-zA-Z]

digito [0-9]
ID [a-z][a-zA-Z0-9]+

NOMAR "L"{letras}*{digito}+
ENTERO [0-9]+

%%

```
{ENTERO} {yyval.numero=atoi(yytext); return(ENTERO);}
{ID} {return (ID);}
{NOMAR} {yyval.texto=yytext; return (NOMARR);}
```

```
<ML_COMMENT>"\n" { lineno += 1; }
```

```
"PARTIR" {return (partir);}
"INICIAR" {return (iniciar);}
"MIRAR" {return (mirar);}
"METER" {return (meter);}
"SACAR" {return (sacar);}
"DATO" {return (dato);}
"FINALIZAR" {return (finalizar);}
```

```
"(" {return (PARA);}
")" {return (PARC);}
"," { return (COMA);}
"=" {return '=';}
[ \t] ; /* ignora espacios */
"\n" { lineno += 1; }
[ \t\r\f]+ /*Borra espacio en blanco */
```

%%

```
/*
void ret_print(char *tipo_token){
    printf("yytext: %s\ttoken: %s\tlineno: %d\n", yytext, tipo_token, lineno);
}
```

```

void yyerror(char *mensaje){
    printf("Error: \"%s\" en linea %d. Token = %s\n", mensaje, lineno, yytext);
    exit(1);
}

int main(int argc, char *argv[]){
    printf("Ingrese Tokens(Palabras reservadas, enteros, símbolos, identificadores): ");
    yylex();
    return 0;
}

```

Anexo 2

sintáctico.y

```

%{

/*****
 * Declaraciones en C *
 *****/

//Importacion de librerias
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "string.h"

extern int yylex(void);
extern char *yytext;
extern FILE *yyin;

//Declaracion de metodos
void yyerror(char *s);
int buscaPosicionArreglo(char id[50]);
void creaArreglo(char id[50],int x1,int x2,int x3,int x4,int x5,int x6,int x7,int x8);
void meterEnArreglo(char id[50],int x,int y);
void sacarDeArreglo(char id[50],int y);
void mirarArreglo(char id[50]);
void datoDeArreglo(char id[50],int x);

```

```
void identificadorArreglo(char id[50]);
```

```
//Declaracion del nodo del buffer de identificadores
```

```
char auxiliarArreglo[50];
```

```
struct arreglo  
{  
    char nombre[50];  
    int valor[8];  
};
```

```
struct arreglo buffer[1000];  
int aux = 0;  
size_t n = sizeof(buffer) / sizeof(buffer[0]);
```

```
%}
```

```
/*  
Declaraciones de Bison *  
***/
```

```
%union  
{  
    int numero;  
    char* texto;  
}
```

```
/*Declaración de tokens*/  
%token <numero> ENTERO  
%token <texto> NOMARR  
%token <texto> ID  
%token partir  
%token iniciar  
%token meter  
%token sacar  
%token mirar  
%token dato  
%token asignar  
%token finalizar
```

%token PARA
%token PARC
%token COMA

%type <numero> constante
%type <texto> arr
%start s

%%

/*

* Reglas Gramaticales *

*/

/*Inicio de la gramatica*/
s :PARTE INST FINALIZA
;

INST: INST INST |
INICIA |
METE |
SACA |
MIRA |
DAT
;

INICIA:

iniciar PARA arr COMA constante COMA constante COMA constante COMA constante
COMA constante COMA constante COMA constante COMA constante PARC
{identificadorArreglo(\$3); creaArreglo(auxiliarArreglo,\$5,\$7,\$9,\$11,\$13,\$15,\$17,\$19);} |

iniciar PARA arr COMA constante COMA constante COMA constante COMA constante
COMA constante COMA constante COMA constante PARC {identificadorArreglo(\$3);
creaArreglo(auxiliarArreglo,\$5,\$7,\$9,\$11,\$13,\$15,\$17,0);} |

iniciar PARA arr COMA constante COMA constante COMA constante COMA constante
COMA constante COMA constante PARC {identificadorArreglo(\$3);
creaArreglo(auxiliarArreglo,\$5,\$7,\$9,\$11,\$13,\$15,0,0);} |

iniciar PARA arr COMA constante COMA constante COMA constante COMA constante
COMA constante PARC {identificadorArreglo(\$3);
creaArreglo(auxiliarArreglo,\$5,\$7,\$9,\$11,\$13,0,0,0);} |

iniciar PARA arr COMA constante COMA constante COMA constante COMA constante PARC
{identificadorArreglo(\$3); creaArreglo(auxiliarArreglo,\$5,\$7,\$9,\$11,0,0,0,0);} |

iniciar PARA arr COMA constante COMA constante COMA constante PARC
{identificadorArreglo(\$3); creaArreglo(auxiliarArreglo,\$5,\$7,\$9,0,0,0,0,0);} |

iniciar PARA arr COMA constante COMA constante PARC {identificadorArreglo(\$3);
creaArreglo(auxiliarArreglo,\$5,\$7,0,0,0,0,0,0);} |

iniciar PARA arr COMA constante PARC {identificadorArreglo(\$3);
creaArreglo(auxiliarArreglo,\$5,0,0,0,0,0,0,0);} |
;

METE: meter PARA NOMARR COMA ENTERO COMA ENTERO PARC {identificadorArreglo(\$3);
meterEnArreglo(auxiliarArreglo,\$5,\$7);} |
;

SACA: sacar PARA NOMARR COMA ENTERO PARC {identificadorArreglo(\$3);
sacarDeArreglo(auxiliarArreglo,\$5);} |
;

MIRA: mirar PARA NOMARR PARC {identificadorArreglo(\$3); mirarArreglo(auxiliarArreglo);} |
;

DAT: dato PARA NOMARR COMA ENTERO PARC {identificadorArreglo(\$3);
datoDeArreglo(auxiliarArreglo,\$5);} |
;

PARTE: partir |
;

FINALIZA: finalizar {exit (-1);} |
;

arr: NOMARR { \$\$ = \$1;} |
;

```
constante: ENTERO { $$ = $1;}  
;
```

```
%%
```

```
/******
```

```
* Codigo C Adicional *
```

```
*****/
```

```
void yyerror(char *s)
```

```
{  
    printf("Error sintactico %s \n",s);  
}
```

```
void identificadorArreglo(char id[50]){
```

```
    int i = 0;
```

```
    while( i < 50 && id[i] != ',' && id[i] !=')' ){
```

```
        auxiliarArreglo[i] = id[i];
```

```
        i++;
```

```
    }
```

```
}
```

```
void creaArreglo(char id[50], int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8)
```

```
{
```

```
    while (aux < n)
```

```
    {
```

```
        if (buffer[aux].valor[0] == NULL)
```

```
        {
```

```
            strcpy(buffer[aux].nombre, id);
```

```
            buffer[aux].valor[0] = x1;
```

```
            buffer[aux].valor[1] = x2;
```

```
            buffer[aux].valor[2] = x3;
```

```
            buffer[aux].valor[3] = x4;
```

```
            buffer[aux].valor[4] = x5;
```

```
            buffer[aux].valor[5] = x6;
```

```
            buffer[aux].valor[6] = x7;
```

```
            buffer[aux].valor[7] = x8;
```

```
            printf("Se ha agregado %s \n", buffer[aux].nombre);
```

```
            break;
```

```
        }
```

```
    else
```

```

    {
        aux = aux + 1;
    }
}
}

```

```

int buscaPosicionArreglo(char id[50])
{
    int p = 0;
    while (strcmp(buffer[p].nombre, id) != 0)
    {
        p++;
    }
    return p;
    //controlar cuando salga del array
}

```

```

void meterEnArreglo(char id[50], int x, int y)
{
    int pos = buscaPosicionArreglo(id);

    for(int i = 6; i >= y-1; i--){
        buffer[pos].valor[i+1] = buffer[pos].valor[i];
    }
    buffer[pos].valor[y-1] = x;
    printf("Se ha metido con exito\n");
}

```

```

void mirarArreglo(char id[50])
{
    int pos = buscaPosicionArreglo(id);
    //printf("el arreglo %s contiene los siguientes elementos: ", buffer[pos].nombre);
    for (int i = 0; i < 8; i++)
    {
        if (buffer[pos].valor[i] != 0) printf("%i ", buffer[pos].valor[i]);
    }
    printf("\n");
}

```

```

void sacarDeArreglo(char id[50],int y)
{
    int pos = buscaPosicionArreglo(id);

    buffer[pos].valor[y-1] = buffer[pos].valor[y];

    int aux = 0;
    for(int i = y; i<7 ;i++) buffer[pos].valor[i] = buffer[pos].valor[i+1];

    buffer[pos].valor[7] = 0;
    printf("Se ha sacado con exito \n");

}

```

```

void datoDeArreglo(char id[50],int x)
{
    int pos = buscaPosicionArreglo(id);
    printf("%i ", buffer[pos].valor[x-1]);

}

```

```

int main(int argc,char **argv) //Programa Principal
{
    yyparse(); //funcion propio de bison que ejecuta el analizador sintactico
    printf("La ejecucion termino de manera correcta ");
    return 0;

}

```