Facultad de Ciencias de la Ingeniería UACh

Facultad de Ciencias Económicas y Administrativas

TRADUCTOR DIRIGIDO POR SINTAXIS LENGUAJE ETT

Asignatura: Compiladores INFO 165
Profesora: Maria Eliana de la Maza Werner
Integrantes: Felipe Saldias
Henry Fehrmann

Fecha: 22/11/2018

INTRODUCCIÓN

Este trabajo consistirá en desarrollar un traductor dirigido por sintaxis que permite ejecutar programas escritos en lenguaje ETT. El traductor deberá aceptar una secuencia de instrucciones del lenguaje ETT y efectuar las acciones a medida que se realiza el análisis sintáctico en tiempo real. Para esto se debe realizar etapas de analisis lexico y sintactico del programa a realizar, junto con una traducción dirigida por sintaxis. Utilizando las herramientas Flex y Bison (Lenguaje de programación C).
Este traductor será autoexplicativo con tal de que no sea necesidad de un manual de usuario, las instrucciones se encontrarán disponibles al momento de ejecutar el programa.

Las instrucciones que posee el lenguaje son:

EDITAR: La instrucción que da inicio al programa.
COLOR: Fija el color de los trazos a dibujar, estos colores pueden ser rojo, verde, azul, amarillo o blanco.
POS(x,y): Fija el cursor en las coordenadas x, y para el próximo trazado.
DER(x): Dibuja x trazos hacia la derecha.
IZQ(x): Dibuja x trazos hacia la izquierda.
ARR(x): Dibuja x trazos hacia arriba.
ABA(x): Dibuja x trazos hacia abajo.
DAVALOR id = dato: asigna al identificador id, el dato ingresado, dónde dato es un d entero o uno de los cinco colores.
TERMINO: Última instrucción del programa.

Cabe destacar que la entrada se realizará mediante la consola de comandos de la terminal mientras que la salida será a medida que se ingresan las instrucciones en una ventana aparte

DESARROLLO

En primer lugar se analizarán las herramientas Flex y Bison a utilizar durante la construcción del traductor. Veremos cómo funciona cada una de ellas, los elementos necesarios de entrada y de qué manera se integran para lograr trabajar en conjunto y facilitar la construcción de herramientas de traducción.

FLEX

“Flex es un 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.”
El fichero Flex, de extensión .l, generalmente sigue la siguiente sintaxis para su definición:

Definiciones
%%
Reglas
patrón1 {acción1}
patrón2 {acción2} …
%%
Código de usuario

este fichero contiene 3 secciones principales descritas a continuación:

- **Definiciones:** esta sección contiene declaraciones de nombres simples para referirse de manera más sencilla a expresiones regulares en la siguiente sección de especificación del escáner. Por ejemplo: si se escribiera una línea de la forma “DIGITO [0-9]”, en esta parte del programa, cuando queramos referirnos a la expresión regular que representa un número del 0 al 9 solo deberemos escribir DIGITO y el analizador comprenderá a lo que nos referimos.
- **Reglas**: esta sección define líneas de la forma **patrón {acción}** en donde los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares y usando como alfabeto cualquier carácter **ASCII** (“ \ [^ - ? . * + | ( ) $ /) y la acción corresponde a un trozo de código en C a ejecutar al reconocer dicho patrón.
- **Código de usuario**: código escrito en lenguaje C que se copiará directamente en el fichero de salida generado.

Flex genera como salida un fichero fuente en C, **‘lex.yy.c’**, que define

una función **‘yylex()’**. Este fichero se compila y se enlaza con la librería de Flex para producir un ejecutable que analizará la entrada en busca de coincidencias y al encontrarlas ejecutara su respectivo código en C.

BISON

“Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática.”
El fichero Bison de extensión .y, generalmente sigue la siguiente sintaxis para su definición:

%{
Declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional

Los símbolos de ‘%%’, ‘%{‘ y ‘%}’ son parte de la sintaxis e indican cuándo termina una sección y comienza la siguiente. Para las secciones tenemos:

- **Declaraciones en C:** esta sección define variables globales las cuales serán visibles para ser utilizadas en las acciones a ejecutar para las reglas gramaticales.
- **Declaraciones de Bison**: esta sección se utiliza para definir tipos de variables a utilizar tanto en terminales (tokens) y variables de nuestra gramática, es decir el tipo de dato que tendrán asociadas estas en caso de ser utilizadas para operar en la sección de acciones a ejecutar.
- **Reglas gramaticales**: en esta sección van directamente las producciones de la gramáticas las que pueden llevar acciones asociadas las cuales se ejecutarán cuando el analizador calce con la regla descrita por la producción.
- **Código de usuario**: código escrito en lenguaje C que se copiará directamente en el fichero de salida generado, en este caso utilizado para definir el método principal (main) del programa.

El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática.

COMBINANDO FLEX Y BISON

Para la construcción del traductor solicitado en esta oportunidad es necesario utilizar en conjunto las herramientas descritas en la sección anterior. Básicamente el archivo creado por Bison contiene una función llamada yyparse() la cual, en sí, es nuestro analizador y se comenzará a ejecutar, simplemente al llamar dicha función desde el método principal de nuestro programa (que como se mencionó con anterioridad se encuentra definido en la sección de código de usuario del fichero bison). El analizador sintáctico será el encargado de llamar internamente a la función yylex() cada vez que necesite un token para seguir adelante con su ejecución.

LA GRAMÁTICA A UTILIZAR

A continuación encontrará la gramática utilizada para la construcción del traductor solicitado, si desea ver la manera en que esta se debe ingresar en el archivo de Bison, diríjase al Anexo 2 que encontrará al final de este informe.

S:	**editar** inst **termino**	(1)
inst :	inst inst	(2)
	| inst_color	(3)
	| inst_pos	(4)
	| inst_asig	(5)
	| inst_asig2	(6)
	| inst_dir	(7)
inst_color:	color(col)	(8)
Inst_pos:	pos(exp,exp)	(9)
Inst_asig:	davalor id = dato	(10)
Inst_asig2:	davalor id = colores	(11)
Inst_dir:	der(exp)	(12)
	|arr(exp)	(13)
	|ab(exp)	(14)
	|izq(exp)	(15)
colores:	rojo	(16)
	| verde	(17)
	| azul	(18)
	| amarillo	(19)
	| blanco	(20)
dato :	const	(21)
exp:	id	(22)
	| const	(23)
col:	colores	(24)
	| id	(25)

Donde id corresponde a una letra seguida de cualquier combinación de letras y dígitos

y const a un tipo integer definidas en Flex como [a-Z][a-Z0-9]* y [0-9]+ respectivamente.

Graphics.h

Es necesario definir esta librería antes de explicar las funciones y métodos utilizados en el programa puesto que justamente de aquí provienen varios de los métodos usados. La inclusión de esta librería es el resultado de una búsqueda que inició con la pregunta “¿cómo podemos dibujar en consola tal y como lo solicita el trabajo?”. En primera instancia aparecieron por nuestra cabeza ideas del tipo: “podemos hacer un arreglo bidimensional e ir dibujando en sus casilleros paréntesis de diferentes colores dependiendo de lo que solicite el usuario” idea que fue descartada rápidamente luego de darle un par de vueltas y un rapido analisis en grupo. Algo no estábamos viendo, mas bien, no sabiamos que existia. Luego de un par de búsquedas por internet, por fin dimos con ella, una librería que hacía justamente todo lo que necesitábamos. Graphics.h permitió crear una ventana de 640 x 480 pixeles y dibujar como si fuera un lápiz puesto en coordenadas arbitrarias definidas por el usuario y que incluye, desde; funciones para trazar líneas desde el punto actual a otro también definido por el usuario, posicionar el “lápiz” en coordenadas a voluntad e incluso fijar el color de lo que se dibuje desde ese momento. A partir de ahora, para referirnos a las funciones provenientes de esta librería, se antepondra el prefijo graph:: a la función que necesitemos referenciar.

ESTRUCTURAS DE DATOS UTILIZADAS

Fue necesario crear una estructura de datos para almacenar las variables y sus valores numéricos provenientes de la instrucción de asignación, en este caso por la simplicidad y el uso que se le dará al programa se decidió utilizar un arreglo de registros para este propósito. Definiendo un registro de la siguiente manera:

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

los cuales se encontrarán dentro de un arreglo del estilo:

```
struct id buffer[1000];
```

FUNCIONES Y MÉTODOS

A continuación se realizará una pequeña explicación de cada uno de los métodos utilizados para la construcción de nuestro programa. Posteriormente, en la siguiente sección, veremos de qué manera y en qué momento estos métodos y funciones fueron utilizados.

- **void graph::moveto(int x,int y):** esta función posiciona el “lápiz” en las coordenadas x e y.
- **void graph::setcolor(int x):** establece el color en el valor de x (los colores tienen un número específico para referirse a ellos) para los próximos dibujos hasta que el color sea cambiado nuevamente.
- **int graph::getx():** retorna la coordenada x de la posición del “lápiz”.
- **int graph::gety():** retorna la coordenada x de la posición del “lápiz”.
- **int graph::lineto(int x,int y):** dibuja una línea desde la posición actual del “lápiz”, hasta las coordenadas x ,y.
- **void separar(char palabra[50]):** recibe como entrada una cadena de texto que en este caso en particular será siempre del estilo “variable=10” almacena en una variable llamada parte1[50] el string ”variable” y en otra llamada parte2[50] el string “10”.
- **void agregarID(int pos, char id[50], int num):** Almacena en la posición pos del arreglo buffer un nombre que toma del parámetro de entrada id[50] y un valor equivalente al parámetro de entrada núm.
- **int buscarID(char id[50]):** Recorre el arreglo buffer hasta encontrar la posición que tiene almacenado en su campo de nombre el id[50], luego retorna el valor que tiene asociada esa variable en el registro.

FUNCIONAMIENTO

Una vez teniendo la piezas solo queda definir de qué manera se utilizaron todas ellas para dar solución al problema presentado. Básicamente la mayor pega se la llevan los programas bison y flex ya que a partir de las definiciones que revisamos más arriba, generan los programas que nos permiten analizar palabras y determinar si estas pertenecen al lenguaje que nuestra gramática define. Teniendo en cuenta esto el primer paso fue definir de manera correcta las expresiones regulares y gramática, en los archivos ".l" y ".y". Una vez comprobado que los programas estuvieran realizando el paso de información de manera correcta entre Flex y Bison, es decir, que Flex además de avisar que encontró una determinada expresión regular, pudiera enviar este valor a Bison en el caso de los identificadores y constantes numéricas a través de la sentencia yylval.texto=yytext la cual deja el "valor semántico" de la entrada en una variable a la que Bison puede acceder simplemente referenciando la posición del terminal dentro de la producción.

Una vez la primera parte se encontró funcionando de manera deseada se implementaron las acciones asociadas a las producciones las cuales se describirán a continuación haciendo referencia mediante los números entregados anteriormente a las producciones en la sección "GRAMÁTICA A UTILIZAR":

- **Producción (8)**: Esta producción hace uso de la función **graph::setcolor(int x)** entregando el valor que proviene de la variable col que a su vez puede provenir de un identificador almacenado en la tabla de símbolos o colores que es ingresado directamente como parámetro al escribir en la consola.
- **Producción (9):** Esta producción hace uso de la función **graph::moveto(int x,int y)** y los parámetros ingresados pueden provenir de un identificador o de una constante
- **Producción (10):** Esta producción hace uso de la función **separar(char palabra[50]) y agregarID(int pos, char id[50], int num)** para almacenar variables definidas como una constante numérica**.** En un principio se pretendía implementar las producciones (10) y (11) como una sola, pero surgieron algunos inconvenientes que se comentarán en la próxima sección de este informe.
- **Producción (11):** Esta producción hace uso de la función **separar(char palabra[50]) y agregarID(int pos, char id[50], int num)** para almacenar variables definidas como un color.
- **Produciones (12),(13),(14),(15):** Todas estas producciones tienen en común que utilizan la función **int graph::lineto(int x,int y)** pero se le pasan los parámetros en diferente orden dependiendo de la dirección hacia la cual se requiera dibujar la línea.

- **<u>Produciones (22),(25):</u>** Ambas producciones hacen uso de la función **buscarID(char id[50])** y asignan el valor retornado a la variable que hizo referencias a ellas

EL PROBLEMA DE LA INSTRUCCIÓN DE ASIGNACIÓN

Como se mencionó anteriormente, en un principio, la gramática original consideraba las producciones (10) y (11) como una sola, de hecho el código fue escrito de esa manera en primera instancia, pero al momento de llegar a la implementación de esta instrucción (la dejamos para el final) nos encontramos con que el token IDENTIFICADOR estaba almacenando toda la entrada del teclado después de la palabra da valor, ejemplo: al ingresar la instrucción davalor d=10, el token IDENTIFICADOR almacenaba “d=10”. primeramente solucionamos esto diseñando el método separar que nos permitió dividir en dos cadenas de caracteres distintas llamadas parte1 y parte2 las cuales almacenan “d” y “10” respectivamente después de invocar el método. Pensamos que el problema estaba solucionado con esto pero luego nos percatamos que esa instrucción sólo iba a servir para el caso que la instrucción de asignación se utilizará solo para asignar un valor de constante entera a la variable por lo que la solución final para este problema fue dividir las producciones para asignación, utilizando una exclusivamente para asignar valores enteros y la otra para asignar colores a variables.

ESPECIFICACIÓN DEL PROGRAMA

Nuestros programas son dos, lexico.l el cual al ser compilado por flex este da como resultado un archivo llamado lexico.yy.c el que transforma este archivo flex en formato c. El segundo programa es sintactico.y el que al ser compilado entrega como salida sintactico.tab.c, que al igual que el anterior es el archivo bison en formato c. A partir de estos dos archivos podemos hacer el ejecutable, el cual llamamos a analizar, su función es dar vida a nuestra gramática y así crear un analizador de lenguaje con ella.

REQUERIMIENTOS DE HARDWARE Y SOFTWARE

El requerimiento de software es el siguiente:
Ubuntu 16.04 como sistema operativo.
Compilador Flex.
Compilador Bison.
Graphics.h

Hardware mínimo para ejecución

- Procesador de 1 GHz (por ejemplo Intel Celeron) o mejor.
- 1.5 GB RAM (Memoria del sistema, el cual es el caso de ubuntu).
- Acceso a internet (para instalar actualizaciones durante el el proceso de instalación tanto para los paquetes de ubuntu por defecto y los paquetes de flex, bison y graphics.h).

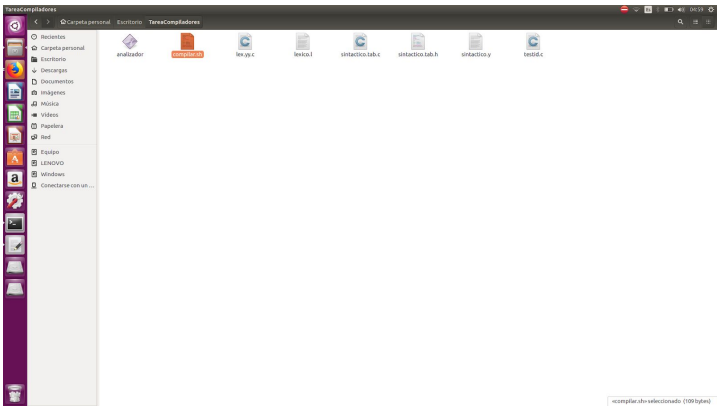
COMPILACIÓN Y EJECUCIÓN

Compilación: hecha en el terminal de ubuntu donde esta la carpeta de nuestro trabajo:
bison -d sintactico.y
flex lexico.l
cc lex.yy.c sintactico.tab.c -o analizador -lfl -lm -lgraph
nosotros utilizamos un archivo para ahorrar tiempo en compilar llamado compilar.sh, el cual introducimos en la consola como “sh compilar.sh” y este ejecuta la compilacion antes descrita.

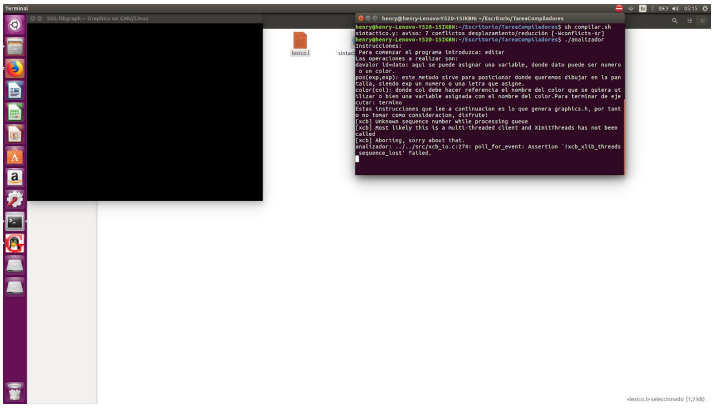
Ejecución: Desde el terminal escribir ./analizador y el programa es ejecutado.

EJEMPLO DE FUNCIONAMIENTO

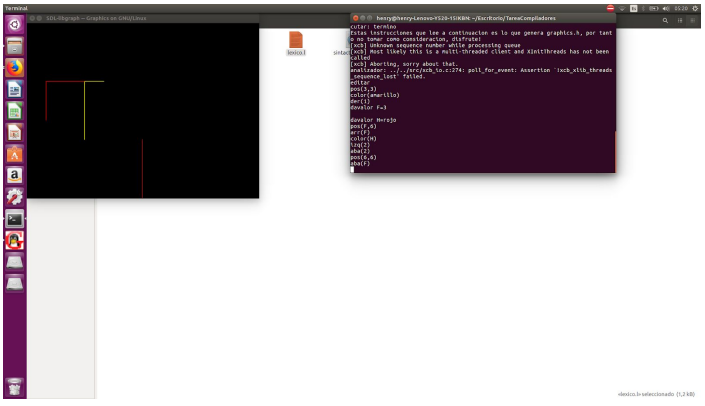
Entramos a la carpeta que está nuestro trabajo



Compilamos con sh compilar.sh, luego ejecutamos ./analizador y nos muestra las instrucciones de como usarlo, y además el programa está listo para utilizar, abriéndose un cuadro negro al lado el cual irá dibujando a medida que se vaya escribiendo en consola a tiempo real.



Aquí se ejecutaron algunas instrucciones del lenguaje para visualizar el funcionamiento del programa, como es el caso de pos, davalor, color, etc.



CONCLUSIONES

En grupo vimos mucho potencial en flex y bison, ya que aprendimos la parte práctica del curso de COMPILADORES. Ocupamos todos los conocimientos de cursos pasados, como es el caso de ESTRUCTURA DE DATOS para usar un arreglo de registros, AUTÓMATAS para definir expresiones regulares con los tokens de Flex y así trabajarlos con Bison y una lógica que nos ayude al desarrollo del programa.
Hay varias mejoras que se le puede hacer al programa. Una de ellas es haber trabajado mejor el tema de davalor identificador=dato ya que al darnos ese error no hayamos la manera de ejecutar bien la producción y que dato pudiese expandir tanto constante como color. La otra es la estructura de datos que utilizamos, ya es la más ineficiente (última clase de Compiladores), teniendo en cuenta esto se podría hacer una lista enlazada, una estructura tipo árbol o un hash.
Las limitaciones que posee nuestro trabajo es el tamaño del arreglo, ya que si se pasa de los 50 caracteres no estaría guardando el identificador de la mejor manera, inclusive hay memoria que se pierde si es que se utiliza pocos datos, teniendo en cuenta que tenemos un buffer de 1000 elementos.

Bibliografía

Introducción a Flex y Bison
http://webdiis.unizar.es/asignaturas/LGA/material_2003_2004/Intro_Flex_Bison.pdf
Instalar libreria graphics.h en ubuntu
https://www.geeksforgeeks.org/add-graphics-h-c-library-gcc-compiler-linux/
Funciones de la librería graphics
https://www.programmingsimplified.com/c/graphics.h

ANEXO 1 lexico.l

```
%{
/* se necesita esto para la llamada a atof() más abajo */
#include <math.h>
#include "sintactico.tab.h"     /*incluye un archivo generado por bison para permitir la comunicacion de los tokens
                                                        entre flex y bison*/

%}
/*Declaracion de expresiones regulares a utilizar*/
DIGITO   [0-9]
ID       [A-z][A-z0-9]*

/*Declaracion de tokens mediante expresiones regulares*/
%%

{DIGITO}+   {yylval.numero=atof(yytext); return(CONST);}

"("         {return(PARA);}
")"         {return(PARC);}
"editar"   { return(EDITAR); printf( "Se enconetro la palabra clave being: %s con un largo de %i\n", yytext ,(int)(yyleng) );}
"termino"     {return (TERMINA);}
"color"             {return (COLOR);}
"pos"     {return (POS);}
"davalor"    {return (DAVALOR);}
"der"     {return (DER);}
"arr"     {return (ARR);}
"aba"     {return (ABA);}
"izq"     {return (IZQ);}
"rojo"     {return(ROJO);}
"verde"             {return (VERDE);}
"azul"    {return (AZUL);}
"amarillo"    {return (AMARILLO);}
"blanco"            {return (BLANCO);}
","         {return(COMA);}
"="         { yylval.texto=yytext;return(IGUAL);}

{ID}       {yylval.texto=yytext;return(IDENTIFICADOR);}
"{"[^}\n]*"}"         /* se come una linea de comentarios */

[ \t\n]+              /* se come los espacios en blanco */
```

ANEXO 2 sintactico.y

```
%{

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

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


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

 //Declaracion de metodos
  void yyerror(char *s);
  void agregarID(int pos, char id[50], int num);
  void separar(char palabra[50]);


//Declaracion del nodo del buffer de identificadores
  struct id {
            char nombre[50];
            int valor;
};

//Declaracion de variables globales
  int buscarID(char id[50]);
  struct id buffer[1000];
  int pos=0;
  char string[50];
  char parte1[50];
  char parte2[50];




%}

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

```
/*Declaración tipo de dato a utilizar en las terminales y variables
  de la gramatica, en este caso entero y string*/
%union
{
  int numero;
  char* texto;
}

/*Declaración de tokens*/
%token <numero> CONST
%token PARA
%token PARC
%token COMA
%token <texto>IGUAL
%token EDITAR
%token TERMINA
%token COLOR
%token POS
%token DAVALOR
%token DER
%token ARR
%token ABA
%token IZQ
%token <texto>ROJO
%token <texto>VERDE
%token <texto>AZUL
%token <texto>AMARILLO
%token <texto>BLANCO
%token <texto> IDENTIFICADOR



/*Declaracion de variables que tendrán valor asociado*/
%type <numero> dato
%type <numero> col
%type <numero> exp
%type <numero> colores




%%
/************************
 * Reglas Gramaticales *
 ************************/

/*Inicio de la gramatica*/
s :                     EDITAR inst TERMINA
                        ;

inst:       inst inst
```

```
                    |inst_color
                    |inst_pos
                    |inst_asig
                    |inst_asig2
                    |inst_dir
                    ;

inst_color: COLOR PARA col PARC {setcolor($3);}
                    ;

inst_pos:   POS PARA exp COMA exp PARC {moveto(53*$3,53*$5);}
                    ;

inst_asig:                          DAVALOR IDENTIFICADOR IGUAL dato
{separar($2);agregarID(pos,parte1,atoi(parte2)); pos++;}
                    ;
inst_asig2:      DAVALOR IDENTIFICADOR IGUAL colores {separar($2);agregarID(pos,parte1,$4);
pos++;}
                    ;

inst_dir:   DER PARA exp PARC    {lineto(53*$3+getx(),gety());}
                    |ARR PARA exp PARC   {lineto(getx(),gety()-53*$3);}
                    |ABA PARA exp PARC   {lineto(getx(),gety()+53*$3);}
                    |IZQ PARA exp PARC   {lineto(getx()-53*$3,gety());}
                    ;

colores:    ROJO          {$$=4;}
                    |VERDE          {$$=2;}
                    |AZUL           {$$=1;}
                    |AMARILLO   {$$=14;}
                    |BLANCO         {$$=15;}
                    ;

dato:     CONST {$$=$1; }

                    ;

exp:      IDENTIFICADOR {char pala[50]; strcpy(pala,$1); int i=buscarID(pala);$$=i; }
                    |CONST {$$=$1;}
                    ;

col:      colores
                    |IDENTIFICADOR {$$=buscarID($1);}
                    ;
%%
/**********************
 * Codigo C Adicional *
 **********************/
void yyerror(char *s)
{
   printf("Error sintactico %s \n",s);
```

```
}


void agregarID(int pos, char id[50], int num){ //funcion para agregar un identificador y su respectivo valor en el arreglo buffer en la asignacion

   strcpy(buffer[pos].nombre, id);
         buffer[pos].valor=num;
}
int buscarID(char id[50]){  //esta funcion se encarga de retornar el valor que tiene almacenado un id en la tabla de buffer cuando es referenciado

   int p=0;
   while(strcmp(buffer[p].nombre, id) != 0){
         p++;
   }
   return buffer[p].valor;
}
void separar(char palabra[50]){/*separa la entrada del token IDENTIFICADOR en el error explicado en el informe, para así separarlos en dos strings,
                                                                los cuales seran utilizados en la tabla buffer*/

     for(int k= 0; k<50;k++){
                  parte1[k]=0;
         parte2[k]=0;
   }
     int posi=0;
   while(palabra[posi] != '='){
                  posi++;
   }
   int j=0;
   for(int i =0;i<50;i++){
         if (i<posi) parte1[i]=palabra[i];
         else if (i==posi);
         else{
                  parte2[j]=palabra[i];
                  j++;
         }
    }
}
int main(int argc,char **argv) //Programa Principal
{

   printf("Instrucciones:\n ");
   printf("Para comenzar el programa introduzca: editar\n");
   printf("Las operaciones a realizar son: \n");
     printf("davalor id=dato: aquí se puede asignar una variable, donde dato puede ser número o un color.\n");
     printf("pos(exp,exp): este método sirve para posicionar donde queremos dibujar en la pantalla, siendo exp un número o una letra que asigne.\n");
```

```
    printf("color(col): dónde col debe hacer referencia el nombre del color que se quiera utilizar o bien una variable asignada con el nombre del color.\n");
    printf("Para terminar de ejecutar: termino \n");
    printf("Estas instrucciones que lee a continuación es lo que genera graphics.h, por tanto no tomar como consideración, disfrute! \n");


    int gd = DETECT,gm;
    initgraph (& gd,& gm,NULL); //inicialización de la ventana para dibujar

    yyparse();//funcion propio de bison que ejecuta el analizador sintáctico
    closegraph(); //cierra la ventana si el analisis sintactico termino de manera correcta
    printf("La ejecución termino de manera correcta ");
    return 0;
}
```