

Analizador Léxico y sintáctico

Álvarez González Ian

García Oviedo Jaasiel Osmar

López López Ulysses

Domínguez Cisneros Alexis Saul

Gpo 1.

Compiladores |
2020-1

Analizador léxico

Análisis del problema:

Se debe elaborar un programa capaz de interpretar las 27 gramáticas dejadas por el profesor, para esto debe crear un analizador Léxico.

Diseño de solución:

Para poder hacer este analizador decidimos usar el lenguaje lex o flex, que, precisamente se basa en el uso de expresiones regulares para la identificación ya sea de patrones o cadenas de lenguaje, lo cual nos ayudará con las cadenas que le queramos introducir. Ahora bien, a continuación, se mostrarán algunos de los diferentes componentes léxicos o tokens que deberá reconocer nuestro analizador léxico, así como sus respectivas expresiones regulares, las cuales hemos visto durante el transcurso del semestre:

Clase o id	Nombre	Expresión regular
0	Reservadas	(do for double char void struct false func printf)
1	digito	[0-9]
2	Reales	[Ee][+-]?[0-9]{1,2}
3	Exponentes	(({enteroNum}\.[0-9]*\.{enteroNum}){exponente}? {enteroNum}{exponente})
4	Letra	[a-zA-z]
4	Letra_	({letra} _)+
5	Id	({letra_} {letra_}{letra_} {letra_}{digito})+
6	Operador	[+ - * / %]
7	Cadena	["]({letra}* {digito}*)+["]
8	Cáacter	'.'
9	OpeEsp	[(){}]
10	Condicional	[&& !]
11	Comentario	[/][*]({letra} {entero})*[n]({letra} {entero})*[*][/]
12	Relacional	[< > <= >= != == =]
13	Espacio	[\n\t] +

A continuación, mostraremos las gramáticas que nos dieron:

Gramatica

sin: significa sin tipo, car: tipo carácter

Gramaticas	
1. programa → declaraciones funciones	2. declaraciones → tipo lista_var; declaraciones tipo registro lista_var; declaraciones ε
3. tipo_registro → estructura inicio declaraciones fin	4. tipo → base tipo_arreglo
5. base → ent real dreal car sin	6. tipo_arreglo → [num] tipo arreglo ε
7. lista var → lista var, id id	8. funciones → def tipo id(argumentos) inicio declaraciones sentencias fin funciones ε
9. argumentos → listar_arg sin	10. lista_arg → lista_arg, arg arg
11. arg → tipo_arg id	12. tipo_arg → base param_arr
13. param_arr → [] param_arr ε	14. sentencias → sentencias sentencia sentencia
15. sentencia → si e_bool entonces sentencia fin si e_bool entonces sentencia sino sentencia fin mientras e_bool hacer sentencia fin hacer sentencia mientras e_bool; según (variable) hacer casos predeterminado fin variable := expresion ; escribir expresion ; leer variable ; devolver; devolver expresion; terminar; inicio sentencias fin	16. casos → caso num: sentencia casos caso num: sentencia
17. predeterminado → pred: sentencia ε	18. e_bool → e_bool o e_bool e_bool y e_bool no e_bool (e_bool) relacional verdadero falso
19. relacional → relacional oprel relacional expresion	20. oprel → > < >= <= <> =
21. expresion → expresion oparit expresión expresion % expresion (expresion) id variable num cadena caracter id(parametros)	22. oparit → + - * /
23. variable → dato est_sim arreglo	24. dato est_sim → dato est_sim .id id
25. arreglo → id[expresion] arreglo[expresion]	26. parametros → lista_param ε
27. lista_param → lista_param, expresion expresion	

Implementación:

```
/*
 * Autores: Dominguez Cisneros Alexis Saul y Garcia Oviedo Jaasiel Osmar
 * Creado 22/05/2020 by Dominguez Cisneros Alexis Saul
 * Editado y terminado: 23/05/2020 by Garcia Oviedo Jaasiel Osmar
 *ultima revision: 24/05/2020 by Dominguez Cisneros Alexis Saul y Garcia Oviedo Jaasiel Osmar
 *ESTE PROGRAMA LLEVA A CABO LA GENERACION DEL ANALIZADOR LEXICO
 */
%{
    #include <stdio.h>
    #include <stdlib.h>
    int num_char=0;
    int num_linea=0;
    char linea[20];
    char cadena[20];
    FILE *e;
    int token;
}%
%option noyywrap
%option yylineno
/*
*****EXPRESIONES REGULARES*****
*/
letra [a-zA-Z]
digito [0-9]
letra_ ({letra}|_)+
id ({letra_}|{letra_}({letra_}|{digito}))+
tipo $(int|float|double|char|void|struct)
res #(do|for|double|char|void|struct|false|func|printf)
opesp [(|)|{|}]
termina [;]
espacio [ \n\t]
cadena [""]({letra}*|{digito})*+[""]
```

```

numero ({digito}*.{digito})+|({digito})+
coment [/][*]({letra}|{digito})*[\n]({letra}|{digito})*[*][/]
opera [+|-|*|/|%]
condi [| | && | ! ]
rela [<|>|<=|>=|!=|==|=]
%%
/*ACCIONES LEXICAS*/
/* Se especifican las acciones que se llevaran a cabo cuando se identifiquen cada uno de los
tokens acetados por la gramática propuesta. Igualmente se retorna la clase lexica, la cual
recibira el analizador sintactico para llevar a cabo su tarea*/
{id} {num_char=num_char+yylen;
      fputs("\n<id\t,t",yyout);
      fputs(yytext,yyout);
      fputs(">",yyout);
      yylex();
      printf("\n<(1)id,%s>\n",yytext);
    }
/*Para los casos: res, opesp, opera, condi y rela se retorna una clase lexica por cada uno de los elementos
que los conforman y asi estos puedan ser diferenciados e identificados de forma individual por el
analizador sintactico.*/
{tipo} {num_char=num_char+yylen;
        fputs("\n<tipo\t,t",yyout);
        fputs(yytext,yyout);
        fputs(">",yyout);
        yylex();
        printf("\n<(2)tipo,%s>\n",yytext);
      }
{res} {num_char=num_char+yylen;
       fputs("\n<res\t,t",yyout);
       fputs(yytext,yyout);
       fputs(">",yyout);
       yylex();

```

```

        printf("\n<(3)res,%s>\n",yytext);
    }
{opesp} {num_char=num_char+yyldeng;
        fputs("\n<opesp\t,\t",yyout);
        fputs(yytext,yyout);
        fputs(">",yyout);
        yylex();
        printf("\n<(4)opesp,%s>\n",yytext);
    }
/*los espacios los ignorados*/
{espacio} {num_char=num_char+yyldeng;}

{cadena} {num_char=num_char+yyldeng;
        fputs("\n<cadena\t,\t",yyout);
        fputs(yytext,yyout);
        fputs(">",yyout);
        yylex();
        printf("\n<(6)cadena,%s>\n",yytext);
    }
{numero} {num_char=num_char+yyldeng;
        fputs("\n<numero\t,\t",yyout);
        fputs(yytext,yyout);
        fputs(">",yyout);
        yylex();
        printf("\n<(7)numero,%s>\n",yytext);
    }
{opera} {num_char=num_char+yyldeng;
        fputs("\n<opera\t,\t",yyout);
        fputs(yytext,yyout);
        fputs(">",yyout);
        yylex();
        printf("\n<(8)opera,%s>\n",yytext);
    }

```

```

{coment} {num_char=num_char+yyleng;
    fputs("\n<coment\t,\t",yyout);
    fputs(yytext,yyout);
    fputs(">",yyout);
    yylex();
    printf("\n<(9)coment,%s>\n",yytext);
}
{condi} {num_char=num_char+yyleng;
    fputs("\n<condicional\t,\t",yyout);
    fputs(yytext,yyout);
    fputs(">",yyout);
    yylex();
    printf("\n<(10)condi,%s>\n",yytext);
}
{rela} {num_char=num_char+yyleng;
    fputs("\n<relacional\t,\t",yyout);
    fputs(yytext,yyout);
    fputs(">",yyout);
    yylex();
    printf("\n<(11)rela,%s>\n",yytext);
}
{termina} {num_char=num_char+yyleng;
    fputs("\n<termina\t,\t",yyout);
    fputs(yytext,yyout);
    fputs(">",yyout);
    printf("\n<(12)End_Linea,%s>\n",yytext);
}
[ \n\t]+ {}
/*Si se encuentra algun error, son agregados a un archivo en el    cual se muestra tanto la linea
    como la columna donde se produjo el error lexico.*/
. {printf("Error lexico: %s\n", yytext);
    num_char=num_char+yyleng;

```

```

    num_linea=yylineno;
    sprintf(linea, "%i", num_linea);
    sprintf(cadena, "%i", num_char);
    fputs("\n<error\t,\t", e);
    fputs(linea,e);
    fputs(":", e);
    fputs(cadena,e);
}

```

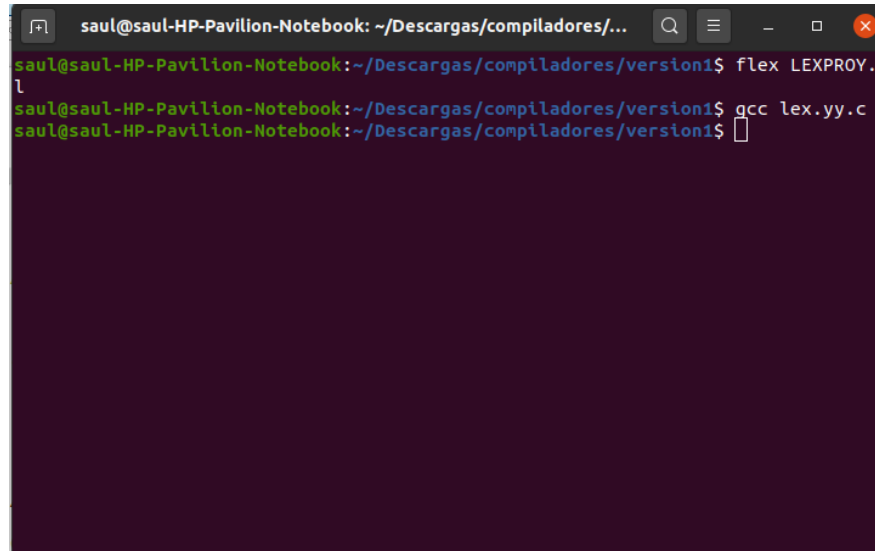
```

%%
%%

```

Compilar

1. Para poder ejecutar el programa tenemos que ir a la consola de nuestro equipo.
2. Tenemos que ingresar al fichero donde se encuentra nuestro archivo.
3. Después ingresamos el siguiente comando: *flex "nombre.l"*, si no genera ningún error podemos proceder con el siguiente paso.
4. Ejecutamos el siguiente comando: *gcc lex.yy.c*
5. Se muestra el resultado.

A terminal window with a dark purple background. The title bar shows 'saul@saul-HP-Pavilion-Notebook: ~/Descargas/compiladores/...'. The terminal shows the following commands and their outputs:

```
saul@saul-HP-Pavilion-Notebook:~/Descargas/compiladores/version1$ flex LEXPROY.l
saul@saul-HP-Pavilion-Notebook:~/Descargas/compiladores/version1$ gcc lex.yy.c
saul@saul-HP-Pavilion-Notebook:~/Descargas/compiladores/version1$
```

Compilo correctamente

Código main.c

```
/******Main C******/

/*En esta parte se lleva a cabo el procedimiento para abrir/crear y cerrar los archivos utilizados por el analizador
lexico. En total se hace uso de dos archivos: Tokens y Errores.*/

int main(int argc, char **argv){
    FILE *f, *t;
    f=fopen(argv[1], "r");
    yyin=f;
    t=fopen("Tokens.txt", "w");
    e=fopen("Errores.txt", "w");
    yyout=t;
    if(yyin==NULL||yyout==NULL){
        printf("Error\n");
    }else{
        yylex();
    }
    fclose(yyin);
    fclose(yyout);
    fclose(e);
}
```


Análisis Sintáctico

Análisis del problema:

El problema principal es poder obtener una correcta interpretación de una serie de gramáticas propuestas por el profesor, esto forma parte de un conjunto de problemas, los cuales tienen como objetivo un proyecto final el cual es construir un compilador.

En el análisis sintáctico se deben tomar varias decisiones, la gramática en algunas partes presenta ambigüedad, sin embargo; esta se puede ignorar utilizando en bison la precedencia de operadores y dejar que el programa la resuelva de manera automática o eliminar la ambigüedad antes de transcribir la gramática en bison.

Otra consideración que se debe tener en cuenta para resolver la ambigüedad del if - else, es que se debe asignar una precedencia a else como si fuera el operador de mayor precedencia. Ahora bien, teniendo en cuenta las gramáticas que se nos dieron, hicimos sus producciones, así como sus respectivas reglas semánticas como se mostrarán a continuación:

Producción	Regla semántica
$P \rightarrow DF$	$F.tipo = D.tipo$
$D \rightarrow TL; D$	$L.TIPO = T.TIPO$ $L.DIM = T.DIM$
$D \rightarrow \epsilon$	$D.tipo = D.base$
$T \rightarrow int$	$T.TIPO = INT$ $T.DIM = 2$
$T \rightarrow float$	$T.TIPO = FLOAT$ $T.DIM = 8$
$T \rightarrow double$	$T.TIPO = DOUBLE$ $T.DIM = 16$
$T \rightarrow char$	$T.TIPO = CHAR$ $T.DIM = 1$
$T \rightarrow void$	$T.TIPO = VOID$ $T.DIM = 0$
$L \rightarrow L1, id A$	$L1.TIPO = L.TIPO$ IF SYMTAB.ADD(ID, L.TIPO, DIR, 'VAR', -) != -1 $DIR = DIR + TYPE.TAB.GETDIM(L.TIPO)$ ELSE ERROR $A.BASE = L1.TIPO$
$L \rightarrow id A$	$SYMTAB.ADD(ID, LEXVAL, L.TIPO)$ $A.BASE = L.TIPO$
$A \rightarrow [num] A1$	IF $NUM.TIPO = INT \parallel NUM.TIPO = FLOAT \parallel$ $NUM.TIPO = DOUBLE \parallel NUM.TIPO = CHAR \parallel$ $NUM.TIPO = VOID$ $A1.BASE = A.BASE$ $A.TIPO = TYPETAB.ADD("ARRAY", NUM.VAL, A1.TIPO)$ ELSE

	ERROR
$A \rightarrow \epsilon$	A.TIPO=A.BASE
$F \rightarrow \text{func } T \text{ id } (\text{ ARG }) \{ D S \} F$	
$F \rightarrow \epsilon$	F.TIPO=F.BASE
$\text{ARG} \rightarrow L_A$	L_A.SIG=NUEVAETQ() ARG.CODIGO=L_A.CODIGO ETQ(L_A.SIG)
$\text{ARG} \rightarrow \epsilon$	ARG.TIPO=ARG.BASE
$L_A \rightarrow L_A , T \text{ id } P_A$	L_A.TIPO=T.TIPO IF SYMTAB.ADD(ID,L_A.TIPO,DIR,'VAR',-) != -1 DIR=DIR+TYPE.TAB.GETDIM(L_A.TIPO) ELSE ERROR P_A.TIPO=T.TIPO
$L_A \rightarrow T \text{ id } P_A$	IF SYMTAB.ADD(ID,L_A.TIPO,DIR,'VAR',-) != -1 DIR=DIR+TYPE.TAB.GETDIM(L_A.TIPO) ELSE ERROR P_A.TIPO=T.TIPO
$P_A \rightarrow [] P_A$	P_A.TIPO=L_A.TIPO
$P_A \rightarrow \epsilon$	P_A.TIPO=P_A.BASE
$C \rightarrow C1 \parallel C2$	C1.VERDADERO=C.VERDADERO C1.FALSO=NUEVAETIQUETA() C2.VERDADERO=C.VERDADERO C2.FALSO=C.FALSO C.CODIGO=C1.CODIGO ETQ(C1.FALSO) C2.CODIGO
$C \rightarrow C1 \& \& C2$	C1.VERDADERO=NUEVAETIQUETA()C1.FALSO=C.FALSO C2.VERDADERO=C.VERDADERO C2.FALSO=C.FALSO C.CODIGO=C1.CODIGO ETQ(C1.VERDADERO) C2.CODIGO
$C \rightarrow !C1$	C1.VERDADERO=C.FALSO C1.FALSO=C.VERDADERO C.CODIGO=C1.CODIGO
$C \rightarrow (C1)$	C.CODIGO=C1.CODIGO
$C \rightarrow E1 \text{ R } E2$	C.CODIGO=IF E1.DIR R.VAL E2.DIR GOTO C.VERDAERO GOTO C.FALSO
$C \rightarrow \text{TRUE}$	C.DIR=RES.LEXVAL
$C \rightarrow \text{FALSE}$	C.DIR=RES.LEXVAL
$E \rightarrow E1 + E2$	E.VAL=E1.VAL+E2.VAL
$E \rightarrow E1 - E2$	E.VAL=E1.VAL-E2.VAL
$E \rightarrow E1 * E2$	E.VAL=E1.VAL*E2.VAL
$E \rightarrow E1 / E2$	E.VAL=E1.VAL/E2.VAL
$E \rightarrow E1 \% E2$	E.VAL=E1.VAL%E2.VAL
expresión :expresion MAS expresión	

<pre> { E } </pre>	
$E \rightarrow V_A$	$E.VAL = V_A.VAL$
$E \rightarrow cadena$	$E.DIR = cadena.lexval$
$E \rightarrow numero$	$E.DIR = numero.lexval$
$E \rightarrow caracter$	$E.DIR = caracter.lexval$
$E \rightarrow id(P_A)$	
$V_A \rightarrow id[E]$	
$V_A \rightarrow V_A[E]$	
$P_A \rightarrow \epsilon$	$P_A.TIPO = P_A.BASE$
$P_A \rightarrow L_P$	$P_A.CODIGO = L_P.CODIGO$
$L_P \rightarrow E$	$L_P.VAL = E.VAL$
$L_P \rightarrow L_P, E$	

De las anteriores reglas las que utilizaremos en el programa son:

1	$P \rightarrow DF$
2	$D \rightarrow T L; D \mid \epsilon$
3	$T \rightarrow int \mid float \mid double \mid char \mid void \mid struct \{D\}$
4	$L \rightarrow L, id A \mid id A$
5	$A \rightarrow [num] A \mid \epsilon$
6	$F \rightarrow func T id (ARG) \{ DS \} F \mid \epsilon$
7	$ARG \rightarrow L_A \mid \epsilon$
8	$L_A \rightarrow L_A, T id P_A \mid T id P_A$
9	$P_A \rightarrow [] P_A \mid \epsilon$
10	$S \rightarrow SS \mid if (C) S \mid if (C) S else S \mid while (C) S \mid do S while (C); \mid for (S ; C ; S) S \mid P_I = E ; \mid return E ; \mid return ; \mid \{ S \} \mid switch (E) \{ CAS PRED \} \mid break ; \mid print E ;$
11	$CAS \rightarrow case: num S PRED \mid \epsilon$
12	$PRED \rightarrow default: S \mid \epsilon$
13	$P_I \rightarrow id \mid V_A \mid id.id$
14	$V_A \rightarrow id [E] \mid V_A [E]$
15	$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \% E \mid V_A \mid cadena \mid num \mid carácter \mid id (PAR)$
16	$PAR \rightarrow \epsilon \mid L_P$
17	$L_P \rightarrow L_P, E \mid E$
18	$C \rightarrow C \mid C \mid C \&\& C \mid ! C \mid (C) \mid E R E \mid true \mid false$
19	$R \rightarrow < \mid > \mid >= \mid <= \mid != \mid ==$

Los siguientes árboles fueron generados para llevar a cabo el análisis sintáctico

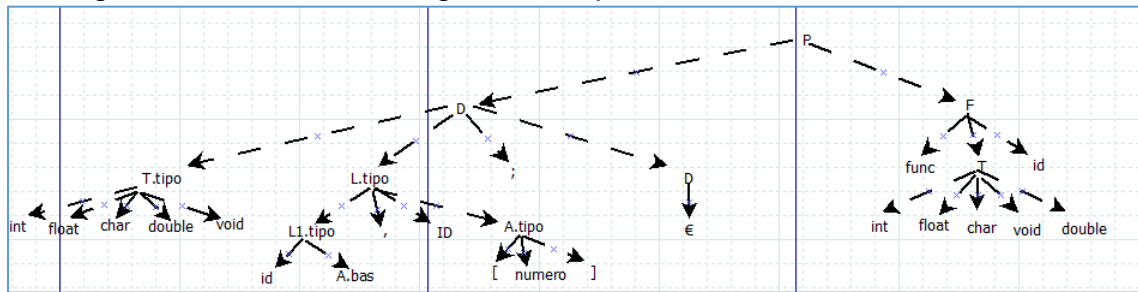


Ilustración 1: Árbol 1

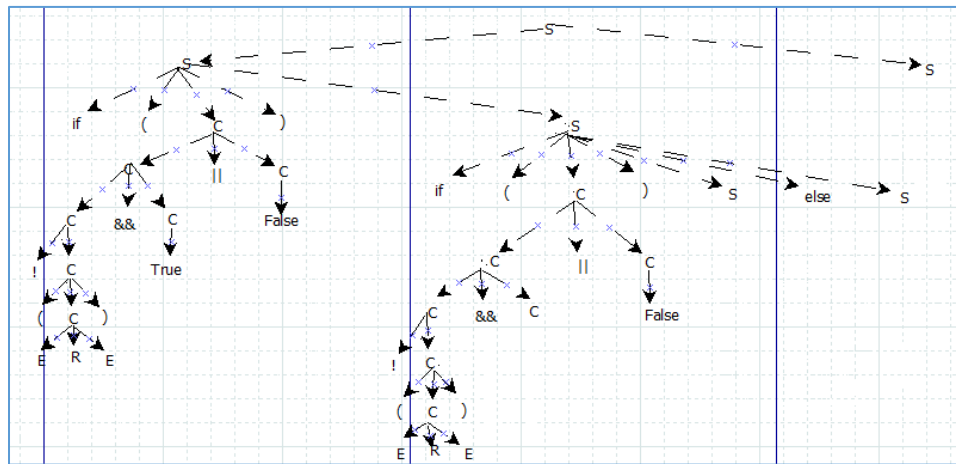


Ilustración 2: Árbol

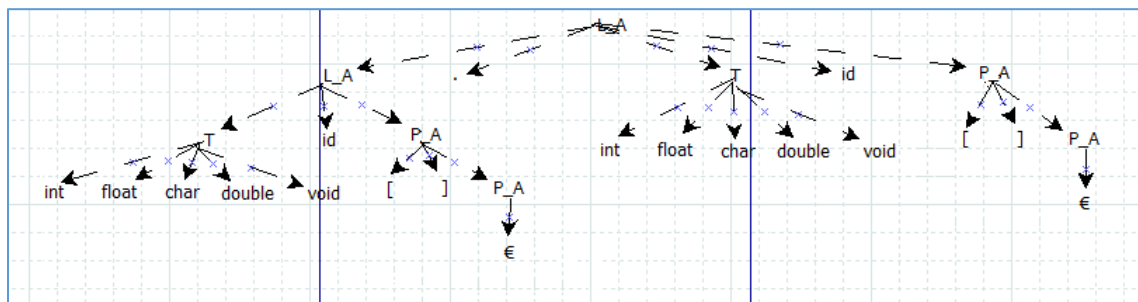


Ilustración 3:Árbol 3


```

%token ID
%token INT FLOAT DOUBLE CHAR VOID STRUCT
%token DO FOR FALSE FUN PRINTF TRUE IF ELSE WHILE SWITCH RETURN BREAK CASE DEFAULT
%nonassoc PARA PARC LLAA LLAC CORA CORC
%token CADENA
%token CARACTER
%token NUMERO
%right NEG
%left POR DIV MOD
%left MAS MEN
%token MENQUE MAYQUE MENIGUAL MAYIGUAL
%token IGUAL DIF
%left AND
%left OR
%right ASIGNA
%token COMENT
%token PYC
%token COMA DP PUNTO

%start prog

%%
/*-----DECLARACION DE LA GRAMATICA-----*/
/* A = Arreglo
   ARG = Argumento
   L_A = Lista de Argumentos
   P_A = Parte Arreglo
   P_I = Parte Izquierda
   CAS = Casos
   PRED= predeterminado
   V_A = var_arreglo
   PAR = Parámetros
   L_P = lista_parametros
   prog = programa
   decl = declaraciones
   array = arreglo
   arg = argumentos
   pred = predeterminado
*/

/* P → DF */
prog : decl funcion;

/* D → T L; | E */
decl : tipo lista PYC | ;

/* T → int | float | double | char | void | struct {D} */
tipo : INT | FLOAT | DOUBLE | CHAR | VOID | STRUCT LLAA decl LLAC;

/* L → L, id A | id A */
lista : lista COMA ID array | ID array;

/* A → [ num ] A | E */
array : CORA NUMERO CORC array | ;

/* F → func T id ( ARG ) { D } F | E */
funcion : FUN tipo ID PARA arg PARC LLAA decl sent LLAC funcion | ;

/* ARG → L_A | E */
arg : lista_arg | ;

/* L_A → L_A , T id P_A | T id P_A */
lista_arg : lista_arg COMA tipo ID parte_array | tipo ID parte_array;

/* P_A → CORA CORC P_A | E */
parte_array : CORA CORC parte_array | ;

/* S → SS | if ( C ) S | if ( C ) S else S | while ( C ) S | do S while ( C ); |
   for ( S ; C ; S ) S | P_I = E ; | return E ; | return; | { S } |
   switch ( E ) { CAS PRED } | break; | print E; */
sent : sent sent | IF PARA cond PARC sent | IF PARA cond PARC sent ELSE sent | WHILE PARA cond PARC sent |
DO sent WHILE PARA cond PARC PYC | FOR PARA sent PYC cond PYC sent PARC sent | parte_izq ASIGNA exp PYC |
RETURN exp PYC | RETURN PYC | LLAA sent LLAC | SWITCH PARA exp PARC LLAA caso pred LLAC | BREAK PYC | PRINTF exp PYC;

/* CAS → case: num S PRED | E */
caso : CASE DP NUMERO sent pred | ;

/* PRED → default: S | E */
pred : DEFAULT DP sent | ;

/* P_I → id | V_A | id.id */
parte_izq : ID | var arg | ID PUNTO ID;

```

```

/* V_A → id [ E ] | V_A [ E ] */
var_arg : ID CORA exp CORC | var_arg CORA exp CORC;

/* E → E + E | E - E | E * E | E / E | E % E | V_A | cadena | num | caracter | id (PAR) */
exp : exp MAS exp | exp MEN exp | exp POR exp | exp DIV exp | exp MOD exp | var_arg | CADENA | NUMERO | CARACTER | ID PARA param PARC;

/* PAR → E | L_P */
param : ; | lista_param;

/* L_P = L_P , E | E */
lista_param : lista_param COMA exp | exp;

/* C → C || C | C && C | ! C | ( C ) | E R E | true | false */
cond : cond OR cond | cond AND cond | NEG cond | PARA cond PARC | exp rel exp | TRUE | FALSE;

/* R → < | > | >= | <= | != | == */
rel : MENQUE | MAYQUE | MAYIGUAL | MENIGUAL | DIF | IGUAL ;

%%

/*DEFINICION FUNCION ERROR*/
/*La funcion error lo que va a hacer es una llamada de función en un archivo c que implementa la función error*/

void yyerror (s) char *s {
    printf ("Error: %s\n",s);
}

```

Compilación:

1. Para poder ejecutar el programa tenemos que ir a la consola de nuestro equipo.
2. Tenemos que ingresar al fichero donde se encuentra nuestro archivo.
3. Después ingresamos el siguiente comando: *bison -yd "nombre.y"*.

A terminal window titled "saúl@saul-HP-Pavilion-Notebook: ~/Descargas/compiladores/..." shows the following commands and output:

```

saúl@saul-HP-Pavilion-Notebook:~/Descargas/compiladores/version1$ bison -yd par
ser5.y
parser5.y: aviso: 47 conflictos desplazamiento/reducción [-Wconflicts-sr]
saúl@saul-HP-Pavilion-Notebook:~/Descargas/compiladores/version1$

```