Universidad Pontificia de Salamanca
Informática Teórica
2022/2023

LEX, YACC
Analizador Sintáctico y Semático

Pablo Martín Sánchez
Emma Pérez García

Índice

Introducción

La práctica consiste en la codificación, usando lex/flex y yacc, de un analizador léxico, sintáctico y semántico que reconozca los tokens del lenguaje miniC. El lenguaje miniC que debe utilizarse en la práctica será un subconjunto del lenguaje C con algunas limitaciones.

El objetivo principal de esta práctica es comprender el funcionamiento del analizador sintáctico y semántico, de forma que se profundice en el entendimiento de la primera de las fases de un compilador y ser capaces de generar un analizador sintáctico y semántico para un lenguaje dado.

Contenido de la carpeta

Se entrega en el fichero comprimido de Moodle junto a este documento:
1. El código fuente (.l) y (.y)
2. El código de la lista dinámica
3. El fichero de entrada que se han utilizado para las pruebas (.c)
4. El fichero de salida que se crea con los errores.

Fase sintáctica

De la fase sintáctica se va obtener la siguiente gramática, tal como pide el enunciado que quede reflejado. El fichero yacc se explicará de forma más detallada en la fase semántica.

Comenzamos por explicar la gramática de las diferentes declaraciones de variables y de constantes.
Una declaración de constantes se puede encontrar vacía o ir seguida una o de más declaraciones de constantes.

```
2 declaracionesCtes: /* empty */
3                  | declaracionCte declaracionesCtes
4                  | error
```

Una constante se estructura empezando por una almohadilla, seguida de la palabra reservada *define*, un identificador y el valor de la constante que depende de su tipo. Podemos encontrar una constante de tipo numérico, carácter o cadena. Para recoger los tipos se ha quedado una regla auxiliar "*tipoCte*".

```
5 declaracionCte: '#' DEFINE ID tipoCte

6 tipoCte: NUM
7        | CARACTER
8        | CADENA
```

```
      9       | BOOL
```

Cuando encontramos una variable puede estar vacía o ir seguida de una o más variables. Ésta puede ser entera, de tipo float, char, string o boolean. Le sigue un identificador y un punto y coma. Además, puede reconocer en una misma línea varias declaraciones de variables. Por ello se ha creado una regla más que agrupa varios identificadores separados entre comas.

```
   10 declaracionesVbles: /* empty */
   11                   | declaracionesVble
declaracionesVbles
   12                   | error

   13 declaracionesVble: tipo ID ';'
   14                  | tipo ID '=' exp ';'
   15                  | tipo variosID ';'

   16 variosID: variosID ',' variosID
   17         | ID
   18         | ID '=' exp ';'

   19 tipo: INT
   20     | FLOAT
   21     | CHAR
   22     | STRING
   23     | BOOLEAN
```

Para poder identificar la función principal o main, se encuentra la palabra reservada *main* seguida de paréntesis y las llaves de apertura y cierre. Dentro de la sentencia se pueden encontrar variables declaradas e instrucciones:

```
   24 declaracionMain: MAIN '(' ')' '{' declaracionesVbles instrucciones
'}'
   25                | error
```

La sección de instrucciones es una instrucción seguida o no de más instrucciones. También podemos encontrar la regla *instrucciones* vacía. Los diferentes tipos de instrucciones son la de asignación, de visualización, de lectura, de incremento y decremento, condicionales y bucles.

```
   26 instrucciones: /* empty */
   27              | instruccion instrucciones

   28 instruccion: insAsig
   29            | insVis
   30            | insLec
   31            | insIncDec
   32            | senIfElse
```

```
33              | senWhile
34              | senFor
35              | error
```

La primera de la que se va a hablar es la instrucción de asignación. Para asignar un valor a una variable tiene que seguir la estructura de identificador, el símbolo igual, una expresión y punto y coma. La expresión es más compleja de explicar, puede ser un identificador, un valor numérico, una operación de expresiones, agrupación entre paréntesis y expresiones negativas. Más adelante se explicará mejor.

```
36 insAsig: ID '=' exp ';'

37 exp: ID
38    | valor
39    | exp ope exp
40    | '(' exp ope exp ')'
41    | '-' exp

42 ope: '+'
43    | '-'
44    | '/'
45    | '*'
```

Para seguir, la instrucción de visualización, se trata de identificar la orden printf. Se podrá mostrar una variable, una constante de las definidas en la sección de declaración de constantes o una constante de tipo cadena.

La orden printf va seguida de paréntesis de apertura, los datos a mostrar, el paréntesis de cierre y el punto y coma. Como ya se ha mencionado, los datos que se pueden mostrar pueden ser el identificador de una variable o una constante y una cadena.

```
46 insVis: PRINTF '(' datosMostrar ')' ';'

47 datosMostrar: exp
48             | exp ',' datosMostrar
```

En las instrucciones de lectura se podrá leer una variable utilizando la orden scanf. La palabra scanf va seguida de paréntesis de apertura, un identificador, paréntesis de cierre y el punto y coma.

```
39 insLec: SCANF '(' ID ')' ';'
```

La instrucción de incremento y decremento sigue la estructura de identificador y los operadores – o ++.

```
50 insIncDec: ID operadores ';'

  51 operadores: '+' '+'
  52           | '-' '-'
```

La sentencia condicional de if y else pueden seguir varias sintaxis. La un if simple y la de un if con la sentencia else.
La expresión booleana trata las expresiones con &&, || y !. Cumple con la estructura de identificador, comparadores y una expresión definida anteriormente para la instrucción de asignación.

```
53 senIfElse: IF '(' expresionesBooleanas ')' '{' instrucciones '}'
54          | IF '(' expresionesBooleanas ')' '{' instrucciones '}' ELSE
                '{' instrucciones '}'

55 expresionesBooleanas: expresionBooleana
56                     | OPNOT expresionBooleana
57                     | expresionBooleana OPBOOLEANO expresionesBooleana
58                     | OPNOT expresionBooleana OPBOOLEANO
                         expresionesBooleanas

59 expresionBooleana: exp
60                  | OPNOT exp
61                  | exp COMPARADORES exp
```

La sentencia while tiene la siguiente sintaxis, la palabra while seguida de el paréntesis de apertura, una/s expresión/es booleana/s, el paréntesis de cierre, llave de apertura, instrucciones y la llave de cierre.

```
62 senWhile: WHILE '(' expresionesBooleanas ')' '{' instrucciones '}'
```

La sentencia for se puede escribir con la palabra reservada, seguida de el paréntesis de apertura, después le sigue una instrucción de asignación. De este modo se puede inicializar la variable iteradora. A continuación, se separan las expresiones mediante puntos y comas de la expresión que limita las iteraciones del bucle. De nuevo, encontramos otro punto y coma con un incremento o decremento. Se cierran los paréntesis y entre las llaves se añaden las instrucciones.

```
63 senFor: FOR '(' insAsig expresionesBooleanas ';' ID operadores ')' '{' instrucciones '}'
```

Para detectar los errores es necesaria tener declarada en la zona de definiciones, la función yyerror. Es la encargada de mostrar el mensaje con el tipo de error. Para concretarlo, los posibles errores son escritos en la regla añadiendo "*|error* "de la siguiente manera:

```
instruccion:  insAsig
            | insVis
            | insLec
            | insIncDec
            | senIfElse
            | senWhile
            | error {yyerror("\n ERROR en una instruccion. ");}
            ;
```

Los errores los hemos incluido en las reglas que aparecen en la regla programa:

```
declaracionesCtes: /*vacia*/
                 | declaracionCte declaracionesCtes
                 | error {yyerror("\n Declaracion de constante. ");}
                 ;
```

```
 declaracionesVbles: /*vacia*/
                   | declaracionesVble declaracionesVbles
                   | error {yyerror("\n ERROR Declaracion de variable");}
                   ;
```

```
declaracionMain: MAIN '('')''{' declaracionesVbles instrucciones'}' {printf("Declaracion Main\n");}
               | error {yyerror("\n ERROR Instruccion Main ");};
```

```
instrucciones: /*vacia*/
             | instruccion instrucciones
             ;
instruccion:  insAsig
            | insVis
            | insLec
            | insIncDec
            | senIfElse
            | senWhile
            | error {yyerror("\n ERROR en una instruccion. ");}
            ;
```

Es de esta manera como el programa es capaz de recuperarse de manera automática.

Fase semántica

Fichero lista

Este fichero contiene el código de las funciones de una lista dinámica. Primero se define la estructura que va a tener un elemento en la tabla de símbolos:

```
struct simbolo{
    char nombre[30];
    int tipo; //1 - int, 2 - float, 3 - caracter, 4 - cadena
    int cte; //0 - NO, 1 - SI
    int inicializado; //0 - NO, 1 - SI
};

typedef struct simbolo tipoelementolista;
typedef  struct componente* lista;
struct componente
{
     tipoelementolista elemento;
     lista enlace;
};
```

Para esta práctica necesitamos la función para comprobar que la lista está vacía. De esta manera si lo está reconocemos que no tiene que proceder al recorrido de la lista.

```
int esVaciaLista(lista l)
{
     return (l==NULL);
}
```

A continuación, las funciones de primero y resto para poder ir recorriendo la lista.

```
void primero(lista l, tipoelementolista* x)
{
     if (l==NULL)
     {
          printf("La lista esta vacia");
     }
     else
     {
          *x=l->elemento;
     }
}

void resto(lista l, lista *sigl)
{
     if (l==NULL)
     {
          printf("Lista vacia");
     }
```

```
      else
      {
            *sigl=l->enlace;
      }
}
```

Por último, la función necesaria para insertar en la lista.

```
void insertarLista(lista *l, tipoelementolista x)
{
      lista aux;

      //Reservamos memoria
      aux=(struct componente *)malloc(sizeof(struct componente));

      //Tratamos los errores si no hay memoria
      if (aux==NULL)
      {
            printf("No hay memoria. No se puede realizar la
insercion\n");
      }
      else
      {
            //Si hay memoria insertamos
            aux->elemento=x;
            aux->enlace=*l;
            *l=aux;
      }
}
```

Fichero YACC

```
  zona de definiciones
%%
  zona de reglas
%%
  zona de rutinas de usuario
```

Zona de definiciones

```
{%
  zona de definiciones
%}
declaraciones en yacc
```

Sección de definiciones

En esta sección, declaramos las librerías con las que vamos a trabajar. Se tendrá que incluir el fichero librería que contiene el código de la lista de la tabla de símbolos. También tendremos que definir una variable externa que nos servirá para llevar el recuento de las líneas para el mensaje de líneas totales y para la línea de un error.

Es en esta sección donde aparecen las definiciones de yylex e yyerror. Además, de la definición de la lista y la declaración del prototipo de la función para buscar un símbolo en la tabla de símbolos.

```
%{
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "listascodigodinamicas.h"

extern int nLineas;

void yyerror(const char* msg) {
    fprintf(stderr, "%s. Linea %d\n", msg, nLineas);
}

int yylex(void);

FILE *yyin;
FILE *yyout;

char nfe[20], nfs[20];

lista tablaSimbolos = NULL;

int nErrores = 0;
int tipoAsign;

lista buscarSimbolo(lista tablaSimbolos,char nombre[30]);
void compDeclaraciones(lista* tablaSimbolos, int tipo, int
cte, int ini, char nombre[]);

%}
```

Sección de declaraciones

En esta sección se define la directiva *union*, los símbolos terminales o no terminales y el símbolo inicial de la gramática.
La directiva *union* se utiliza para declarar los tipos de los valores semánticos. Se ha definido el nombre que tendrá el identificador y el tipo del mismo:

```
%union{
    char nombreId[30];
```

```
    int tipo; //1 - int, 2 - float, 3 - caracter, 4 - cadena, 5 - boolean
}
```

Para llevar a cabo la declaración de tipos, hay que modificar el fichero lex para añadir en la zona de reglas lo siguiente:

```
{real} {printf("\n NUM");
            yylval.tipo = 2;
            return NUM;}
```

Para poder definir los símbolos terminales en el fichero yacc también hay que modificar el fichero de lex. Hay que poner por cada regla, un return con el nombre del token que recibirá el fichero .y .
Para que no sea repetitivo, se pueden ver los siguientes ejemplos:

```
printf {printf("\n PRINTF");
          return PRINTF;}
```

```
{cadena} {printf("\n CADENA");
          yylval.tipo = 4;
          return CADENA;}
```

En el fichero yacc si se trata de un símbolo terminal se le añade "*%token*", si se trata de un símbolo no terminal hay que añadir "*%type*".
Para que estos símbolos tengan un tipo concreto se escribe el tipo entre los símbolos de comparación <> tal y como se han definido en la directiva *union*.

```
%token  <nombreId> ID
```

```
%token  <tipo> INT
```

Por tanto, en el fichero yacc la definición de los símbolos terminales queda de la siguiente manera:

```
//Definicion de símbolos terminales y no terminales
%token  <nombreId> ID
%token  DEFINE
%token  <tipo> INT
%token  <tipo> FLOAT
%token  <tipo> CHAR
%token  <tipo> STRING
%token  <tipo> BOOLEAN
%type   <tipo> valor
%type   <tipo> tipo
```

```
%type   <tipo> exp
%token  <tipo> NUM
%token  <tipo> CARACTER
%token  <tipo> CADENA
%token  <tipo> BOOL
%token  MAIN
%token  PRINTF
%token  SCANF
%token  IF
%token  ELSE
%token  WHILE
%token  OPBOOLEANO
%token  OPNOT
%token  COMPARADORES
```

También se han añadido las declaraciones de precedencia, para la instrucción de asignación. De este modo se determina como un operador se anida con otros. Los tokens declarados en la misma declaración tienen la misma precedencia. Y se ordenan de manera creciente de precedencia.

```
%left '+' '-'
%left '/' '*'
%left UNARIO
```

Para definir el símbolo inicial de la gramática se hace de la siguiente manera:

```
//Definicion del símbolo inicial de la gramatica
%start  programa
```

Zona de reglas

Para comenzar, se encuentra la primera regla gramatical, la del programa. La definición del símbolo inicial de la gramática no haría falta añadirlo si la siguiente regla fuera la primera. La estructura es la indicada por el enunciado. Primero encontramos una posible declaración de constantes, lo mismo con las variables y por último la expresión main.

```
%%
programa:  declaracionesCtes declaracionesVbles declaracionMain
{printf("\n Todo correcto numero de lineas %d\n",nLineas); }};
```

Declaración de constantes y variables

Para poder reconocer una constante hay recorrer la tabla de símbolos comprobando que no existe ya una constante o una variable con el mismo nombre. Si no existe aún,

se crea un símbolo y se añade a la tabla de símbolos mediante la función de insertar. La función encargada de hacer lo anterior es la siguiente:

```
void compDeclaraciones(lista* tablaSimbolos, int tipo, int cte, int ini, char nombre[])
{

       lista buscado = buscarSimbolo(*tablaSimbolos, nombre);

    //Comprobacion para saber si existe un identificador con el mismo nombre de antes
    if (buscado != NULL){
       printf("\n ERROR lin: %d: identificador redeclarado %s",nLineas,
               buscado->elemento.nombre);
       }
    else{

        //Simbolo auxiliar de la tabla
       tipoelementolista sim;

        //Se modifica la informacion
        sim.tipo = tipo;
        sim.cte  = cte;
        sim.inicializado = ini;
        strcpy(sim.nombre, nombre);

        //Se añade simbolo auxiliar a la tabla de simbolos
        insertarLista(tablaSimbolos, sim);
   }
}
```

En la zona de reglas podremos encontrar la llamada a la función:

```
declaracionesCtes: /*vacia*/
                             | declaracionCte declaracionesCtes
                             | error {yyerror("\n Declaracion de constante. ");}
                             ;
declaracionCte: '#' DEFINE ID valor  {compDeclaraciones(&tablaSimbolos, $4, 1, 1, $3);};

valor: NUM {$$ = $1;}
             |CARACTER {$$ = $1;}
             |CADENA {$$ = $1;}
             |BOOL {$$ = $1;}
             ;
```

Para la declaración de variables si no fuera porque en una misma línea se pueden declarar varias variables y se pueden inicializar, se podría reutilizar la función anteriormente comentada. Mucha parte del código es repetida como también se va a ver más adelante.

```
declaracionesVbles: /*vacia*/
                    | declaracionesVble declaracionesVbles
                    | error {yyerror("\n ERROR Declaracion de variable");}
                                ;
declaracionesVble: tipo ID ';' {compDeclaraciones(&tablaSimbolos, $1, 0, 0, $2);};

                | tipo ID '=' exp ';' {lista buscado = buscarSimbolo(tablaSimbolos, $2);

                                      //Comprobacion para saber si existe el identificador
                                      if (buscado != NULL){
```

```
                                                printf("\n ERROR lin: %d: identificador
                                                        redeclarado %s",nLineas,
                                                      buscado->elemento.nombre);
                                            }
                                            else{

                                             //Comprobacion de tipos
                                                    if ($1 == $4 || ($1 >= $4 && ($1 == 1 || $1
== 2))){

                                                //Simbolo auxiliar de la tabla
                                                tipoelementolista sim;

                                                //Se modifica la informacion
                                                sim.tipo = $1;
                                                sim.cte  = 0;
                                                 sim.inicializado = 1;
                                                strcpy(sim.nombre, $2);

                                                //Se añade simbolo a la tabla
                                                insertarLista(&tablaSimbolos, sim);

                                             }
                                              else {
                                                printf("\n ERROR lin: %d: Tipos
                                                       incompatibles en la
                                                       asignación.",nLineas);
                                              }
                                    }
                                                                    };
                            | tipo variosID ';' {tipoAsign = $1;}

variosID: variosID ',' variosID
                |ID {compDeclaraciones(&tablaSimbolos, tipoAsign, 0, 0, $1);}
                |ID '=' exp ';' {compDeclaraciones(&tablaSimbolos, tipoAsign, 0, 0, $1);}


tipo: INT {$$ = $1;}
        |FLOAT {$$ = $1;}
        |CHAR {$$ = $1;}
        |STRING {$$ = $1;}
        |BOOLEAN {$$ = $1;}
;
```

Declaración Main

Dentro de la sentencia main se pueden encontrar variables declaradas e instrucciones:

```
declaracionMain: MAIN '('')''{' declaracionesVbles instrucciones'}'
{printf("Declaracion Main\n");} | error {yyerror("\n ERROR Instruccion
Main ");};
```

La sección de instrucciones es una instrucción seguida o no de más instrucciones.

```
instrucciones: /*vacia*/
                    | instruccion instrucciones
                    ;
```

```
instruccion: insAsig | insVis | insLec | insIncDec | senIfElse |
senWhile | error {yyerror("\n ERROR en una instruccion. ");}
                 ;
```

Instrucción de visualización

La primera de la que se va a hablar es la instrucción de asignación. Para asignar un valor a una variable tiene que seguir la estructura de identificador, el símbolo igual, una expresión y punto y coma.

La expresión puede ser un simple identificador, un número, una operación de expresiones, puede haber paréntesis para agrupar y expresiones negativas.

Esto último es importante porque como se ha explicado al principio existen precedencias, aquí encontramos que el mismo símbolo "-" necesita dos precedencias distintas, la primera como operador y la segunda para hacer negativa la expresión. Es por lo que se pone "*%prec UNARIO*":

```
exp: ID      {  lista buscado = buscarSimbolo(tablaSimbolos, $1);

                //Comprobacion para saber que existe el identificador
                if (buscado == NULL){
                     printf("\n ERROR lin: %d: Variable %s sin declarar.",nLineas, $1);
                }
                else{

                   //Comprobacion para saber si la variable ya esta inicializada
                   if (buscado->elemento.inicializado != 1){
                           printf("\n ERROR lin: %d: Acceso a la variable %s sin
                                   inicializar.",nLineas, buscado->elemento.nombre);
                     }
                     else {
                           $$ = buscado->elemento.tipo;
                     }
                }
             };
    |valor {$$ = $1;}
    |exp ope exp { //Comprobacion para saber que existe el identificador
                   if ($1 > 2 && $3 > 2){
                       printf("\n ERROR lin: %d: Tipos incompatibles valor no
                              numerico.",nLineas);
                    }
                   //Se pasa el tipo en cualquier caso para que el programa pueda seguir
                   comprobando en otros casos
                    $$ = $1;
                  };
    |'(' exp ope exp')' { //Comprobacion para saber que existe el identificador
                          if ($2 > 2 && $4 > 2){
                              printf("\n ERROR lin: %d: Tipos incompatibles valor no
                                     numerico.",nLineas);
                          }
                           //Se pasa el tipo en cualquier caso para que el programa
                           pueda seguir comprobando en otros casos
                          $$ = $2;
                         };
    |'-' exp %prec UNARIO {  //Comprobacion para saber que existe el identificador
                             if ($2 > 2){
                                 printf("\n ERROR lin: %d: Tipos incompatibles valor no
                                        numerico.",nLineas);
                             }
```

```
                                //Se pasa el tipo en cualquier caso para que el programa
                                pueda seguir comprobando en otros casos
                                $$ = $2;
                            };

ope: '+' | '-' | '/' | '*'
   ;
```

También hay que comprobar como siempre que el identificador exista en la tabla de símbolos, además de comprobar que esté inicializada.

Si vamos a la sección del código de la instrucción, para que la asignación se haga correctamente hay que buscar el identificador primero. Cuando se haya encontrado habrá que comprobar que no se trate de una constante. Ya que a una constante no se puede modificar el valor. A continuación se comprueban los tipos y si todo es correcto se inicializa la variable. De lo contrario, se mostrarán los errores oportunos:

```
insAsig: ID '=' exp ';'  { printf("\n Instrucción Asignacion");
                           lista buscado = buscarSimbolo(tablaSimbolos, $1);

                           //Comprobacion para saber que existe el identificador
                          if (buscado == NULL){
                             printf("\n ERROR lin: %d: Variable %s sin declarar.",nLineas, $1);
                          }
                          else{

                             //Comprobacion para asegurarse de que se trate de una variable
                             if (buscado->elemento.cte == 0){

                                   //Comprobacion de tipos
                                   if (buscado->elemento.tipo == $3 || (buscado->elemento.tipo
                                   >= $3 && ($3 == 1 || $3 == 2))){

                                            //La variable esta inicializada
                                            buscado->elemento.inicializado = 1;
                                       }
                                       else {
                                              printf("\n ERROR lin: %d: Tipos incompatibles en
                                                     la asignación.",nLineas);
                                       }
                              }
                              else {
                                   printf("\n ERROR lin: %d: Intento de modificación de
                                          la constante %s.",nLineas, buscado->elemento.nombre);
                                   }
                            }
                            };


exp: ID      {  lista buscado = buscarSimbolo(tablaSimbolos, $1);

                //Comprobacion para saber que existe el identificador
                if (buscado == NULL){
                     printf("\n ERROR lin: %d: Variable %s sin declarar.",nLineas, $1);
                }
                else{
```

```
                    //Comprobacion para saber si la variable ya esta inicializada
                    if (buscado->elemento.inicializado != 1){
                            printf("\n ERROR lin: %d: Acceso a la variable %s sin
                                    inicializar.",nLineas, buscado->elemento.nombre);
                    }
                    else {
                            $$ = buscado->elemento.tipo;
                    }
                }
            };
    |valor {$$ = $1;}
    |exp ope exp { //Comprobacion para saber que existe el identificador
                    if ($1 > 2 && $3 > 2){
                        printf("\n ERROR lin: %d: Tipos incompatibles valor no
                                numerico.",nLineas);
                    }
                   //Se pasa el tipo en cualquier caso para que el programa pueda seguir
                   comprobando en otros casos
                    $$ = $1;
                   };
    |'(' exp ope exp')' { //Comprobacion para saber que existe el identificador
                          if ($2 > 2 && $4 > 2){
                              printf("\n ERROR lin: %d: Tipos incompatibles valor no
                                      numerico.",nLineas);
                          }
                           //Se pasa el tipo en cualquier caso para que el programa
                           pueda seguir comprobando en otros casos
                          $$ = $2;
                         };
    |'-' exp %prec UNARIO {  //Comprobacion para saber que existe el identificador
                             if ($2 > 2){
                                 printf("\n ERROR lin: %d: Tipos incompatibles valor no
                                         numerico.",nLineas);
                             }
                              //Se pasa el tipo en cualquier caso para que el programa
                             pueda seguir comprobando en otros casos
                             $$ = $2;
                          };

ope: '+' | '-' | '/' | '*'
   ;
```

Instrucción de visualización

Para seguir, la instrucción de visualización, se trata de identificar la orden printf. Se podrá mostrar una variable, una constante de las definidas en la sección de declaración de constantes o una constante de tipo cadena.

La orden printf va seguida de paréntesis de apertura, los datos a mostrar, el paréntesis de cierre y el punto y coma. Como ya se ha mencionado, los datos que se pueden mostrar pueden ser el identificador de una variable o una constante y una cadena. Eso al fin y al cabo es una lista de expresiones. Por lo que realmente los datos a mostrar pueden ser o una expresión o una expresión, una coma y más expresiones. La regla de la expresión ya cuenta con el tratamiento de comprobar que ya existe un identificador con ese nombre. Si es así comprueba que la constante o variable ya esté inicializada, porque si no es así no se sabe que se puede mostrar.

```
insVis: PRINTF '(' datosMostrar ')' ';' {printf("\n Instrucción visualizacion");}  ;
datosMostrar: exp
```

```
                    |exp ',' datosMostrar
            ;
```

Instrucción de lectura

En las instrucciones de lectura se podrá leer una variable utilizando la orden scanf. La palabra scanf va seguida de paréntesis de apertura, un identificador, paréntesis de cierre y el punto y coma. Para identificarlo correctamente se comprueba que existe una constante o variable con ese nombre, si es así comprobamos que sea una variable obligatoriamente e inicializamos la variable si no lo estaba aún. De no ser así se mostrará un mensaje de error, debido a que no puede modificarse el valor de la constante.

```
insLec: SCANF '(' ID ')' ';' {  printf("Instrucción lectura\n");
                                lista buscado = buscarSimbolo(tablaSimbolos, $3);

                                //Comprobacion para ver si existe el identificador
                                if (buscado == NULL){
                                    printf("\n ERROR lin: %d: Variable %s sin
                                          declarar.",nLineas, $3);
                                }
                                else {


                                        //Comprobacion para saber que NO se trata de
                                        una constante
                                        if(buscado->elemento.cte == 0){

                                           //Comprobacion para saber si la variable
                                              esta inicializada
                                           if (buscado->elemento.inicializado == 0){

                                                 //El elemento se inicializa
                                                 buscado->elemento.inicializado = 1;
                                           }
                                        }
                                        else {
                                               printf("\n ERROR lin: %d: Intento
                                                           de modificación de la
                                                           constante %s.",nLineas,
                                                           buscado->elemento.nombre);
                                              }
                                        }

                                                 }
                                                 ;
```

Instrucción de incremento y decremento

La instrucción de incremento y decremento sigue la estructura de identificador, operadores – o ++ y el punto y coma que caracteriza el final de cada instrucción. Como ya se ha visto se comprueba que exista una variable con ese nombre, después se comprueba que realmente si sea una variable, ya que a una constante no se le puede modificar el valor, que esté inicializada la variable y que se trate de una variable de tipo entera.

```
insIncDec: ID operadores ';' {  printf("Instrucción Inc-Dec\n");
```

```
                                lista buscado = buscarSimbolo(tablaSimbolos, $1);

                                //Comprobacion para saber que existe el identificador
                               if (buscado == NULL){
                                     printf("\n ERROR lin: %d: Variable %s sin
                                            declarar.",nLineas, $1);
                               }
                               else {

                                   //Comprobacion para saber que NO se trata de una
                                       contante --> es una variable
                                   if(buscado->elemento.cte == 0){

                                       //Comprobacion variable incializada
                                       if (buscado->elemento.inicializado == 1){

                                           //Comprobacion de tipos --> Tiene que ser un
                                             entero
                                           if (buscado->elemento.tipo != 1){

                                               //Si no es un entero error:
                                               printf("\n ERROR lin: %d: Tipos
                                                      incompatibles en la
                                                    asignación.",nLineas);

                                           }
                                       }
                                        else {


                                          printf("\n ERROR lin: %d: Acceso a la variable
                                                  %s sin inicializar.",nLineas,
                                                buscado->elemento.nombre);
                                       }
                                   }
                                   else {
                                        printf("\n ERROR lin: %d: Intento de
                                               modificación de la constante
                                               %s.",nLineas, buscado->elemento.nombre);
                                   }
                               }
                           }
                          ;
operadores: '+''+' | '-''-'
            ;
```

Sentencia condicional

La sentencia condicional de if y else pueden seguir varias sintaxis. La un if simple y la de un if con la sentencia else.

La expresión booleana trata las expresiones con &&, || y !. Cumple con la estructura de identificador, comparadores y una expresión definida anteriormente para la instrucción de asignación. Primero, se comprueba que existe una variable con ese nombre. Después, se comprueba que la variable esté inicializada. Para continuar se comprueban los tipos para que no haya incompatibilidades:

```
senIfElse: IF '(' expresionesBooleanas ')' '{' instrucciones '}'
          | IF '(' expresionesBooleanas ')' '{' instrucciones '}' ELSE '{' instrucciones
            '}' {printf("Sentencia IF/ELSE\n");};

expresionesBooleanas: expresionBooleana
```

```
                                    | OPNOT  expresionBooleana
                                    | expresionBooleana OPBOOLEANO expresionesBooleanas
                                    | OPNOT expresionBooleana OPBOOLEANO
expresionesBooleanas
                                   ;

expresionBooleana:  exp
                  | OPNOT exp
                  | exp COMPARADORES exp {  //Comprobacion de compatibilidad de tipos
                                            if ($1 != $3 && ($1 > 2 || $3 > 2)){
                                                 printf("\n ERROR lin: %d: Tipos
                                                           incompatibles en la
                                                           asignación.",nLineas);

                                            }
                                          }
                                          ;
```

Sentencia while

La sentencia while tiene la siguiente sintaxis. No requiere de ninguna acción en específico.

```
senWhile: WHILE '(' expresionesBooleanas ')' '{' instrucciones
'}' {printf("Sentencia WHILE\n");}
;

%%
```

Zona de funciones del usuario

En esta parte se declaran las funciones main y la función encargada de buscar el símbolo en la lista de la tabla de símbolos.

En la primera función(main) se lee el fichero de entrada, que contiene el código a analizar. Tras ello, se crea un fichero para guardar los errores de en la salida. Después, se llama a la función yyparse, encargada de leer tokens, ejecuta acciones y termina cuando encuentra el final del fichero. Se comprueba que no haya ningún problema en la apertura de los ficheros.

```
int main()
{
     yyin=fopen("ejemplo.c","r");

     yyin = fopen("ejemplo.c", "r");
     yyout = fopen ("salida.txt", "w");
     yyparse();
     if (yyin != NULL && yyout!= NULL){
          yylex();
          fprintf(yyout,"Número de líneas: %d\n", nLineas);
```

```
            if(nErrores != 0){
                fprintf(yyout, "Número de errores semánticos:
                        %d\n", nErrores);
            }
        }
        else {printf("ERROR de apertura ");}

}
```

Para continuar, la función encargada de buscar el símbolo en la tabla de símbolos. Funciona de manera que recorra la lista comparando los nombres de los identificadores. Si hay coincidencias, la función devuelve una lista con el elemento encontrado.

```
lista buscarSimbolo(lista tablaSimbolos, char nombre[30]) {

    lista sim = (lista)malloc(sizeof(struct componente));

    //Recorrido de la tabla de simbolos
    while (!esVaciaLista(tablaSimbolos)){

        //Se lee el primero
        primero(tablaSimbolos, &sim);

        //Se comprueban los nombres
        if (strcmp(nombre, sim->elemento.nombre) == 0){

            //Si hay coincidencias fin
                return sim;
        }

        //Se pasa al siguiente
        resto(tablaSimbolos, &tablaSimbolos);
    }
    return NULL;
}
```