Documentación práctica sintáctico y semántico

Hecho por: Juan Vázquez López y Alejandro Prieto Ramírez.

1 Zona de C:

En la zona de C definiremos el código C que debe estar implementado antes del análisis sintáctico.

Empezamos declarando las librerias necesarias.

```
%{

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

Así declaramos e inicializamos las variables contador.

- **nLinea** una variable contador para llevar la cuenta del número de lineas. Al provenir del lexico esta variable ya esta inicializada.

- **nErrores** una contador que lleva la cuenta del número de errores.

```
extern int nLineas;
int nErrores = 0;
```

Seguimos definiendo los ficheros a utilizar. Cabe destacar los siguientes ficheros:

- **yyout** registrara los datos que hemos utilizado durante el desarrollo para comprobar el funcionamiento en el fichero salidacompleta.txt.
- **yyoutc** este por otro lado registrara los mensajes de error en un fichero salida.txt.

```
FILE *yyin;
FILE *yyout;
FILE *yyoutc;
```

Después implementamos las funcionalidades del análisis definiendo:

- **yyerror** que recibirá un mensaje de error, imprimiendolo junto con la linea en que esta. Además sumara uno a el numero de errores totales.

```
void yyerror(const char* msg) {
    fprintf(yyoutc, "\nLinea: %d - %s", nLineas, msg);
    nErrores++;
}
int yylex(void);
```

Por otro lado tendremos que definir la estructura de nuestra tabla de símbolos. Para definir la estructura decidimos utilizar listas enlazadas. Cada elemento se compondra de:

- **nombre** Almacenamos el nombre del símbolo.
- **tipo** Almacena los siguientes valores en función de su tipo: int 1 , float 2 , char 3 , string 4 , boolean 5.
- **cte** Indica si es contante (1) o si no lo es (1).
- **init** Indica si eestá inicializado (1) o si no lo está (1).
- **sig** Un puntero que apunta al siguiente elemento.

Para almacenar nuestra tabla de símbolos tendremos:

- **tablasSim** que será un puntero que apunte al primer elemento de nuestra lista.
- **simUlt** otro puntero que apunta al último de la lista.

Por último declaramos la estructura de la función buscarSimbolo() que comprobará si un símbolo se encuentra en la tabla devolviendolo si está y así poder hacer los tratamientos requeridos.

```
struct simbolo
{
```

```
    char nombre[30];
    int tipo; // int 1 | float 2 | char 3 | string 4| boolean 5
    int cte; // si 1 | no 0
    int init; // si 1 | no 0
    struct simbolo *sig;
};
struct simbolo *tablaSim = NULL;
struct simbolo *simUlt = NULL;

struct simbolo* buscarSimbolo(struct simbolo *tablaSim, char nombre[30]);

%}
```

2 Zona de Definiciones:

2.1 Sintáctico:

2.1.1 Tokens:

- **COMENTARIO** reconoce los comentarios.
- **ID** reconoce los identificadores.
- **DEFINE** reconoce la palabre define.
- **MAIN** reconoce la palabra main.
- **NUM** reconoce los números.
- **CADENA** reconoce las cadenas.
- **CARACTER** reconoce los caracteres.
- **BOOLVAL** reconoce la palabra reservada true o false.
- **INT** reconoce la palabra reservada int.
- **FLOAT** reconoce la palabra reservada float.
- **CHAR** reconoce la palabra reservada char.
- **STRING** reconoce la palabra reservada string.
- **BOOLEAN** reconoce la palabra reservada bool.
- **PRINTF** reconoce la palabra reservada printf.
- **SCANF** reconoce la palabra reservada scanf.
- **ELSE** reconoce la palabra else.
- **IF** reconoce la palabra if.
- **WHILE** reconoce la palabra while.

```
/*define los tokens*/
%token COMENTARIO
%token ID
%token DEFINE
%token MAIN
%token NUM
%token CADENA
%token CARACTER
%token BOOLVAL
%token INT
%token FLOAT
%token CHAR
%token STRING
%token BOOLEAN
%token PRINTF
%token SCANF
%token ELSE
%token IF
%token WHILE
```

2.1.2 Aclaraciones de preferencia:

Por otro lado definimos que la multiplicación y división sean preferentes a la suma y la resta y que la aparición individual sobre todos(de esta forma podremos controlar los casos en lso que un elemento es precedidido por -).

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

2.2 Semántico:

Esta misma sección de codigo desde el punto de vista semántico contiene un union que definirá los tipos de los posibles valores a tomar por yylval.

```
%}
%union{
    char nombre[30];
    int tipo;
}
```

Si analizamos los tokens desde el punto de vista semántico estos ahora recibirán una definición del tipo de valor que pueden tomar segun el union anterior. asi ID su valor sera del tipo mientras que num sera del tipo . Ademas debemos añadir ciertos type que definirán el tipo de valor que toma un símbolo no terminal.

```
%token <nombre> ID
%token <tipo> NUM

%type <tipo> cte
```

3 Zona de Reglas:

3.1 Sintáctico:‘

3.1.1 Programa:

La gramática que reconoce nuestro análisis sintáctico reconoce este símbolo como el inicial. El símbolo programa a su vez nos lleva a la siguiente estructura:

```
programa: declaracionesCtes declaracionesVariables MAIN '(' ')' '{'
declaracionesVariables instrucciones '}'
```

3.1.2 Declaración De Constantes:

- **declarionesCtes** que es un símbolo invocado por programa y puede resultar en que se encuentre vacio o declaracionCte declaracionesCtes
- **declaracionCte** símbolo invocado por por declaracionesCtes y puede seguir la siguiente estructura *‘#’* seguido de los tokens *DEFINE ID* vistos anteriormente y el token *cte* que veremos en el siguiente apartado.
  Además podemos encontrar un comentario o puede ser que de un error.
- **cte** símbolo invocado por declaracionCte que puede ser un número una cadena o un caracter o un valor booleano.

```
declaracionesCtes: /* empty */
         | declaracionCte declaracionesCtes

declaracionCte: '#' DEFINE ID cte
          | COMENTARIO
          | error

cte: NUM
   | CADENA
   | CARACTER
   | BOOLVAL
```

3.1.3 Declaración De Variables

- **declaracionesVariables** que es un símbolo invocado por program y puede estar vacío o declaracionesVariable declaracionesVariables.

- **declaracionesVariable** es un símbolo invocado por declaracionesVariables y puede seguir la siguiente estructura *tipo* que explicaremos en el siguiente apartado seguido del token *IDs ;*.
  Además podemos encontrar un comentario o puede que de un error.
- **tipo** es un símbolo invocado por declaracionesVariable y nos indica que el *ID* puede ser un *INT* o un *FLOAT*.
- **IDs** Este simbolo agrupa las apariciones de *ID* o *ID = cte* o la aparición de combinaciones de las anteriores separadas por coma.

```
declaracionesVariables: /* empty */
              | declaracionesVariable declaracionesVariables

declaracionesVariable: tipo IDs ';'
             | COMENTARIO
             | error

IDs: IDs ',' IDs
   | ID
   | ID '=' cte

tipo: INT
    | FLOAT
    | CHAR
    | STRING
    | BOOLEAN
```

3.1.4 Instrucciones:

- **instrucciones** es un símbolo invocado por program y puede estar vacío o contener instruccion instrucciones.
- **instruccion** es un símbolo invocado por instrucciones y define todos los tipos de instrucción reconocidos por el programa , un comentario o un tipo de error.

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

instruccion: asignacion
       | visualizacion
       | lectura
       | incremento
       | condicional
       | repetitiva
       | COMENTARIO
       | error
```

3.1.4.1 Asignación:

- Esta intrucción sigue el esquema: primero el token *ID* luego un = luego el símbolo *expresion ;*

```
asignacion: ID '=' expresion ';'
```

3.1.4.2 Visualizacion:

- Esta instrución sigue el esquema: primero el token *PRINTF* , luego entre paréntesis el símbolo *visualizado* y por último ,.
- **Visualizado** es un símbolo invocado por visualizacion que puede llevar el token expresion o combinaciones de este separados por coma.

```
visualizacion: PRINTF '(' visualizado ')' ';'

visualizado: expresion
       | expresion ',' visualizado
```

3.1.4.3 Lectura:

- La instrución lectura sigue le sigiente esquema: primero el token *SCANF* luego entre paréntesis encontramos el token *ID* que hace referencia al elemento que afecta scanf y por último un *;*.

```
lectura: SCANF '(' ID ')' ';'
```

3.1.4.4 Incremento:

- La instrcucción incremento sigue el esquema: primero el token *ID* seguido de *++* o – haciendo esto referencia a si incrementamos o disminuimos en 1 el contenido del token *ID*.

```
incremento: ID '+' '+' ';'
      | ID '-' '-' ';'
```

3.1.4.5 Instrucción Condicional:

- La instrucción condicional puede ser un token *IF* seguido de una *expbool* de las *instrucciones* el con o sin el token *ELSE*. Además puede ser un *SWITCH* llamando a un *ID* entre paréntesis y a *CASES* entre corchetes.
- **CASES** puede ser un *CASE* seguido de *ctei : instruccion BREAK ;* y la repetición de mas *CASES*.
- **expbool** es un símbolo invocado por la instrucción condicional que puede ser un *expresion* , otra expbool , una expresión precedido por distinto o entre paréntesis, la igualación de dos expresiones o dos expbool con un operador *and* o *or*.

- **igualación** reconoce las combinaciones de los operadores =><!.

```
condicional: IF '(' expbool ')' '{' instrucciones '}' ELSE '{' instrucciones '}'
           | IF '(' expbool ')' '{' instrucciones '}'
           | SWITCH '(' ID ')' '{' cases '}'

cases: CASE cte ':' instruccion BREAK ';'
     | CASE cte ':' instruccion BREAK ';' cases


expbool: expresion
       | '(' expbool ')'
       | '!' expresion
       | expresion igualacion expresion
       | expbool '|' '|' expbool
       | expbool '&' '&' expbool

igualacion: '=' '='
      | '!' '='
      | '<' '='
      | '<'
      | '>' '='
      | '>'
```

3.1.4.6 Instrcución repetitiva :

- La instrucción repetitiva sigue un esquema parecido al de la condicional. En este empezamos con el token *WHILE* y luego como en el if una expbool seguida de sus instrcuccines. O bien un *FOR*.

```
repetitiva: WHILE '(' expbool ')' '{' instrucciones '}'
          | FOR '(' expresion ';' expbool ';' expresion ')' '{' instrucciones '}'
```

3.1.4.7 Elemento expresion

- El símbolo expresión pueder hacer referencia al token *ID* o al token *cte* , además puede ser un conjunto con la estructuctura *expresion operador* expresion y una expresion entre paréntesis o precedida por un menos.
- operador

: este símbolo es llamado por expresión y engloba los caracteres de suma, resta, multiplicación y división.

```
expresion: ID
         | cte
         | expresion operador expresion
```

```
        | '(' expresion ')'
        | '-' expresion

operador: '+'
        | '-'
        | '*'
        | '/'
```

3.2 Semántica:

3.2.1 buscarSimbolo:

- A esta fucnión le pasamos la tabla de símbolos y el nombre que queremos buscar. A mayores declaramos una variable this que tomará la posición actual de la lista.
- Para empezar a recorrer debemos comprobar que la tabla no está vacía si este es el caso devolveremos *null* ; si no mientras la posición actual no sea nula recorrerenmos la lista comprobando que el nomre de la posición en la que nos encontramos no coincida con el nombre que recibimos como parámetro. En el caso de que los nombres coincidan devolvemos el elemento y si no devolveremos *null*.

```
struct simbolo* buscarSimbolo(struct simbolo *tablaSim, char nombre[30])
{
    struct simbolo *this = tablaSim;

    if(this == NULL)
    {
        return NULL;
    }
    while(this != NULL)
    {
        if(strcmp(this->nombre, nombre) == 0)
            return this;
        this = this->sig;
    }
    return NULL;
}
```

3.2.2 insertarSimbolo:

- A esta fucnción le pasamos el nombre del objeto que queremos insertar, el tipo del elemento , y dos variables una para saber si es una constante y la otra nos indicca si está inicializado. A mayores llamamos dentro a la función buscarSimbolo y guardamos su resultado en una variable a la que llamamos *pos*.
- Si *pos* es distinto de nulo indica que no está declarado (incrementamos el uno el contador de errores).
- Si *pos* es nulo creamos un nuevo símbolo que reserva en memoria un espacio y almacena en el los nuevos datos, luego modificamos los punteros de la lista para que apunten al nuevo

elemento

```
void insertarSimbolo(char nombre[30], int tipo, int cte, int init)
{
      struct simbolo* pos = buscarSimbolo(tablaSim, nombre);
      if(pos != NULL)
      {
           fprintf(yyoutc,"\nERROR: linea %d: Identificador %s redeclarado",
nLineas+1, nombre);
           nErrores++;
      }
      else
      {
        struct simbolo *tmp;
        tmp = (struct simbolo*) malloc(sizeof(struct simbolo));
        strcpy(tmp->nombre, nombre);
        tmp->tipo = tipo;
        tmp->cte = cte;
        tmp->init = init;
        tmp->sig = NULL;

        if(tablaSim == NULL)
        {
            tablaSim = tmp;
            simUlt = tmp;
        }
        else
        {
               simUlt->sig = tmp;
               simUlt = tmp;
        }

      }
}
```

3.2.3 declararTipo:

- Esta función es utilizada a la hora de declarar las variables para asignar su tipo una vez han sido insertados todos los *ID* de la expresion *declararVariable*. Así recorrera la tabla de símbolos buscando los elementos con tipo 0 o negativo y en ese caso les dara el tipo que recibe como parametro y comprobara si los elementos inicializados tienen el mismo tipo, mostrando un error si no es igual.

```
void declararTipo( int tipo)
{
    struct simbolo *this = tablaSim;
```

```
        if(this == NULL)
        {
            return;
        }
        while(this != NULL)
        {
            if( this->tipo <= 0)
            {
                if(this->tipo * (-1) != tipo && this->init == 1)
                {
                    fprintf(yyoutc,"\nERROR: linea %d: Valor de distinto tipo",
nLineas+1);
                    nErrores++;
                }
                this->tipo = tipo;
            }
            this = this->sig;
        }
    }
```

3.2.4 errorIncremento:

- A esta función recibe el nombre del elemento que está comprobando y comprueba que: esté inicializado, esté declarado y el tipo sea o entero o real.

```
    void errorIncremento(char nombre[30])
    {
             struct simbolo* pos = buscarSimbolo(tablaSim, nombre);
              if(pos == NULL)
              {
               fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no declarado",
nLineas+1, nombre);
               nErrores++;
              }
              else if(pos->tipo > 2)
              {
                 fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no numerico",
nLineas+1,nombre);
               nErrores++;
              }
              else if(pos->init == 0)
              {
               fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no inicializado",
nLineas+1, nombre);
               nErrores++;
              }
```

```
    }
```

3.2.5 Tipo:

- La expresión que encotramos en tipo *$$=$1* indica que el elemento tipo guarda el valor de *$1*.

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

3.2.6 Asignacion:

- Esta pieza de código comprueba que el elemento está declarado, no es una constante y que los tipos coinciden. Si lo anterior es correcto indica que el elmento actual está inicializado.

```
    asignacion: ID '=' expresion ';' {
                            struct simbolo* pos = buscarSimbolo(tablaSim, $1);
                        if(pos == NULL)
                        {
                             fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no
declarado",nLineas+1, $1);
                             nErrores++;
                        }
                        else if(pos->cte == 1)
                        {
                            fprintf(yyoutc,"\nERROR: linea %d: Constante %s no
modificable",nLineas+1, $1);
                             nErrores++;
                        }
                        else
                        {
                            if(pos->tipo == $3)
                                pos->init = 1;
                            else
                            {
                                fprintf(yyoutc,"\nERROR: linea %d: Identificador
de distinto tipo",nLineas+1);
                                nErrores++;
                            }
                        }
                    }
    ;
```

3.2.7 Expresion:

- Comprobamos que el *ID* esté declarado e inicializado y si está declarado asignamos a expresión el tipo de la posición actual.

```
    expresion: ID {
                            struct simbolo* pos = buscarSimbolo(tablaSim, $1);
                        if(pos == NULL)
                        {
                             fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no
declarado",nLineas+1, $1);
                             nErrores++;
                        }
                        else
                        {       if (pos->init == 0)
                            {
                                  fprintf(yyoutc,"\nERROR: linea %d: Identificador %s
no inicializado",nLineas+1, $1);
                                  nErrores++;
                            }
                            $$ = pos->tipo;
                        }
                }
```

- Para las siguientes operaciones de caracter numérico debemos comprobar a mayores que solo las realizan valores numéricos.

```
     | expresion operador expresion {
                        if($1 > 2 && $3 > 2)
                        {
                            fprintf(yyoutc,"\nERROR: linea %d: Expresion no numerico",
nLineas+1);
                            nErrores++;
                        }
                             $$ = $1;
                        }
     | '(' expresion ')' {
                        if($2 > 2)
                        {
                            fprintf(yyoutc,"\nERROR: linea %d: Expresion no numerico",
nLineas+1);
                            nErrores++;
                        }
                             $$ = $2;
                    }
     | '-' expresion  %prec UNARIO  {
                        if($2 > 2)
```

```
                    {
                        fprintf(yyoutc,"\nERROR: linea %d: Expresion no numerico",
nLineas+1);
                        nErrores++;
                    }
                        $$ = $2;
                     }
;
```

3.2.8 Lectura:

- Si el elmento está declarado y no es una constante, este se marca como incializado.

```
    lectura: SCANF '(' ID ')' ';' {
                            struct simbolo* pos = buscarSimbolo(tablaSim, $3);
                        if(pos == NULL)
                        {
                             fprintf(yyoutc,"\nERROR: linea %d: Identificador %s no
declarado",nLineas+1, $3);
                             nErrores++;
                        }
                        else if(pos->cte == 1)
                        {
                            fprintf(yyoutc,"\nERROR: linea %d: Constante %s no
modificable",nLineas+1, $3);
                             nErrores++;
                        }
                        else
                        {
                            pos->init = 1;
                        }
                      }
    ;
```

4 Zona de Funciones del Usuario:

4.1 Funcion Main:

Controlamos los posibles errores de apertura del fichero a leer y a escribir.

```
    if(yyin == NULL )
    {
        printf("ERROR DE APERTURA\n");
        return 0;

    }
```

```
if(yyout == NULL )
{
    printf("ERROR DE APERTURA DEL FICHERO DE ESCRITURA\n");
    return 0;
}
if(yyoutc == NULL )
{
    printf("ERROR DE APERTURA DEL FICHERO DE ESCRITURA COMPLETA\n");
    return 0;
}
```

Llamamos a la función que solicita a yylex los tokens de entrada y hace las respectivas comprobaciones.

```
yyparser();
```

Por último imprimimos un mensaje de todo correcto o error en la salida.txt segun el número de errores.

```
if(nErrores == 0)
    fprintf(yyoutc,"\n Todo correcto numero de lineas: %d",nLineas);
else
    fprintf(yyoutc,"\nNumero de errores semanticos: %d",nErrores);
```

5 Contenido extra:

Se adjunta al trabajo un MakeFile para que se pueda ver como hemos compilado el proyecto simplemente llamando a la comando:

- **make** lo compila y ejecuta.
- **make open** lo compila ejecuta y muestra con less el fichero de salida.txt.
- **make opencomplete:** hace lo que make pero muestra en less el fichero de salidacompleta.txt.