CORRECCIÓN DE ERRORES PRÁCTICA COMPILADORES JULIO 2017.

A resolver:
- No se insertan los identificadores en la lista de variables cuando no tienen expresión de inicialización.

Para cada tipo de identificador (let, var o float), cuando no tienen expresión de inicialización, adquieren un valor por defecto (0 en el caso de let y var, 0.0 en el caso del float).

```
asig_var:       ID { printf("asig_var -> ID\n"); insertarVar(&lVar,$1,0,0); }
asig_let:       ID { printf("asig_let -> ID\n"); insertarVar(&lVar,$1,0,1); }
asig_float      :       ID { printf("asig_float -> ID\n"); insertarVar(&lVar,$1,0.0,2); }
```

- Tampoco se introduce la indicación de si han sido declarados como variables (VAR) o constantes (LET).

De la misma manera que se separó la asignación de identificadores de tipo float con respecto a las otras variables, haremos de nuevo separación entre los identificadores de tipo let y var, teniendo cada uno un tipo diferente de identificador en la lista auxiliar usada para almacenar todas los identificadores que usamos.

```
declarations            :       declarations VAR identifier_list_var PYC { printf("declarations
-> declarations var identifier_list_var ;\n"); }
                        |       declarations LET identifier_list_let PYC { printf("declarations
-> declarations let identifier_list_let ;\n"); }

identifier_list_var     :       asig_var { printf("identifier_list_var -> asig\n"); }
                        |       identifier_list_var COMMA asig_var { printf("identifier_list_var
-> identifier_list_boolean , asig_float\n"); }
                        ;

asig_var        :       ID { printf("asig_var -> ID\n"); insertarVar(&lVar,$1,0,0); }
                        |       ID ASIGN expression { printf("asig_var -> ID =
expression(%s)\n",tipos[0]); insertarVar(&lVar,$1,$3,0); }
                        ;

identifier_list_let     :       asig_let { printf("identifier_list_let -> asig\n"); }
                        |       identifier_list_let COMMA asig_let { printf("identifier_list_let
-> identifier_list_boolean , asig_float\n"); }
                        ;

asig_let        :       ID { printf("asig_let -> ID\n"); insertarVar(&lVar,$1,0,1); }
                        |       ID ASIGN expression { printf("asig_let -> ID =
expression(%s)\n",tipos[0]); insertarVar(&lVar,$1,$3,1); }
```

- En las sentencias de asignación, no se comprueba si el identificador ha sido declarado y es variable.
- En las sentencias de read, no se comprueba si el identificador ha sido declarado y es variable.

En ambas realizamos una consulta a la lista, de manera que podamos saber si el identificador está declarado. En caso de no estar declarado, se produce un error.

En caso de estar declarado, pero ser de tipo let, no es posible realizar tanto la asignación como el read, por lo que también indicaría un error, esta vez, de que el identificador es de tipo let.

```
statement       :       ID ASIGN expression PYC {
                                if (consultarVar(lVar,$1) == -1) {
                                        printf("statement -> ID(%s) = expression | ERROR -> ID
                                        NO DECLARADO EN LINEA %d\n",$1,yylineno);
                                } else if (consultarVar(lVar,$1) == 1) {
                                        printf("statement -> ID(%s) = expression | ERROR -> ID
                                        NO VARIABLE EN LINEA %d\n",$1,yylineno);
                                } else {
                                       printf("statement -> ID(%s) = expression\n",$1); }
                                }

read_list       :       ID {
                                if (consultarVar(lVar,$1) == -1) {
                                        printf("read_list -> ID(%s) | ERROR -> ID NO DECLARADO
                                        EN LINEA %d\n",$1,yylineno);
                                } else if (consultarVar(lVar,$1) == 1) {
                                        printf("read_list -> ID(%s) | ERROR -> ID NO VARIABLE
                                        EN LINEA %d\n",$1,yylineno);
                                } else {
                                        printf("read_list -> ID(%s)\n",$1); }
                                }
                |       read_list COMMA ID {
                                if (consultarVar(lVar,$3) == -1) {
                                        printf("read_list -> read_list , ID(%s) | ERROR -> ID
                                        NO DECLARADO EN LINEA %d\n",$3,yylineno);
                                } else if (consultarVar(lVar,$3) == 1) {
                                        printf("read_list -> read_list , ID(%s) | ERROR -> ID
                                        NO VARIABLE EN LINEA %d\n",$3,yylineno);
                                } else {
                                        printf("read_list -> read_list , ID(%s)\n",$3); }
                                }
                ;
```

- En las expresiones no se comprueba si el identificador ha sido declarado.

Más simple que el explicado anteriormente, ya que solamente nos interesa saber si el identificador está declarado. Un identificador no declarado en la expresión nos producirá un error de identificador no inicializado.

```
expression      |       ID {
                                $$ = consultarVar(lVar,$1);
                                if ($$ == -1) {
                                        $$ = 4;
                                        printf("expr(%s) -> ID(%s) | ERROR -> ID NO DECLARADO
                                        EN LINEA %d\n",tipos[$$],$1,yylineno);
                                } else {
                                        printf("expr(%s) -> ID(%s)\n",tipos[$$],$1); }
                                }
```

Ampliaciones:

-Operadores de comparación y lógicos. Lo ideal hubiese sido que los operadores AND y OR no requiriesen paréntesis. Habría que definir la precedencia y asociatividad de los dos operadores.

Se ha redefinido para que los operadores AND y OR no requieran paréntesis. Se ha definido la precedencia y asociatividad de los dos operadores gracias a

```
%left AND OR
```

Quedando las nuevas reglas:

```
expression_comparacion  |       expression_comparacion AND expression_comparacion {
                                        if ($1 != $3) {
                                                $$ = 4;
                                        } else { $$ = $1; }
                                                 printf("expr_comp(%s) -> expr_comp(%s) &&
                                                 expr_comp(%s)\n",tipos[$$],tipos[$1],tipos[$3])
                                                 ; }
                        |       expression_comparacion OR expression_comparacion {
                                        if ($1 != $3) {
                                                $$ = 4;
                                        } else { $$ = $1; }
                                                 printf("expr_comp(%s) -> expr_comp(%s) ||
                                                 expr_comp(%s)\n",tipos[$$],tipos[$1],tipos[$3])
                                                 ; }
```

En cuanto al análisis léxico, se han corregido dos apartados que no funcionaban bien.

Por un lado teníamos que las cadenas sin cerrar al final del fichero no las reconocía porque, al parecer, antes de llegar al EOF también reconocía un retorno de línea por lo que siempre mostraba el tratamiento de una cadena en línea sin cerrar, tanto si estaba en mitad del fichero como al final. Para solucionarlo, nos creamos una condición de contexto auxiliar para activarla cuando encuentre el "\n" y una vez ahí comprobará si hay cualquier carácter o EOF. Dependiendo de lo que haya volveremos al estado inicial o retornaremos 0 para indicar que no hay más por analizar. La corrección quedaría así:

```
\"                              { BEGIN(cadena); yymore(); }
<cadena>\"                      { BEGIN(INITIAL); yylval.str = strdup(yytext); return STRING; }
<cadena>[^"\n]+                 { yymore(); }
<cadena>"\n"                    { BEGIN(aux); }

<aux>.                          { BEGIN(INITIAL); printf("Cadena abierta en línea %d sin cerrar\n", yylineno); yyless(0); }
<aux><<EOF>>                    { printf("Cadena abierta al final del documento sin cerrar\n");  return 0; }
```

Por otro lado teníamos la situación en las que nos aparece el operador “|” después del “&” y viceversa. Ahora con la mejora puede aparecer uno detrás del otro y mostrar el error léxico de los dos, mientras que antes no podía ya que la macro “ok” que compartían no contenía ninguno de los dos elementos. Ahora hemos definido dos macros “okand” y “okor” en la que a la del “&” le añadimos el “|” y viceversa. Se quedaría así:

```
okand        [a-zA-Z_0-9\";,+\-*/=(){}<>!| \t\n\r]
okor         [a-zA-Z_0-9\";,+\-*/=(){}<>!& \t\n\r]
```

```
"&"                             { BEGIN(andlogico); yymore(); }
<andlogico>"&"                  { BEGIN(INITIAL); return AND; }
<andlogico>{okand}+             { BEGIN(INITIAL); printf("Error léxico (línea %d): &\n",yylineno); yyless(1); }


"|"                             { BEGIN(orlogico); yymore(); }
<orlogico>"|"                   { BEGIN(INITIAL); return OR; }
<orlogico>{okor}+               { BEGIN(INITIAL); printf("Error léxico (línea %d): |\n",yylineno); yyless(1); }
```