UNIVERSIDAD
DE MURCIA

COMPILADORES
CURSO 2023/24

PROYECTO DE PRÁCTICAS

MINIC
DISEÑO DE UN COMPILADOR

ANTONIO LÓPEZ TOBOSO
54635347C - Grupo 2.2

MERCEDES LÓPEZ CABALLERO
49597021M - Grupo 2.2

PROF.:
MARÍA ANTONIA
CÁRDENAS VIEDMA

CONVOCATORIA
DE JUNIO

I. ÍNDICE

II. INTRODUCCIÓN Y ESPECIFICACIÓN

Este proyecto, desarrollado para la asignatura *Compiladores* perteneciente al *Grado en Ingeniería Informática* de la *Universidad de Murcia,* ahonda en una disciplina esencial en nuestro campo de estudio y desarrollo, pues es la base sobre la que se edifican los diversos lenguajes de programación y su posterior traducción a código maquina. Esto se lleva a cabo gracias a la *traducción e interpretación*, *análisis léxico, sintáctico y semántico.* Todo ello desarrollado bajo un exhaustivo análisis y diseño fundado en las necesidades del lenguaje y en las limitaciones teóricas y practicas que presentan las gramáticas en esta ciencia.

A. Presentación de la practica

Para demostrar los conocimientos adquiridos durante el espacio temporal de la asignatura, y adaptándonos a la especificación de la parte práctica de esta, hemos diseñado e implementado un compilador para un lenguaje de programación básico llamado *'miniC'*, el cual se asemeja al lenguaje *'C',* presentando una sintaxis y unas funciones muy simplificadas.

Para llevar a cabo la implementación nos hemos ayudado de las herramientas *'lex/flex'*, dedicada al análisis sintáctico; *'bison'*, dedicada al análisis sintáctico; y *'C'*, base sobre la que hemos fundado este compilador. Ademas, la salida que producirá nuestro compilador usara la estructura '*Mips Assembly'* , lenguaje ensamblador diseñado para *'Mips architecture'.*

B. Símbolos terminales

La gramática sobre la que se construye nuestro compilador consta de diversos símbolos terminales, los cuales vamos a enumerar a continuación.

Por un lado encontramos los números enteros (token `NUMBER`), comprendidos entre -2^{31} y 2^{31}; así como las cadenas de texto (token `STRING`), reconocibles al estar rodeadas por comillas dobles..

Por otro lado, los identificadores (token `ID`) presentan una limitación en longitud, precisamente de tamaño máximo igual a 32 caracteres, y están formados por la combinación de caracteres y dígitos, no pudiendo comenzar por estos últimos. Ademas encontramos palabras reservadas para el lenguaje, como son: var (token `VAR`), const (token `CONST`), if (token `IF`), else (token `ELSE`), do (token `DO`), while (token `WHILE`), print (token `PRINT`), read (token `READ`).

Finalmente, presentamos los caracteres que nos permiten implementar la lógica de operaciones y puntuación: ";" (token `SEMICOLON`), "," (token `COMMA`), "+" (token `PLUS`), "-" (token `MINUS`), "*" (token `TIMES`), "/" (token `DIV`), "=" (token `EQUAL`), "(" (token `LPAR`), ")" (token `RPAR`), "{" (token `LKEY`), "}" (token `RKEY`), ">" (token `GREATER`), ">=" (token `GEQUAL`), "<" (token `LESS`), "<=" (token `LEQUAL`), "==" (token `EEQUAL`), "!=" (token `NEQUAL`).

C. Especificación de la gramática

program	→	id ( ) { declarations statement_list }
declarations	→	declarations *var* identifier_list ;
	|	declarations *const* identifier_list ;
	|	λ
identifier_list	→	identifier
	|	identifier_list , identifier
identifier	→	id
	|	id = expression
statement_list	→	statement_list statement
	|	λ
statement	→	id = expression ;
	|	{ statement list }
	|	*if* (expression) statement else statement
	|	*if* (expression) statement
	|	*do* statement while (expression)
	|	*while* (expression) statement
	|	*print* (print_list) ;
	|	*read* (read_list) ;
print_list	→	print_item
	|	print_list , print_item
print_item	→	expression
	|	string
read_list	→	id
	|	read_list , id
expression	→	expression + expression
	|	expression - expression
	|	expression * expression
	|	expression / expression
	|	expression > expression
	|	expression >= expression
	|	expression < expression
	|	expression <= expression
	|	expression == expression
	|	expression != expression
	|	- expression
	|	(expression)
	|	id
	|	num

Esta gramática se encuentra escrita en notación BNF y representa fielmente la especificación sobre la que hemos construido el analizador sintáctico de nuestro nuestro compilador.

III. ESTRUCTURA Y USO

A. Estructura del código fuente

A la hora de diseñar la estructura del proyecto nos hemos decantado por una organización por módulos, proveyendo así de una estructura clara y ordenada, tal como se estima para proyectos grandes, que empiezan contener una cantidad cierta de ficheros.

Es por esto que dentro del directorio raíz *src* encontramos los siguiente directorios y ficheros:

- */Lexical*
  - *miniCLexical.l*
  - *lex.yy.x* *{generado por Makefile}*
- */Syntactic*
  - *miniCSyntactic.y*
  - *miniCSyntactic.tab.h* *{generado por Makefile}*
  - *miniCSyntactic.tab.c* *{generado por Makefile}*
  - *miniCSyntactic.output* *{generado por Makefile}*
- */Semantic*
  - *miniCCodeList.h*
  - *miniCCodeList.c*
  - *miniCSymbolTable.h*
  - *miniCSymbolTable.c*

- *miniCCompilerMain.c*
- *miniCCompiler* *{binario generado por Makefile}*
- *MakeFile*

B. Compilación y uso

Para ejecutar las directivas de compilación necesarias para ejecutar el proyecto se hará uso de la herramienta *make*. En la raíz del proyecto se ejecuta el comando *make*, el cual automáticamente llevara a cabo todas las acciones necesarias, creando finalmente el fichero *miniCCompiler*, que se corresponde con nuestro ejecutable, es decir, nuestro compilador de miniC.

Tras esto podemos, ejecutar el comando *make run*, el cual compilara el fichero *testFile.mc* que se encuentra en el directorio raíz, o bien usar el binario *miniCCompiler* de la siguiente forma:

./miniCCompiler ficheroEntrada.mc > ficheroCompilado.s

Obteniendo finalmente en *ficheroCompilado*, el cual será generado en el directorio de trabajo, el fichero compilado listo para usar en *Mars o Spim.*

IV. DISEÑO E IMPLEMENTACIÓN

En este apartado se explica la función de los distintos modelos que componen el proyecto y ciertas características de estos. Ademas se introducen las mejoras implementadas, seguidas de una explicación del proceso llevado a cabo para ello.

A. Análisis Léxico

Este módulo, el fichero *miniCLexical.l* se encarga de reconocer y clasificar los distintos tokens que componen el código fuente del programa de entrada, para así poder ser usados en el analizador sintáctico.

Este reconoce tokens tales como identificadores, palabras reservadas del lenguaje, números enteros, cadenas de texto, operadores, símbolos de prioridad, comentarios…

Además incluye límites para la definición de identificadores, los cuales no pueden superar los 32 caracteres, caso ante el que notifica con un error; limites en el rango de los números enteros, lo cuales deben encontrarse entre -2^{31} y 2^{31}, caso en el que también notifica del error.

Finalmente incluimos una recuperación de errores en modo pánico que actúa en caso de que algún token de la entrada no pueda reconocerse dentro del léxico del lenguaje, especificando en que linea se encuentra el fallo.

Las funciones que hemos implementado para apoyar la función del análisis léxico son las siguientes:

Caso:
Identificador superior a 32 caracteres
	`int in_range(){`
	`if( atoll(yytext) > MAX_INTEGER ) {`
	`fprintf(stderr, "Linea %d: Error: El numero ( %s ) excede el tamaño permitido. \n", yylineno, yytext); lexicalErr++;`
	`}`
	`return NUMBER;`
	`}`

Caso: Identificador superior a 32 caracteres	`({letra}|_)({letra}|{digito}|_)* {fprintf(stderr, "Linea %d: Error: El identificador (%s) excede el tamaño permitido. \n", yylineno, yytext); yylval.str = strdup(yytext); return ID; lexicalErr++; };`
Caso: Recuperación de errores en modo pánico	`{panico}` `{fprintf(stderr, "Linea %d: Error: Caracter (%s) no permitido.\n",yylineno, yytext); lexicalErr++; }`

Estas son las comprobaciones de error y el caso en el que actúa la recuperación de errores en modo pánico.

B. Análisis Sintáctico

El modulo dedicado al análisis sintáctico se encarga de comprobar que el programa de entrada es correcto desde un punto de vista sintáctico. Esto se lleva a cabo mediante el uso de la herramienta '*bison*'.

Para llevar a cabo todo esto, definiremos tantas reglas de producción como sean necesarias para desarrollar la gramática del apartado II.C. El fichero resultante de esto es *miniCSyntactic.y*.

Para que la gramática cumpla con la especificación, se ha establecido la precedencia y asociación de los distintos operadores, como los relativos a la suma, resta, multiplicación, división, menos unario… Esto se ha llevado a cabo mediante la definición de precedencias mediante la regla '*%left …*'.

Definición: tokens

```
%token VAR CONST IF ELSE WHILE PRINT READ DO
%token GREATER GEQUAL LESS LEQUAL EEQUAL NEQUAL
%token SEMICOLON COMMA PLUSOP MINUSOP TIMES
%token DIV EQUALS LPAR RPAR LKEY RKEY
%token <str> STRING ID NUMBER
```

Definición: precedencias y asociatividad

```
%nonassoc GREATER GEQUAL LESS LEQUAL EEQUAL NEQUAL
%left PLUSOP MINUSOP
%left DIV TIMES
%left UMINUS
```

En cuanto a esta gramática, produce un conflicto desplazamiento/reducción respecto de las reglas de producción de *if* y *if-else*. Por suerte, '*bison*' desplaza por defecto, que es lo que nosotros necesitamos para resolver el conflicto en este caso. Aun así para evitar warnings en tiempo de compilación se ha añadido la regla '*%expect 1*' en la cabecera del fichero previamente mencionado.

Comando: %expect

```
%expect 1
```

Ademas hemos implementado una recuperación de errores en modo pánico de la parte sintáctica, la cual comentamos más tarde en el apartado de mejoras.

Estas es la función que hemos redefinido para el análisis sintáctico:

Funcion: yyerror()

```
void yyerror(const char *str){
        syntacticErr++;
        fprintf(stderr, "Error sintáctico: (línea %d): %s \n", yylineno, str);
}
```

C. Análisis Semántico

El análisis semántico se compone de diversas partes, y nos ayuda a gestionar la declaración y uso de las distintas variables y constantes del programa de entrada.

Para implementarlo nos hemos apoyado en el uso de una tabla de símbolos, la cual usaremos para registrar la definición de variables, constantes y cadenas de texto. Gracias a esta, en el momento de realizar el análisis sintáctico podremos registrar para más tarde comprobar si se llevan a cabo casos como los enumerados a continuación.

1. Uso de variables y constantes sin definir.
2. Redefinición de variables y constantes.
3. Reasignación a constantes.

En caso de experimentar uno de los episodios mencionados con anterioridad, se emitirá un mensaje de error, especificando el problema concreto que lo ha generado.

A continuación mostramos algún ejemplo del código implementado para llevar a cabo el análisis semántico, seguido de las funciones implementadas para gestionar la tabla de símbolos usada.

Caso:
identifier : ID

```
…
if (!perteneceTS(symbolTable, $1)) añadeEntrada(symbolTable, $1,
        symbolType); else { fprintf(stderr, "Error en linea %d:
        Variable %s ya declarada\n", yylineno, $1); semanticErr++;
…
```

Caso:
statement. :
ID EQUALS expression
SEMICOLON

```
if (!perteneceTS(symbolTable, $1)) { fprintf(stderr, "Error en linea
        %d: Variable %s no declarada\n", yylineno, $1); semanticErr
        +; } else { if (esConstante(symbolTable,$1))
        { fprintf(stderr, "Error en linea %d: %s es constante\n",
        yylineno, $1); semanticErr++; }
```

En cuanto a las funciones de las que hablábamos con anterioridad, puesto que hemos utilizado la tabla de símbolos proporcionada por el profesorado de la asignatura, solo hemos requerido de la declaración de las siguientes funciones, cuyo código se puede consultar en *miniCSymbolTable.c* y *miniCSyntactic.y*. A la derecha de puede observar la estructura de un nodo de la tabla de símbolos.

1. void imprimirTablaS(Lista lista);
2. int perteneceTS(Lista lista, char * simbolo);
3. int esConstante(Lista lista, char * simbolo);
4. void añadeEntrada(Lista lista, char * simbolo, Tipo tipo);

Estructura:
Nodo
Simbolo

```
typedef struct Nodo {
  char *nombre;
  Tipo tipo;
  int valor;
} Simbolo;
```

D. Generación de código

Para llevar a cabo la generación de código, nos hemos basado el la lista de código proporcionada por el profesorado de la asignatura, *miniCCodeList.c y miniCCodeList.h.*

Para rellenar esta lista, la cual usaremos mas tarde para imprimir el código ensamblador final, añadimos un apéndice a cada regla de producción pertinente, por el cual se genera o reutiliza un nodo *Operación. T*odos estos nodos se van concatenando, o en su defecto, liberando, por lo que se forma una lista en la que almacenamos todas las lineas de código en lenguaje ensamblador que volcaremos para la generación en el fichero de salida, mediante el uso de la función *imprimirCodigo()*.

Estructura:
Operacion
	`typedef struct {`
	`  char * op;`
	`  char * res;`
	`  char * arg1;`
	`  char * arg2;`
	`} Operacion;`

Estructura:
PosicionListaCRep
 	
(nodo de la lista)
	`struct PosicionListaCRep {`
	`  Operacion dato;`
	`  struct PosicionListaCRep *sig;`
	`};`

Estructura:
ListaCRep
	`struct ListaCRep {`
	`  PosicionListaC cabecera;`
	`  PosicionListaC ultimo;`
	`  int n;`
	`  char *res;`
	`};`

Estas son las definiciones del nodo operación y de las estructuras que conforman las lista de código.

Gracias al uso de las previas estructuras de datos y de las funciones que presentamos continuación se lleva a cabo prácticamente la totalidad de la generación de código. Ademas, solo generamos código en caso de que no surge ningún tipo de error léxico, sintáctico o semántico, pues llevamos un control de ellos mediante contadores

```
char * obtenerReg(){

        for (int i = 0; i < MAX_REGISTERS; i++) {
                        if (registers[i] == 0) {
                                registers[i] = 1;

                                char* resultado = (char*) malloc(4 * sizeof(char));

                                snprintf(resultado, 4, "$t%d", i);
                                return resultado;
                        }
        }
        semanticErr++;
        fprintf(stderr, "No es posible compilar el programa. Registros insuficientes.
        \n");
```

```
void liberarReg(char * registro){

        char * numeroRegistro = registro + 2;
        int indice = atoi(numeroRegistro);

        if (indice ⩾ 0 && indice < MAX_REGISTERS) {
                        registers[indice] = 0;
        } else {
                fprintf(stderr, "Índice fuera de rango.\n");
        }

}

void imprimirCodigo(ListaC codigo) {

        printf("##################\n");
        printf("# Seccion de codigo\n");
        printf("\t.text\n");
        printf("\t.globl main\n");
        printf("main:\n");

        PosicionListaC p = inicioLC(codigo);
        while (p ≠ finalLC(codigo)) {
                Operacion oper = recuperaLC(codigo,p);
                char labelCheck[7];
                strncpy(labelCheck, oper.op, 6);
                if (strcmp(labelCheck, "$label")){
                        printf("\t");
                }
                else{
                        printf("\n");
                }
                printf("%s",oper.op);
                if (oper.res) printf(" %s",oper.res);
                if (oper.arg1) printf(",%s",oper.arg1);
                if (oper.arg2) printf(",%s",oper.arg2);
                printf("\n");
                p = siguienteLC(codigo,p);
        }

        printf("\n##################\n");
        printf("# Fin de la ejecucion\n");

        printf("\tli $v0, 10\n");
        printf("\tsyscall\n");
}


char * concatenaStr(char * str0, char * str1){

        char * string;
        asprintf(&string, "%s%s", str0, str1);
        return string;
}



char * newLabel(){
        char * label;
        asprintf(&label, "$label_%d",labelCount++);
        return label;
}
```

E. Mejoras implementadas

A modo de ampliación, hemos implementado las siguientes mejoras disponibles a nuestro compilador:

1. Tratamiento de errores sintácticos

Para llevar implementar el tratamiento de errores sintácticos y su pertinente recuperación de errores, nos hemos ayudado del uso del token error ofrecido por la herramienta *`bison'*.

Mediante el uso de este token, hemos definido nuevas reglas de producción muy similares a las ya existentes, pero ligeramente modificadas incluyendo el token error, o en su defecto hemos creado nuevas reglas desde cero, incluyendo en ellas únicamente el token error.

Estas son las reglas introducidas para llevar a cabo el tratamiento y recuperación de las que hablábamos con anterioridad. Han sido elegidas tras un estudio basado en los potenciales fallos de un programador al escribir código, buscando conocer cuales son los errores mas comunes, y cuales de ellos podíamos manejar sin tener que modificar gran parte de la gramática.

Caso:
declarations :
declarations CONST error SEMICOLON
	`$$ = creaLC();`
	`guardaResLC($$, "" );`
	`fprintf(stderr, "Linea %d: Error en la declaración de una constante\n", yylineno);`
	`syntacticErr++;`

Caso:
declarations :
declarations VAR error SEMICOLON
	`$$ = creaLC();`
	`guardaResLC($$, "" );`
	`fprintf(stderr, "Linea %d: Error en la declaración de una variable\n", yylineno);`
	`syntacticErr++;`

Caso:
statement : error
	`$$ = creaLC();`
	`guardaResLC($$, "" );`
	`fprintf(stderr, "Linea %d: Error en una sentencia\n", yylineno);`
	`syntacticErr++;`

Caso:
expression : error
	`$$ = creaLC();`
	`guardaResLC($$, "" );`
	`fprintf(stderr, "Linea %d: Error en una expresión aritmética\n", yylineno);`
	`syntacticErr++;`

En caso de alcanzar una de estas reglas, será porque se ha producido un error sintáctico, pero gracias a la inclusión de estas, el análisis del código continuara, permitiendo analizar la gran parte del código restante.

2. Sentencia *do-while*

Siendo esta muy similar a la sentencia while, para implementarla hemos modificado los ficheros *miniCLexical.l* y *miniCSyntactic.y*.

En el primero, hemos introducido una nueva "palabra reservada": *do*. No hemos tenido la necesidad de añadir el while ya que la sentencia *while* ya estaba previamente declarada.

En el caso del fichero relacionado principalmente con el análisis sintáctico y al igual que en el fichero anterior, hemos añadido el nuevo token *do*, en su correspondiente "lugar" teniendo en cuenta la precedencia que este conlleva. Hemos, además, añadido su regla de producción, la cual queda detallada a continuación a modo de comparación con la regla de producción de la sentencia *while*. Como queda reflejado en la tabla siguiente mediante el uso del color rojo para las diferencias entre ambas sentencias, las modificaciones implementadas en la regla *while* para así conseguir la regla *do-while*, consisten en 2 cambios principales, es decir, es mínima la diferencia entre ellas.

```
WHILE LPAR expression RPAR statement

{ $$ = creaLC();
char * label_WHILE = newLabel();
char * label_ENDW = newLabel();

Operacion oper;

oper.op = concatenaStr(label_WHILE, ":");
oper.res = NULL;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);

concatenaLC($$, $3);

oper.op = "beqz";
oper.res = recuperaResLC($3);
oper.arg1 = label_ENDW;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);
liberarReg(oper.res);

concatenaLC($$, $5);

oper.op = "b";
oper.res = label_WHILE;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);

oper.op = concatenaStr(label_ENDW, ":");
oper.res = NULL;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);
```

```
DO statement WHILE LPAR expression RPAR

{ $$ = creaLC();
char * label_DOWHILE = newLabel();
char * label_ENDDW = newLabel();

Operacion oper;

oper.op = concatenaStr(label_DOWHILE, ":");
oper.res = NULL;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);

concatenaLC($$, $2);
concatenaLC($$, $5);

oper.op = "beqz";
oper.res = recuperaResLC($5);
oper.arg1 = label_ENDDW;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);
liberarReg(oper.res);

oper.op = "b";
oper.res = label_DOWHILE;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);

oper.op = concatenaStr(label_ENDDW, ":");
oper.res = NULL;
oper.arg1 = NULL;
oper.arg2 = NULL;
insertaLC($$, finalLC($$), oper);
```

a. El primero de ellos es el cambio de la etiqueta, obviamente, pues se trata de una nueva sentencia, por lo que no podrían llamarse igual. Esto afecta tanto a la etiqueta de declaración (*label_DOWHILE*) como la de fin (*label_ENDDW*).
b. La mayor diferencia entre la sentencia while y la do-while es el orden de la expresión y la sentencia que utiliza. Por ello, el segundo cambio notable es a la hora de concatenar con la lista de código creada. Mientras que en while se concatena la expresión, se comprueba y entonces se concatena la sentencia, en do-while se concatenan ambas directamente, pues siempre va a ejecutar la sentencia sin necesidad de comprobar si se cumple o no lo que dicta la expresión.

3. Operadores relacionales para las condiciones de las sentencias

A la hora de implementar los operadores relacionales, hemos, al igual que en la sentencia do-while, modificado tanto el fichero *miniCLexical.l* como el *miniCSyntactic.y*. Del mismo modo, hemos añadido al primer fichero las palabras reservadas correspondientes a cada uno de los operadores:

> : GREATER	<= : LEQUAL
>= : GEQUAL	== : EEQUAL
< : LESS	!= : NEQUAL

Entonces, hemos añadido dichas "palabras" en forma de token en el fichero *miniCSyntactic.y* guardando el orden de precedencia y haciendo que no sean asociativos entre ellos mediante el comando *%nonassoc*. Esto indica que no se podría llevar a cabo la siguiente expresión, por ejemplo:

$$12 < x > 4$$

Para definir las reglas de producción referentes a los operadores relacionales, lo hemos hecho en el apartado "expresiones". Estas siguen en cierta manera el esquema del resto de expresiones, como las sumas, restas, multiplicaciones… Estas se tratan como una especie de booleanos, es decir, para hacer las comparaciones, obtenemos como resultado un 0 o un 1. Ponemos como ejemplo la expresión *less* (menor que), ya que el resto son prácticamente iguales salvo mínimos cambios junto con el cambio obvio de operación.

```
expression LESS expression
{ $$ = $1;
concatenaLC($$, $3);
Operacion oper;
oper.op = "slt";
oper.res = recuperaResLC($1);
oper.arg1 = recuperaResLC($1);
oper.arg2 = recuperaResLC($3);
insertaLC($$, finalLC($$), oper);
guardaResLC($$, oper.res);
liberaLC($3);
liberarReg(oper.arg2);}
```

Lo que hace en esta expresión es, concatenar ambas expresiones (*$1* y *$3*) y las compara haciendo un "*slt*", es decir, "*set less than*". Lo que hace esta operación es poner un 1 si *$1* es menor que *$3* (1 == *true*) o un 0 en caso contrario (0 == *false*).

V. VALIDACIÓN DEL PROYECTO

A. Validación Léxica

Para verificar la función del analizador léxico, hemos desarrollado este caso de prueba, del cual mostramos la salida de su compilación a la derecha de la tabla

```
/*prueba lexico*/
lexico(){
const c = 0; //bien
const a = 214748364802; //numero fuera de rango
var {b} = 80; //caracteres no permitidos
var ? = 45;
var d = c+33; //bien
var qwertyuioplkjhgfdsazxcvbnmnbvcxzas = 32; //id fuera de rango
}
```

```
./miniCCompiler pruebaLexico.mc > pruebaLexico.s
Linea 6: Error: El numero ( 214748364802 ) excede el
        tama?o permitido.
Error sintáctico: (línea 8): syntax error
Linea 8: Error en la declaración de una variable
Linea 10: Error: Caracter (?) no permitido.
Linea 10: Error en la declaración de una variable
Linea 16: Error: El identificador
        (qwertyuioplkjhgfdsazxcvbnmnbvcxzas) excede el
        tama?o permitido.
```

B. Validación Sintáctica

Para verificar la función del analizador sintáctico, hemos desarrollado este caso de prueba, del cual mostramos la salida de su compilación a la derecha de la tabla

```
/*prueba sintactico*/
sintactico(){
print ("Inicio del programa\n”);
const a 23; //falta el igual
var b =; //falta el valor
const c = 0;
const = 75;
if ( c < 7 {   //falta cerrar el “)”
        print ("Hola");
   }
print ( 4 < );
}
```

```
./miniCCompiler pruebaSintactico.mc > pruebaSintactico.s
Error sintáctico: (línea 7): syntax error
Linea 7: Error en una sentencia
Linea 9: Error en una sentencia
Linea 9: Error en una expresión aritmética
Error en linea 9: Variable b no declarada
Linea 11: Error en una sentencia
Error en linea 11: Variable c no declarada
Error sintáctico: (línea 13): syntax error
Linea 13: Error en una sentencia
Error en linea 15: Variable c no declarada
Error sintáctico: (línea 15): syntax error
Linea 15: Error en una expresión aritmética
Linea 16: Error en una expresión aritmética
Linea 16: Error en una sentencia
```

C. Validación Semántica

Para verificar la función del analizador semántico, hemos desarrollado este caso de prueba, del cual mostramos la salida de su compilación a la derecha de la tabla

```
/*prueba semantico*/

semantico(){

var a;

var a = 33; //variable ya declarada

const c = 4;

b = 2; //no declarada

c = 2+a; //constante modificada

}
```

```
./miniCCompiler pruebaSemantico.mc > pruebaSemantico.s
Error en linea 5: Variable a ya declarada
Error en linea 7: Variable b no declarada
Error en linea 8: c es constante
```

D. Validación general

Para realizar una verificación general hemos desarrollado este caso de prueba, el cual debería compilar satisfactoriamente, sin errores, y en el que están presentes diversas funciones del lenguaje miniC.

Mostramos la salida de su compilación a la derecha de la tabla y en la siguiente pagina.

```
/*prueba general*/

prueba() {
const  a=0, b=0;
var c=5+2-2;
var d;
print ("Inicio del programa\n");
if (a)  print ("a","\n");
  else if (b) print ("No a y b\n");
       else while (c)
            {
                  print ("c = ",c,"\n");
                  c = c-2+1;
            }
do{
  print( "Este mensaje debería ser
impreso una única vez\n\n");
} while (a > 10)

read (d);

do{
  print ("d = ",d,"\n");
  d = d -1;
} while (d > 5)

print ("Final","\n");
}
```

```
##################
# Seccion de datos
        .data

$str1:
        .asciiz "Inicio del programa\n"
$str2:
        .asciiz "a"
$str3:
        .asciiz "\n"
$str4:
        .asciiz "No a y b\n"
$str5:
        .asciiz "c = "
$str6:
        .asciiz "\n"
$str7:
        .asciiz "Este mensaje debería
ser impreso una única vez\n\n"
$str8:
        .asciiz "d = "
$str9:
        .asciiz "\n"
$str10:
        .asciiz "Final"
$str11:
        .asciiz "\n"
_a:
        .word 0
_b:
        .word 0
_c:
        .word 0
_d:
        .word 0
```

```
##################
# Seccion de codigo
        .text
        .globl main
main:
        li $t0,0
        sw $t0,_a
        li $t0,0
        sw $t0,_b
        li $t0,5
        li $t1,2
        add $t0,$t0,$t1
        li $t1,2
        sub $t0,$t0,$t1
        sw $t0,_c
        li $v0,4
        la $a0,$str1
        syscall
        lw $t0,_a
        beqz $t0,$label_4
        li $v0,4
        la $a0,$str2
        syscall
        li $v0,4
        la $a0,$str3
        syscall
        b $label_5

$label_4:
        lw $t1,_b
        beqz $t1,$label_2
        li $v0,4
        la $a0,$str4
        syscall
        b $label_3

$label_2:

$label_0:
        lw $t2,_c
        beqz $t2,$label_1
        li $v0,4
        la $a0,$str5
        syscall
        lw $t3,_c
        li $v0,1
        move $a0,$t3
        syscall
        li $v0,4
        la $a0,$str6
        syscall
        lw $t3,_c
        li $t4,2
        sub $t3,$t3,$t4
        li $t4,1
        add $t3,$t3,$t4

$label_1:

$label_3:

$label_5:

$label_6:
        li $v0,4
        la $a0,$str7
        syscall
        lw $t0,_a
        li $t1,10
        slt $t0,$t1,$t0
        beqz $t0,$label_7
        b $label_6

$label_7:
        li $v0,5
        syscall
        sw $v0,_d

$label_8:
        li $v0,4
        la $a0,$str8
        syscall
        lw $t0,_d
        li $v0,1
        move $a0,$t0
        syscall
        li $v0,4
        la $a0,$str9
        syscall
        lw $t0,_d
        li $t1,1
        sub $t0,$t0,$t1
        sw $t0,_d
        lw $t0,_d
        li $t1,5
        slt $t0,$t1,$t0
        beqz $t0,$label_9
        b $label_8

$label_9:
        li $v0,4
        la $a0,$str10
        syscall
        li $v0,4
        la $a0,$str11
        syscall

##################
# Fin de la ejecucion
        li $v0, 10
        syscall
```

Obtenemos tras la compilación con nuestro compilador de miniC el previo resultado, el cual una vez probado en el simulador *Mars* funciona de manera correcta y esperada.

VI. CONCLUSIÓN

La elaboración de esta práctica nos ha parecido un proceso no sólo entretenido, sino también muy enriquecedor para nuestra formación como futuros ingenieros informáticos. Nos ha ayudado a reforzar conceptos teóricos que, una vez vistos aplicados de manera práctica, han quedado mucho más claros.

Nos ha gustado mucho hacer este proyecto, principalmente por 2 motivos: el contenido y los recursos. En términos de contenido, nos ha parecido apasionante descubrir cómo funciona un compilador, cómo se "construye" desde cero, las funcionalidades que ofrece… Y a nivel de recursos, nos gustaría destacar los proporcionados por los profesores de la asignatura a través del aula virtual así como la propia ayuda recibida de forma personal por su parte. Esto lo notamos sobretodo en el hecho de que, aunque en algún momento de las prácticas nos hayamos quedado algo rezagados, se nos han proporcionado vídeos explicativos para poder recordar los conceptos explicados en clase, y nuestra profesora no ha tenido problema en solucionarnos dudas concretas en ningún momento.

A modo de conclusión, esta práctica nos ha parecido muy interesante, no se nos ha hecho "bola" ni "cuesta arriba" en ningún momento, es decir, siempre que tratábamos de avanzar, presentábamos animo y ganas por ello, no lo hacíamos como una obligación. Nos gustaría poder haberle dedicado más tiempo, no tanto a la elaboración de la práctica como tal, sino más a la redacción y cumplimiento de esta memoria, pero debido a la carga de trabajo que tenemos por la época de exámenes que se nos viene encima, no ha podido ser. Sin duda, si tuviéramos que volver hacia atrás y empezar de nuevo, le dedicaríamos un poco más tiempo y trataríamos de realizar todas las mejoras disponibles…, tratando que nuestro compilador fuese “perfecto”.