

Memoria Trabajo Compiladores



**UNIVERSIDAD
DE MURCIA**

**Jaime Cegarra Martínez
DNI: 49335068E**

Grupo: 2.1

Índice de la memoria

| | |
|--|----|
| 1. Introducción..... | 3 |
| 2. Léxico..... | 4 |
| 2.1 Definiciones..... | 4 |
| 2.2 Reglas..... | 4 |
| 2.2.1 Tokens..... | 4 |
| 2.2.2 Definición de reglas..... | 5 |
| 2.3 Subrutinas..... | 6 |
| 3. Sintaxis..... | 7 |
| 3.1 Definiciones..... | 7 |
| 3.2 Reglas..... | 8 |
| 3.2.1 Progam..... | 9 |
| 3.2.2 Declarations y tipo..... | 10 |
| 3.2.3 Lista de variables y constantes..... | 10 |
| 3.2.4 Sentencias..... | 12 |
| 3.2.4.1 Sentencia asignación..... | 13 |
| 3.2.4.2 Sentencia if..... | 14 |
| 3.2.4.3 Sentencia if else..... | 15 |
| 3.2.4.4 Sentencia while..... | 16 |
| 3.2.5 Prints..... | 17 |
| 3.2.6 Reads..... | 19 |
| 3.2.7 Expressions..... | 20 |
| 3.3 Subrutinas de apoyo en C..... | 22 |
| 3.3.1 Subrutinas de variables..... | 22 |
| 3.3.2 Subrutinas de impresión..... | 22 |
| 3.3.3 Subrutinas de operación..... | 23 |
| 3.3.4 Subrutinas de temporales y etiquetas..... | 24 |
| 4. Manual de usuario..... | 25 |
| 5. Ejemplos de entrada y salida..... | 26 |
| 5.1 Prueba aulavirtual “prueba.mc”..... | 26 |
| 5.2 Prueba junio..... | 27 |
| 5.3 Pruebas comentarios..... | 29 |
| 5.3.1 Prueba sin final..... | 29 |
| 5.3.2 Prueba con muchos comentarios pero mal..... | 29 |
| 5.3.3 Prueba con muchos comentarios pero bien..... | 30 |
| 5.4 Prueba de registros..... | 31 |
| 6. Conclusiones finales..... | 32 |

1. Introducción

El proyecto trata de la creación de un compilador para una versión reducida del lenguaje de C llamado miniC.

La estructura del proyecto es la siguiente:

- Léxico: se especifica en el fichero “lexico.l”, y analiza los caracteres, cadenas, identificadores, comprueba que es válida la entrada y si no lanza un error.
- Sintaxis: se especifica en el fichero “sintaxis.y”, valida las cadenas y se aplican las reglas de la gramática, finalmente se encuentran unas rutinas en C para facilitar el trabajo.
- Lista de Símbolos: es un fichero que facilita mucho las operaciones para manipular los datos en cuanto a la declaración y manipulación que se guarda en una estructura de datos, este fichero he decidido no manipularlo.
- Lista de Código: las instrucciones que debemos generar las tenemos que guardar en la estructura que nos brinda este fichero, junto a una serie de acciones y funciones para manipular dicha lista, este fichero tampoco lo he manipulado.
- Programa principal: es el fichero que contiene la función “main” de C, que básicamente comprueba los argumentos de llamada y llama al análisis con `yyparse()`.
- Makefile: es el fichero que mediante la orden `make` en la consola permite generar el programa llamado “minic”

2. Léxico

Un fichero de Flex se divide en tres partes principalmente: definiciones, reglas, subrutinas. En esta sección vamos a tratar cada una de las partes de este fichero.

2.1 Definiciones

Aquí he incluido las bibliotecas de “stdio.h” necesaria para hacer cosas básicas de C como un printf.

Luego “math.h”, que permite hacer operaciones matemáticas, esto lo uso porque en el enunciado de la práctica dice que un entero no puede ser más grande que 2^{31} .

A continuación he incluido la cabecera de la sintaxis generada por Bison, necesaria para el análisis.

Finalmente, la declaración de la subrutina que se lanza si hay un error léxico.



```
%{
#include <stdio.h>
#include "math.h"
#include "sintaxis.tab.h"
void error_lexico();
%}
```

2.2 Reglas

Este apartado se divide en dos partes, la definición de los tokens y la definición de las reglas:

2.2.1 Tokens

En los tokens tenemos, los dígitos del 0 al 9, los caracteres (mayúsculas y minúsculas).

La activación de la variable yylineno que es una variable de Flex que sigue el número de línea que se está analizando y es muy útil para notificar errores.

La declaración de un estado exclusivo, muy útil (y según la lógica que he seguido, necesario) para los comentarios y viene explicada por encima en el pdf de seminario de Flex

```

    ● ○ ●
digito      [0-9]
letra       [a-zA-Z]
%option yylineno
%x          COMENTARIO

```

2.2.2 Definición de reglas

En este apartado, para comenzar tenemos los comentarios de una sola línea, cuya lógica es que comienza con “//” y captura todo hasta un salto de línea o un retorno de carro.

Después, se ignora todo lo que tiene que ver con tabulaciones, saltos de línea, y retornos de carro.

A continuación viene lo más complicado del léxico, los comentarios, que podemos definir como un estado de un autómata que tiene diferentes maneras de comportarse: comienza con “*/”, ignora todo lo que no sean asteriscos y todo los asteriscos que no continúen con “/” o con “*”. Pero si encuentra uno o más “*” seguidos de “/”, termina el comentario.

Seguidamente, viene la declaración de las palabras reservadas y los caracteres especiales.

Luego el error léxico, que básicamente es todo carácter que no esté dentro de lo que hemos definido.

```

    ● ○ ●
/*//(.*)[\n\r]
[ \t\n\r]+;;

/* */
<COMENTARIO>:[*]* BEGIN COMENTARIO;
<COMENTARIO>[*]{/};; BEGIN 0;
<COMENTARIO>[*]+[^/*]*;;

/* PALABRAS RESERVADAS */
"var" return VAR;
"const" return CONST;
"int" return INT;
"if" return IF;
"else" return ELSE;
"while" return WHILE;
"print" return PRINT;
"read" return READ;

/* CARACTERES ESPECIALES */
";" return DOTCOMMA;
"," return COMMA;
"+-" return PLUSOP;
"-_" return MINUSOP;
"*_" return MULTOP;
"/_" return DIVOP;
"{" return LPAREN;
"}_" return RPAREN;
"{" return LBRACE;
"}_" return RBRACE;
":_" return QMARK;
";_" return COLON;
"==" return ASIGN;

/* ERRORES TODO LO QUE NO SEA + FUNCION */
([^\w\-\_]+\*\+|[^\\w\-\_]\*)+ error_lexico();

```

Finalmente, encontramos la definición de lo que es un entero, una cadena de caracteres y un identificador:

En el entero, comprobamos que el valor no sea mayor a 2^{31} si es así informaremos de un error.

En la cadena de caracteres comprobamos que comience con ‘“’ y termine con ‘“’, y se pueda escapar lo demás.

En el identificador comprobamos primero que comienza por letra o “_” y a continuación que la longitud introducida sea menor de 32 caracteres y si es así informamos de error.

```
/* VALORES LITERALES */
{digito}+
{
    if(atol(yytext)>pow(2,31))
        printf("Entero más grande que 2^31\n");
    else{
        yylval.lexema = strdup(yytext);
        return ENTERO;
    }
}
\"([^\n\\]|\\[ntr])*\"
```



```
/* IDENTIFICADOR */
({letra}|_)(_|{digito}|{letra})*
{
    if (yyleng > 32)
        printf("ID tiene más de 32 caracteres\n");
    else{
        yylval.lexema = strdup(yytext);
        return ID;
    }
}
```

2.3 Subrutinas

Aquí simplemente hay una función que imprime por pantalla que hay un error en la línea yylineno y el contenido de lo que da error.

```
void error_lexico(){
    printf("Error léxico en la línea %d: %s \n", yylineno, yytext);
}
```

3. Sintaxis

La sintaxis especificada por el enunciado del proyecto, y la tarea es pasarlo a bison, haciendo uso de las reglas que contienen las comprobaciones y la introducción en la tabla o lista de Símbolos y en la lista de código, este fichero tiene también una estructura como la de Flex con definiciones, reglas y subrutinas en C.

3.1 Definiciones

En este apartado definimos la inclusión primero de todas las librerías que necesitamos junto a los ficheros que necesitamos que serán los que definen y manipulan la tabla de símbolos y la de código.

Luego se definen la cadena de caracteres yytext y el numero de línea yylineno, que proviene de lexico.l.

A continuación se definen las variables globales que vamos a usar: un array con los temporales inicializados a 0, que indica cual se puede usar en cada momento, la tabla de símbolos para almacenarlos, el tipo de dato un contador de etiquetas para el nombre y el contador de cadenas para la impresión

Finalmente, la declaración de las funciones que vamos a usar.

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

extern char *yytext;
extern int yylineno;

#define N_TEMP 10
// las variables globales a usar
int tmp[N_TEMP] = {0};
Lista tablaSimb;
Tipo tipo;
int contEtiquetas = 0;
int contCadenas = 0;

// las funciones a usar
void inicializaTemporales();
void liberaTemporal(char * registro);
int nuevoTemp();
char* nuevaEtiq();

void imprimirTablaS();
void imprimeL(ListaC codigo);
bool perteneceTablaS(char *nombre);
void anadeEntrada(char *nombre, Tipo t, int v);
bool esConstante(char *nombre);
Operacion operSaltoCond(char *op, char *exp, char *etiq);
Operacion operEtiq(char *etiq);
Operacion crearOpArit(char *op, char *oper1, char *oper2, char tempStr[5]);
void siguienteRegistro(char *registro);

int yylex();
void yyerror(char *s);

%}
```

3.2 Reglas

Esta parte se divide también en la declaración de tokens, primero se declara que se necesita la lista de código para la ejecución como se menciona en el vídeo.

```
program      → id ( ) { declarations statement_list }
declarations → declarations var tipo var_list ;
               | declarations const tipo const_list ;
               | λ
tipo          → int
var_list      → id
               | var_list , id
const_list    → id = expression
               | const_list , id = expression
statement_list → statement_list statement
               | λ
statement     → id = expression ;
               | { statement_list }
               | if ( expression ) statement else statement
               | if ( expression ) statement
               | 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 )
               | id
               | num
```

Luego se declara el lexema y la lista de código, y los tokens que pertenecen al lexema, los que están asociados a código y el resto de tokens que no están asociados a ningún atributo y finalmente los que tienen precedencia para evitar ambigüedades.

```
%code requires{#include "listaCodigo.h"}
%union{char *lexema; ListaC codigo;}

%token <lexema> STRING ID ENTERO
%token VAR CONST INT IF ELSE WHILE PRINT READ DOTCOMMA COMMA PLUSOP MINUSOP MULTOP
DIVOP LPAREN RPAREN LBRACE RBRACE QMARK COLON ASIGN MINUSOP_UN
%type <codigo> expression statement_list print_list print_item read_list
declarations var_list const_list

%left PLUSOP MINUSOP MULTOP DIVOP ASIGN MINUSOP_UN
```

A continuación vamos a especificar, punto a punto lo que hace las reglas de la gramática.

3.2.1 Program

La regla de producción program primero crea la tabla de símbolos, luego detecta los terminales, a continuación la declaración, luego la lista de sentencias (ifs, whiles, operaciones, etc.) y lo que hace es crear una lista de código, concatena las listas de código que provienen de las declaraciones, luego las de las sentencias, las libera y después imprime primero la tabla de símbolos, luego la lista de código y finalmente libera las dos listas.

```
● ● ●

program      : {tablaSimb = creaLS();} ID LPAREN RPAREN LBRACE declarations statement_list RBRACE {    ListaC codigo = creaLC();
    concatenalC(codigo, $6);
    concatenalC(codigo, $7);
    liberaLC($6);
    liberaLC($7);
    imprimirTablas();
    imprimeLC(codigo);
    liberaLS(tablaSimb);
    liberaLC(codigo);
}
;
```

Las acciones de imprimir tabla de símbolos e imprimir lista de código son las siguientes:

```
● ● ●

void imprimeLC(ListaC codigo){
    printf(".text\n");
    printf(".globl main\n");
    printf("main:\n");
    PosicionListaC pos_actual = inicioLC(codigo);
    for (PosicionListaC pos = pos_actual; pos != finalLC(codigo); pos = siguienteLC(codigo, pos))
    {
        Operacion op = recuperalC(codigo, pos);

        if (op.op[0] != '$') {
            printf("\t");
        }
        printf("%s", op.op);

        if (op.res) {
            printf(" %s", op.res);
        }
        if (op.arg1) {
            printf(",%s", op.arg1);
        }
        if (op.arg2) {
            printf(",%s", op.arg2);
        }
        printf("\n");
    }
    printf("#####\n");
    printf("# Fin\n");
    printf("vli $v0, 10\n");
    printf("\tsyscall\n");
}
```

```
void imprimirTablaS(){
    printf(".data\n");
    PosicionLista pos_actual = inicioLS(tablaSimb);
    while (pos_actual != finalLS(tablaSimb))
    {
        Simbolo sim = recuperaLS(tablaSimb, pos_actual);
        if(sim.tipo != CADENA) printf("_%s:\n\t.word %d\n",sim.nombre);
        else printf("$str%d:\n\t.ascii \"%s\"\n",sim.valor, sim.nombre);
        pos_actual = siguienteLS(tablaSimb,pos_actual);
    }
}
```

3.2.2 Declarations y tipo

Esta regla, declara si las variables, las constantes o si no se declara nada y la regla tipo que es es entero que es el que tratamos en la gramática.

```
declarations: declarations VAR tipo var_list DOTCOMMA      {
                                $$ = $1;
                            }
                           | declarations CONST tipo const_list DOTCOMMA {
                                $$ = $1;
                                concatenaLC($$, //liberaLC($4);
                            }
                           | /* empty */
                                {
                                $$ = creaLC();
                                ;
                            }

tipo      : INT
;
```

3.2.3 Lista de variables y constantes

En esta regla hay que comprobar, en el caso de las variables que el ID no esté ya en la tabla, y en el caso de las constantes como se asignan con una expresión hay que comprobar que no está en la tabla, introducirlo y luego inicializarlo con la expresión que venga después. Y con las listas prácticamente igual

```

var_list : ID
{
    if(!perteneceTablaS($1))
        anadeEntrada($1, VARIABLE, 0);
    else {
        printf("Error semántico: Variable ya declarada '%s' (linea: %d)\n", $1, yylineno);
        YYERROR; // en semBison pone que hay que abortar ast
    }
}
| var_list COMMA ID
{
    if (!perteneceTablaS($3))
        anadeEntrada($3, VARIABLE, 0);
    else {
        printf("Error semántico: Variable ya declarada '%s' (linea: %d)\n", $1, yylineno);
        YYERROR; // en semBison pone que hay que abortar ast
    }
};

const_list : ID ASIGN expression
{
    if (!perteneceTablaS($1)){
        anadeEntrada($1, CONSTANTE, 0);
        ListaC codigo = $3;

        char nomVar[20];
        sprintf(nomVar, "_%s", $1);
        Operacion op = {
            .op = "sw",
            .res = recuperaResLC(codigo),
            .arg1 = strdup(nomVar),
            .arg2 = NULL
        };
        insertaLC(codigo, finalLC(codigo), op);
        inicializaTemporales();
        $$ = codigo;
    }
    else {
        printf("Error semántico: Variable ya declarada '%s' (linea: %d)\n", $1, yylineno);
        YYERROR; // en semBison pone que hay que abortar ast
    }
}
| const_list COMMA ID ASIGN expression {
    if (!perteneceTablaS($3)){
        anadeEntrada($3, CONSTANTE, 0);
        ListaC codigo = $1;

        concatenaLC(codigo, $5);

        char nombreVar[20];
        sprintf(nombreVar, "_%s", $3);
        Operacion op = {
            .op = "sw",
            .res = recuperaResLC($5),
            .arg1 = strdup(nombreVar),
            .arg2 = NULL,
        };
        insertaLC(codigo, finalLC(codigo), op);
        inicializaTemporales();
        $$ = codigo;
        liberaLC($5);
    }
    else {
        printf("Error semántico: Constante ya declarada '%s' (linea: %d)\n", $1,
               YYERROR; // Aborta el análisis sintáctico***"
    }
}
;

yylineno;

```

Para hacer estas operaciones me he apoyado en unas funciones que se mencionaron en el vídeo, para no tener que hacer manualmente cada comprobación:

“perteneceTablaS” es una función que usa buscaLS que si no lo encuentra devuelve el índice del final de la tabla, entonces lo podemos comparar con el índice del final de la tabla, y si son diferentes es que pertenece

```

bool perteneceTablaS(char *nombre){
    return buscaLS(tablaSimb, nombre) != finalLS(tablaSimb);
}

```

“anadeEntrada” es una acción que según el nombre (generado en la regla), el tipo y el valor (NULL o un número) inserta un símbolo en la tabla de símbolos.

```
void anadeEntrada(char *nombre, Tipo t, int v){
    Simbolo s;
    s.nombre = strdup(nombre);
    s.tipo = t;
    s.valor = v;
    insertaLS(tablaSimb, finalLS(tablaSimb),
);
```

“inicializaTemporales” es una función que lo que hace es recorrer los temporales y les pone como valor en el array un 0 (que se complementa con la búsqueda de la que hablaremos mas tarde de “nuevoTemp”).

```
void inicializaTemporales(){
    for(int i = 0; i < N_TEMP; i++)
        tmp[i] = 0;
}
```

3.2.4 Sentencias

Antes de tratar las sentencias, vamos a evaluar la lista de sentencias que es lo que coge la regla “program”.

Y basicamente concatena la lista actual con la siguiente sentencia, y si no hay nada el resultado de la regla es la creación de una nueva lista vacía

```
statement_list : statement_list statement {
    $2);                                concatenaLC($1,
}                                $$ = $1;
| /* empty */                      {
}                                $$ = creaLC();
};
```

Ahora ya podemos comenzar con las sentencias de manera individual, lo primero que vamos a hacer va a ser ver las sentencias más sencillas y luego las mas “complicadas” como puede ser el if o el while.

Las sentencias más sencillas son: statement → { statement_list }, statement → read (read_list);, statement → print (print_list);. En estas básicamente se concatena la lista a la solución.

```
statement : LBRACE statement_list RBRACE { $$ = $2;
} | PRINT LPAREN print_list RPAREN DOTCOMMA { $$ = creaLC();
concatenaLC($$, $$);
} | READ LPAREN read_list RPAREN DOTCOMMA { $$ = creaLC();
concatenaLC($$, $$);
}
```

Después de analizar las sentencias más básicas vamos a pasar a las complicadas:

3.2.4.1 Sentencia asignación

Primero de todo debemos comprobar que no es constante y que no pertenece a la tabla de símbolos, luego si estas dos cosas se cumplen se crea una lista de Código vacía a la cual se le añade la expresión, se genera un nombre, y se genera la instrucción de “store word” o “sw” y se inserta en la lista de código.

```
ID ASIGN expression DOTCOMMA {
    if (!perteneceTablaS($1)){
        printf("Error semántico: Variable no declarada '%s' (linea: %d)\n", $1,
YYERROR;
    }
    else if (esConstante($1)){
        printf("Error semántico: Asignación a constante '%s' (linea: %d)\n", $1,
YYERROR;
    }
    else{
        $$ = creaLC();
        concatenaLC($$, $$);
        char nombreVar[20];
        sprintf(nombreVar, "%s", $1);
        Operacion op = {
            .op = "sw",
            .res = recuperarResLC($3),
            .arg1 = strdup(nombreVar),
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op);
        inicializaTemporales();
    }
}
```

“esConstante” es una función que según el nombre recupera el símbolo de la tabla de símbolos, según el índice en la tabla con buscaLS y se compara el tipo del símbolo rescatado con “CONSTANTE”.

```
bool esConstante(char *nombre){  
    return recuperarLS(tablaSimb, buscaLS(tablaSimb, nombre)).tipo ==  
CONSTANTE;
```

3.2.4.2 Sentencia if

Esta sentencia en realidad es sencilla, se crea una etiqueta, se concatena la expresión que básicamente es la condición, luego se crea la instrucción que dice que si que si es 0 (false) se salta a la etiqueta de fin (que es la que hemos generado antes). Se concatena las sentencias que vienen después (para que si no es false se ejecuten), y finalmente se crea la instrucción que marca el final con la etiqueta (para el caso en el que sea false seguir el flujo del programa).

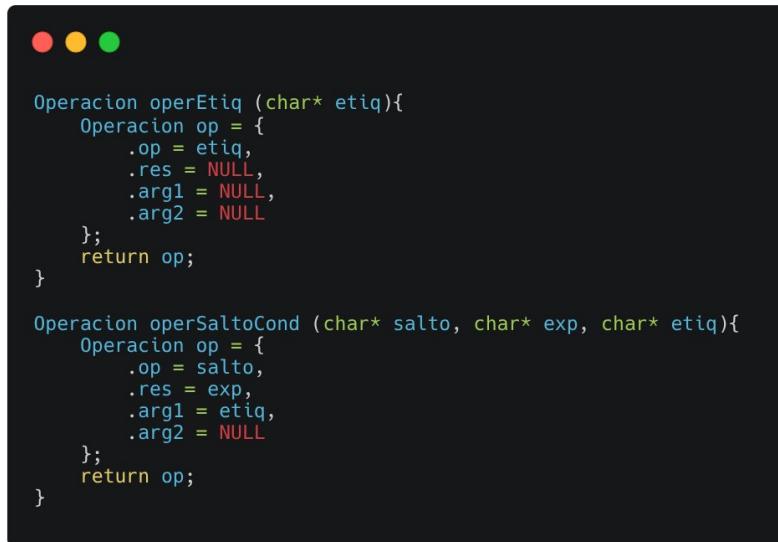
```
| IF LPAREN expression RPAREN statement {  
    $$ = creaLC();  
    char* etiqFin = nuevaEtiq();  
  
    concatenaLC($$, $3);  
    Operacion opCond = operSaltoCond("beqz", recuperarResLC($3),  
    insertaLC($$, finalLC($$), opCond);  
    liberaTemporal(recuperarResLC($3));  
  
    concatenaLC($$, $5);  
    Operacion opEtiq = operEtiq(etiqFin);  
    insertaLC($$, finalLC($$), opEtiq);  
}
```

Para esta función y las siguientes que usan etiquetas y saltos condicionales, he creado unas funciones que permiten simplificar mucho el código y que sea mucho más sencillo:

“nuevaEtiq” es una función que reserva espacio de manera dinámica asigna un nombre con el nombre “In” siendo n un numero que representa al contador de etiquetas.

```
char* nuevaEtiq() {  
    char* etiq = malloc(32);  
    sprintf(etiq, "$l%d", contEtiquetas++);  
    return etiq;  
}
```

“operSaltoCond” es una función que devuelve una Operación y la cree porque vi que era muy sencillo de parametrizar y que la iba a tener que usar bastante, entonces decidí, junto a “operEtiq” hacerlas funciones



```
Operacion operEtiq (char* etiq){
    Operacion op = {
        .op = etiq,
        .res = NULL,
        .arg1 = NULL,
        .arg2 = NULL
    };
    return op;
}

Operacion operSaltoCond (char* salto, char* exp, char* etiq){
    Operacion op = {
        .op = salto,
        .res = exp,
        .arg1 = etiq,
        .arg2 = NULL
    };
    return op;
}
```

3.2.4.3 Sentencia if else

Esta sentencia es un poco más compleja, porque hay que ser muy cuidadoso con las etiquetas y la lógica. La lógica que he aplicado ha sido la que aprendimos en la asignatura de ETC para programar en ensamblador, que era la siguiente:

Evaluación de condición:

- Se cumple: se salta a la etiqueta if y se salta a la etiqueta de Fin.
- No se cumple: se sigue el flujo del código y finalmente se salta a la etiqueta de Fin.

Entonces lo que hago es crear la etiqueta del if y la de fin, concateno la expresión de evaluación, creo la instrucción de salto condicional, si expresión es diferente a 0 (true), a la etiqueta de true. Luego concateno la sentencia del else, y salto al fin. Después creo la instrucción de etiqueta y concateno la información del if y después de eso creo la etiqueta de Fin en la lista de código y lo inserto todo.

En cuanto a la ambigüedad de la gramática, surge por la anidación de varios if y else seguidos, porque puede no quedar claro que if va con que else provocando la ambigüedad pero no es un problema que no nos afecta en gran medida a nuestro compilador.

```

| IF LPAREN expression RPAREN statement ELSE statement {           $$ = creaLC();
char* etiqIf = nuevaEtiq();                                     char* etiqFin = nuevaEtiq();

//ANALISIS, si se da se salta al if
concatenaLC($$, $3);
Operacion opCond = operSaltoCond("bnez",
insertaLC($$, finalLC($$), opCond);
inicializaTemporales();

liberaTemporal(recuperaResLC($3));

//ELSE
concatenaLC($$, $7);

Operacion opSaltoFin = {
    .op = "b",
    .res = etiqFin,
    .arg1 = NULL,
    .arg2 = NULL
};
insertaLC($$, finalLC($$), opSaltoFin);
inicializaTemporales();

// IF
Operacion opIf = operEtiq(etiqIf);
insertaLC($$, finalLC($$), opIf);
concatenaLC($$, $5);

// F IN
Operacion opFin = operEtiq(etiqFin);
insertaLC($$, finalLC($$), opFin);

inicializaTemporales();
}

```

3.2.4.4 Sentencia while

Para esta sentencia el flujo es sencillo y parecido al if else, se crean dos etiquetas, la del while y la del final, primero se introduce la instrucción que marca la etiqueta de inicio (o While), se concatena la condición, después una instrucción de salto condicional que hace que si no se cumple la condición (branch equal zero (false)), se salga del bucle y salte al fin. Se concatenan los statement del while y finalmente se salta a la etiqueta que hemos puesto al principio del bucle y despues se introduce la etiqueta de fin en la lista de codigo.

```
| WHILE LPAREN expression RPAREN statement {  
|     $$ = creaLC();  
|     char* etiqWhile = nuevaEtiqu();  
|     char* etiqFin = nuevaEtiqu();  
|  
|     // Etiqueta para el bucle  
|     Operacion opEtiquW = operEtiqu(etiqWhile);  
|     insertaLC($$, finalLC($$), opEtiquW);  
|  
|     concatenaLC($$, $3);  
|  
|     Operacion opCond = operSaltoCond("beqz", recuperarResLC($3),  
|     insertaLC($$, finalLC($$), opCond);  
|  
|     liberaTemporal(recuperarResLC($3));  
|     concatenaLC($$, $5);  
|  
|  
|     //SUBIDA A COMPROBAR LA CONDICION  
|     Operacion opReinicio = {  
|         .op = "b"  
|         .res = etiqWhile,  
|         .arg1 = NULL,  
|         .arg2 = NULL  
|     };  
|     insertaLC($$, finalLC($$), opReinicio);  
|  
|     //FIN  
|     Operacion opEtiquFin = operEtiqu(etiqFin);  
|     insertaLC($$, finalLC($$), opEtiquFin);  
| }  
|
```

3.2.5 Prints

Primero vamos a ver primero, la impresión de listas que no es más que concatenar ítems a la lista.

```
print_list : print_item {  
    $$ = $1;  
}  
| print_list COMMA print_item {  
    concatenaLC($1,  
    $$ = $1;  
};  
$3;
```

Ahora, vamos a ver que en la impresión de ítems hay dos reglas, una para pintar expresiones y otra para pintar cadenas de caracteres:

Para pintar expresiones, lo que hay que hacer es muy sencillo, cargar en \$v0, el numero de llamada a sistema para pintar un entero que es el 1, cargar en el primer argumento de la llamada a sistema (\$a0) la expresión, y hacer la llamada a sistema.

```

● ● ●

print_item : expression
{
    $$ = $1;

    Operacion op1 = {
        .op = "li",
        .res = "$v0",
        .arg1 = "1",
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op1);

    Operacion op2 = {
        .op = "move",
        .res = "$a0",
        .arg1 = recuperarResLC($1),
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op2);

    Operacion op3 = {
        .op = "syscall",
        .res = NULL,
        .arg1 = NULL,
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op3);

    liberaTemporal(recuperarResLC($1));
}

```

Para imprimir cadenas de caracteres es prácticamente igual, solo que hay que crear un símbolo con la cadena a pintar, pero el resto es igual, cargar en este caso el 4 en \$v0, cargar la cadena que hemos asignado al registro \$a0 y hacer la llamada

```

● ● ●

| STRING
{
    $$ = creaLC();

    Simbolo s;
    s.nombre = strdup($1);
    s.tipo = CADENA;
    s.valor = contCadenas;
    insertaLS(tablaSimb, finalLS(tablaSimb), s);

    char nomCadena[50];
    sprintf(nomCadena, "$str%d", contCadenas++);

    Operacion op1 = {
        .op = "li",
        .res = "$v0",
        .arg1 = "4",
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op1);

    Operacion op2 = {
        .op = "la",
        .res = "$a0",
        .arg1 = NULL,
        .arg2 = strdup(nomCadena)
    };
    insertaLC($$, finalLC($$), op2);

    Operacion op3 = {
        .op = "syscall",
        .res = NULL,
        .arg1 = NULL,
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op3);

}
;
```

3.2.6 Reads

Para los reads hay dos reglas también pero no tan diferentes la una de la otra, una es leer una variable y otra es leer una lista de variables, para ambas hay que hacer lo mismo en cuanto a análisis semántico y es que hay que comprobar que la variable o ID no sea constante y pertenezca a la tabla. Si esto se cumple se carga el numero 5 (para leer) en \$v0, se hace la syscall y lo que devuelve (que se encuentra en \$v0) se guarda en el registro que se le haya asignado.

```
read_list : ID
{
    //COMPROBACIONES INICIALES
    if (!perteneceTablaS($1)) {
        printf("Error semántico: Variable no declarada '%s' (linea: %d)\n", $1, yylineno);
        YYERROR;
    }
    else if (esConstante($1)){
        printf("Error semántico: Asignación a constante '%s' (linea: %d)\n", $1, yylineno);
        YYERROR;
    }
    else{
        $$ = creaLC();
        Operacion op1 = {
            .op = "li",
            .res = "$v0",
            .arg1 = "5",
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op1);

        Operacion op2 = {
            .op = "syscall",
            .res = NULL,
            .arg1 = NULL,
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op2);

        char nombreVar[20];
        sprintf(nombreVar, "%s", $1);
        Operacion op3 = {
            .op = "sw",
            .res = "$v0",
            .arg1 = strdup(nombreVar),
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op3);
    }
}

| read_list COMMA ID
{
    if (!perteneceTablaS($3)) {
        printf("Error semántico: Variable no declarada '%s' (linea: %d)\n", $3, yylineno);
        YYERROR;
    }
    else if (esConstante($3)){
        printf("Error semántico: Asignación a constante '%s' (linea: %d)\n", $3, yylineno);
        YYERROR;
    }
    else{
        $$ = creaLC();
        Operacion op1 = {
            .op = "li",
            .res = "$v0",
            .arg1 = "5",
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op1);

        Operacion op2 = {
            .op = "syscall",
            .res = NULL,
            .arg1 = NULL,
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op2);

        char nombreVar[20];
        sprintf(nombreVar, "%s", $3);
        Operacion op3 = {
            .op = "sw",
            .res = "$v0",
            .arg1 = strdup(nombreVar),
            .arg2 = NULL
        };
        insertaLC($$, finalLC($$), op3);
    }
}
```

3.2.7 Expressions

Las operaciones como suma, resta o multiplicación las he parametrizado y solo tengo que concatenar al resultado la primera expresión, luego la segunda, asignar un nuevo temporal, hacer la operación correspondiente y se inserta en la lista de código.

```
expression : expression PLUSOP expression {  
    $$ = creaLC();  
    concatenaLC($$, $1);  
    concatenaLC($$, $3);  
    int temp = nuevoTemp();  
    char tempStr[5];  
    sprintf(tempStr, "$t%d", temp);  
    Operacion op = crearOpArit("add", recuperarResLC($1), recuperarResLC($3), tempStr);  
    liberaTemporal(recuperarResLC($3));  
    guardaResLC($$, strdup(tempStr));  
    insertaLC($$, finalLC($$), op);  
}
```

Son todas iguales menos el negativo de un número y el operador ternario, que los explicaré a continuación:

Para el operador negativo hay que usar la opción que viene en el seminario de Bison, %prec, tomar la precedencia del operador que hemos llamado en español menos unario. Así resolvemos esa ambigüedad que pasaría por ejemplo en el ejemplo de “x * -y” que el “-” se podría malinterpretar.

```
| MINUSOP expression %prec MINUSOP_UN {  
    $$ = creaLC();  
    concatenaLC($$, $2);  
    int temp = nuevoTemp();  
    char tempStr[5];  
    sprintf(tempStr, "$t%d", temp);  
    Operacion op = crearOpArit("neg", recuperarResLC($2), NULL, tempStr);  
    insertaLC($$, finalLC($$), op);  
    guardaResLC($$, strdup(tempStr));  
}
```

El operador ternario, es como literalmente como un if else en una sola línea entonces el orden de las operaciones es el mismo. Primero se comprueba, la condición y se salta en caso positivo, luego se concatena el caso falso y se salta al final. A continuación se pone la etiqueta true, concatena el caso true, se salta al final y finalmente se pone la instrucción de etiqueta final.

```

| LPAREN expression QMARK expression COLON expression RPAREN
{
    $$ = creaLC();
    char* etiqTrue = nuevaEtiq();
    char* etiqFin = nuevaEtiq();

    int temp = nuevoTemp();
    char tempStr[5];
    sprintf(tempStr, "%d", temp);

    // ANALISIS CASO
    concatenaLC($$, $2);

    Operacion op1 = operSaltoCond("bnez", recuperarResLC($2), etiqTrue);
    insertaLC($$, finalLC($$), op1);

    //CASO FALSE
    concatenalC($$, $6);
    Operacion opFalse = {
        .op = "move",
        .res = strdup(tempStr),
        .arg1 = recuperarResLC($6),
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), opFalse);

    // SALTO FINAL
    Operacion op2 = {
        .op = "be",
        .res = etiqFin,
        .arg1 = NULL,
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), op2);

    // CASO TRUE
    Operacion op3 = operEtiq(etiqTrue);
    insertaLC($$, finalLC($$), op3);

    concatenalC($$, $4);
    Operacion opTrue = {
        .op = "move",
        .res = strdup(tempStr),
        .arg1 = recuperarResLC($4),
        .arg2 = NULL
    };
    insertaLC($$, finalLC($$), opTrue);

    // SALTO IFNAL del TRUE
    Operacion op4 = operEtiq(etiqFin);
    insertaLC($$, finalLC($$), op4);

    guardaResLC($$, strdup(tempStr));
    liberaTemporal(recuperarResLC($4));
    liberaTemporal(recuperarResLC($6));

    liberaLC($4);
    liberaLC($6);
}

```

Para terminar con expression queda expression → ID y expression → ENTERO.

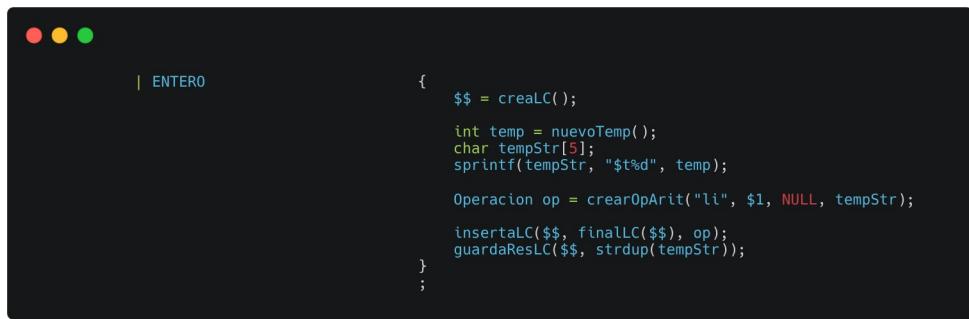
En el caso de ID hay que comprobar antes de comprobar si pertenecen y luego generar un nombre para la variable, generar un nuevo temporal, crear la operación de cargar palabra (“load word” o “lw”) e insertar.

```

| ID
{
    if (perteneceTablaS($1)) {
        $$ = creaLC();
        char nombreVar[20];
        sprintf(nombreVar, "%s", $1);
        int temp = nuevoTemp();
        char tempStr[5];
        sprintf(tempStr, "%d", temp);
        Operacion op = crearOpArit("lw", strdup(nombreVar), NULL, tempStr);
        insertaLC($$, finalLC($$), op);
        guardaResLC($$, strdup(tempStr));
    }
    else{
        printf("Error semántico: Variable no declarada '%s' (linea: %d)\n", $1, yy_lineno);
        YYERROR;
    }
}

```

En el otro caso con el entero únicamente hay que cargar el entero (“load integer” o “li”) en un temporal que previamente hemos de reservar e insertar en en código.



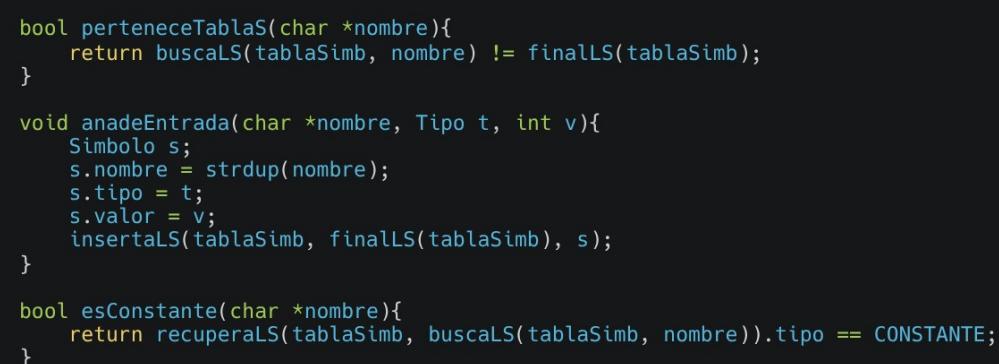
```
| ENTERO
{
    $$ = creaLC();
    int temp = nuevoTemp();
    char tempStr[5];
    sprintf(tempStr, "$t%d", temp);
    Operacion op = crearOpArit("li", $1, NULL, tempStr);
    insertaLC($$, finalLC($$), op);
    guardaResLC($$, strdup(tempStr));
}
```

3.3 Subrutinas de apoyo en C

En esta parte final del código se encuentran una serie de subrutinas de las que ya he ido hablando conforme iba explicando el código, así que voy a resumirlas agrupándolas en 4 grupos:

3.3.1 Subrutinas de variables

En este grupo se encuentran las funciones y acciones que tienen que ver con las variables y su relación con la tabla de símbolos, se encuentran “esConstante”, “anadeEntrada” y “perteneceTablaS”.



```
bool perteneceTablaS(char *nombre){
    return buscaLS(tablaSimb, nombre) != finalLS(tablaSimb);
}

void anadeEntrada(char *nombre, Tipo t, int v){
    Simbolo s;
    s.nombre = strdup(nombre);
    s.tipo = t;
    s.valor = v;
    insertaLS(tablaSimb, finalLS(tablaSimb), s);
}

bool esConstante(char *nombre){
    return recuperarLS(tablaSimb, buscaLS(tablaSimb, nombre)).tipo == CONSTANTE;
}
```

3.3.2 Subrutinas de impresión

Aquí se encuentran dos funciones que usa la regla programa y que sirve para volcar la tabla de símbolos y la lista de código en la salida, ya sea por consola o en un fichero de ensamblador (*.s o *.asm) para poder ejecutarlo después.

Los simbolos se imprimen con un formato, las cadenas de caracteres con str n.^o y los enteros con _[a,b,c, etc.] siendo n el contador de cadenas.

```

void imprimirTablas(){
    printf(".data\n");
    PosicionLista pos_actual = inicioLS(tablaSimb);
    while (pos_actual != finalLS(tablaSimb))
    {
        Simbolo sim = recuperaLS(tablaSimb, pos_actual);
        if(sim.tipo != CADENA) printf("_%s:\n\t.word %d\n",sim.nombre);
        else printf("$str%d:\n\t.asciiz %s\n",sim.valor, sim.nombre);
        pos_actual = siguienteLS(tablaSimb,pos_actual);
    }
}

void imprimeLC(ListaC codigo){
    printf(".text\n");
    printf(".globl main\n");
    printf("main:\n");
    PosicionListaC pos_actual = inicioLC(codigo);
    for (PosicionListaC pos = pos_actual; pos != finalLC(codigo); pos = siguienteLC(codigo, pos)) {
        Operacion op = recuperaLC(codigo, pos);

        if (op.op[0] != '$') {
            printf("\t");
        }
        printf("%s", op.op);

        if (op.res) {
            printf(" %s", op.res);
        }
        if (op.arg1) {
            printf(",%s", op.arg1);
        }
        if (op.arg2) {
            printf(",%s", op.arg2);
        }
        printf("\n");
    }
    printf("#####\n");
    printf("# Fin\n");
    printf("tli $v0, 10\n");
    printf("\tsyscall\n");
}

```

3.3.3 Subrutinas de operación

Son el grupo de declaración de operaciones que usaba mucho y vi que era parametrizable. Están las tres para crear operaciones aritméticas, para poner etiquetas y para crear saltos condicionales.

```

Operacion crearOpArit(char * op, char *oper1, char *oper2, char tempStr[5]){
    Operacion operacion = {
        .op = op,
        .res = strdup(tempStr),
        .arg1 = oper1,
        .arg2 = oper2
    };
    return operacion;
}

Operacion operEtiq (char* etiq){
    Operacion op = {
        .op = etiq,
        .res = NULL,
        .arg1 = NULL,
        .arg2 = NULL
    };
    return op;
}

Operacion operSaltoCond (char* salto, char* exp, char* etiq){
    Operacion op = {
        .op = salto,
        .res = exp,
        .arg1 = etiq,
        .arg2 = NULL
    };
    return op;
}

```

3.3.4 Subrutinas de temporales y etiquetas

Estas funciones sirven por una parte para inicializar, liberar, o buscar temporales libres. Para poder ir funcionando y moviéndonos con temporales todo el rato.

```
void inicializaTemporales(){
    for(int i = 0; i<N_TEMP;i++)
        tmp[i] = 0;
}

void liberaTemporal(char * temp) {
    int idx = atoi(temp + 2);
    tmp[idx] = 0;
}

int nuevoTemp(){
    int i = 0;
    while(i<N_TEMP){
        if(tmp[i]==0){
            tmp[i] = 1;
            return i;
        }
        i++;
    }
    printf("No hay temporales libres\n");
    exit(1);
}
```

En cuanto a etiquetas tenemos la reserva de espacio y el numero de contador de etiquetas para asignar el nombre.

```
char* nuevaEtiqueta() {
    char* etiq = malloc(32);
    sprintf(etiq, "$l%d", contEtiquetas++);
    return etiq;
}
```

Finalmente, la función de error yyerror que informa de los errores sintácticos.

```
void yyerror(char *s) {
    extern char *yytext;
    fprintf(stderr, "Error sintáctico en línea %d cerca de '%s': %s\n", yylineno, yytext, s);
}
```

4. Manual de usuario

Para la correcta ejecución, hay que cumplir unos requisitorios en el equipo (si quieras compilar y ensamblar el programa):

- (Obligatorio) Tener instalado gcc
- (Obligatorio) Tener instalado Flex
- (Obligatorio) Tener instalado Bison
- (Obligatorio) Tener instalado make
- (Opcional) Para poder ejecutar el código es necesario tener un simulador de MIPS, como MARS o SPIM.

Una vez se cuenta con todo lo necesario para la compilación, simplemente habrá que abrir una nueva terminal y ejecutar:

```
[usuario@so ~] cd ruta/del/directorio  
[usuario@so directorio] make
```

Ya tendríamos el proyecto recompilado y podríamos ejecutarlo. Para hacer esto segundo es necesario que pasemos como parámetro, únicamente el fichero *.mc que queramos traducir a ensamblador y podemos (o no) volcarlo en un fichero *.s para poder ejecutarlo en el simulador MIPS, la ejecución se hace de la siguiente manera:

```
[usuario@so directorio] ./minic programa.mc          #Para que se imprima el código por pantalla  
[usuario@so directorio] ./minic programa.mc > programa.s  #Para que se rediriga el código al *.s  
[usuario@so directorio] spim programa.s                #Para que ejecutar el código generado
```

5. Ejemplos de entrada y salida

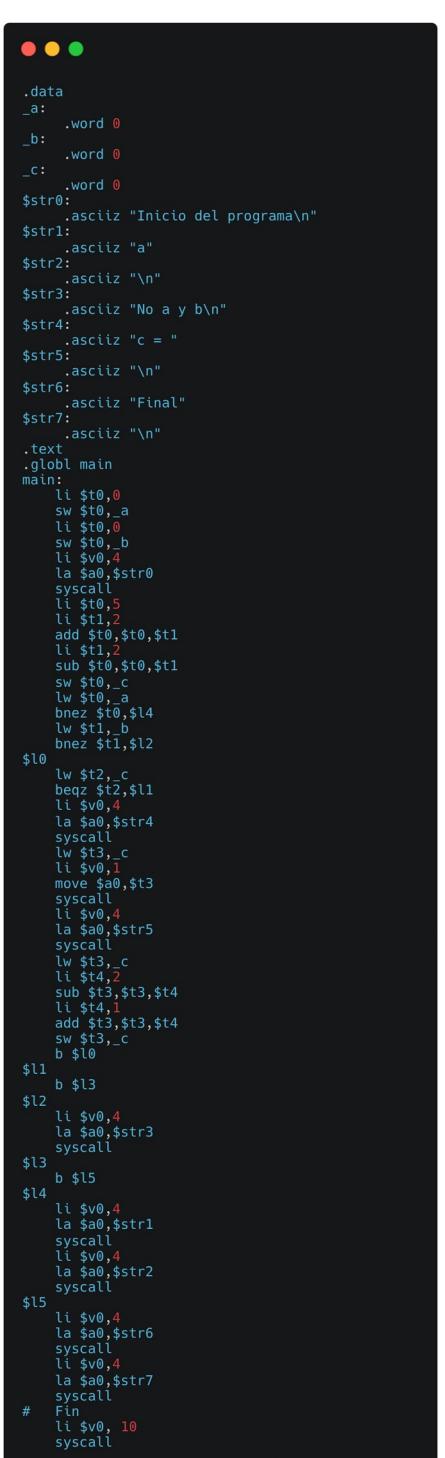
Para este apartado he usado la ayuda de la IA para generar muchos casos que pongan de verdad a prueba la gramática, en cuanto a concatenaciones, registros, comentarios, etc.

5.1 Prueba aulavirtual “prueba.mc”

La siguiente función es para probar el funcionamiento básico de sentencias, de la impresión de expresiones y gestión de registros.



```
prueba() {
const int a=0, b=0;
var int c;
print ("Inicio del programa\n");
c = 5+2-2;
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;
}
print ("Final","\n");
}
```

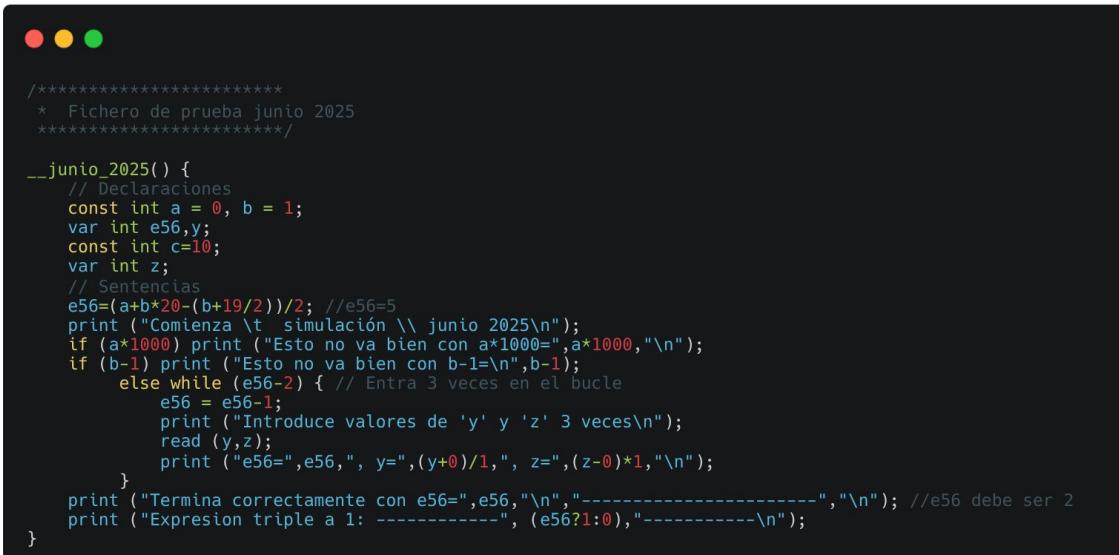


```
.data
_a: .word 0
_b: .word 0
_c: .word 0
$str0: .asciiz "Inicio del programa\n"
$str1: .asciiz "a"
$str2: .asciiz "\n"
$str3: .asciiz "No a y b\n"
$str4: .asciiz "c = "
$str5: .asciiz "\n"
$str6: .asciiz "Final"
$str7: .asciiz "\n"
.text
.globl main
main:
    li $t0,0
    sw $t0,_a
    li $t0,0
    sw $t0,_b
    li $v0,4
    la $a0,$str0
    syscall
    li $t0,5
    li $t1,2
    add $t0,$t0,$t1
    li $t1,2
    sub $t0,$t0,$t1
    sw $t0,_c
    lw $t0,_a
    bnez $t0,$l4
    lw $t1,_b
    bnez $t1,$l2
$l0
    lw $t2,_c
    beqz $t2,$l1
    li $v0,4
    la $a0,$str4
    syscall
    lw $t3,_c
    li $v0,1
    move $a0,$t3
    syscall
    li $v0,4
    la $a0,$str5
    syscall
    lw $t3,_c
    li $t4,2
    sub $t3,$t3,$t4
    li $t4,1
    add $t3,$t3,$t4
    sw $t3,_c
    b $l0
$l1
    b $l3
$l2
    li $v0,4
    la $a0,$str3
    syscall
$l3
    b $l5
$l4
    li $v0,4
    la $a0,$str1
    syscall
    li $v0,4
    la $a0,$str2
    syscall
$l5
    li $v0,4
    la $a0,$str6
    syscall
    li $v0,4
    la $a0,$str7
    syscall
# Fin
    li $v0, 10
    syscall
```

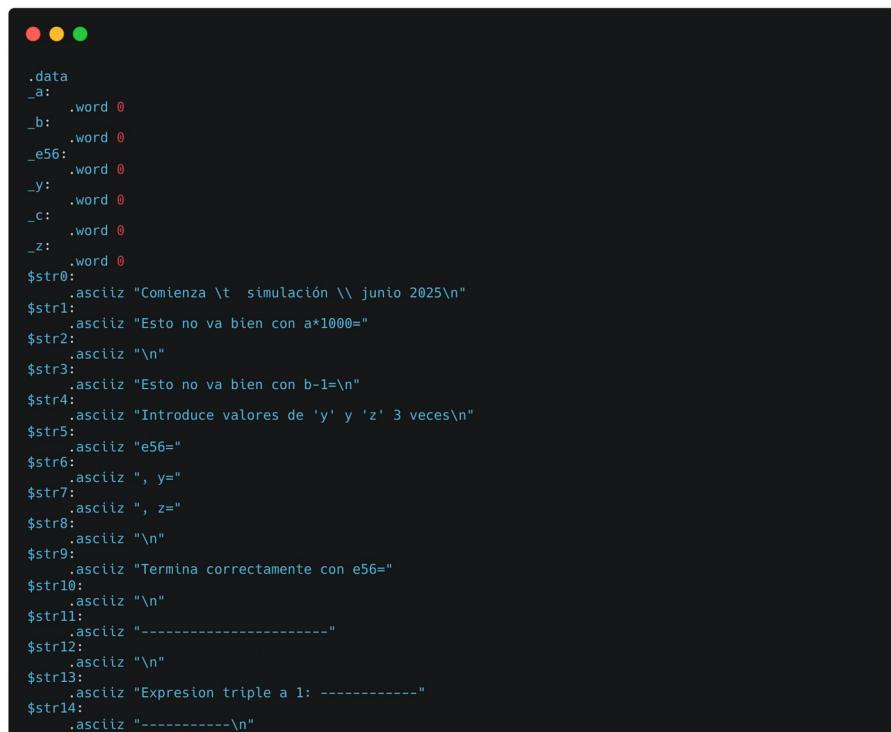
Como podemos observar el código se genera con los nombres de las variables que hemos declarado con una “_” delante, se guardan las cadenas a imprimir desde cero, se cargan las inicializaciones. Se imprime todo, se marcan los if, los else junto al while y finaliza la ejecución con una marca que he puesto para indicarlo y saber que llega al final, como está comentada no afecta en nada.

5.2 Prueba junio

La siguiente prueba que he hecho me ha hecho tener que hacer unos ligeros cambios en el léxico y la sintaxis que no tenía en cuenta antes. Básicamente en “read_list” no concatenaba al principio y en cuanto al léxico en el reconocimiento de cadena no gestionaba bien el \\ que hay en la impresión por ejemplo.



```
*****  
* Fichero de prueba junio 2025  
*****  
  
--junto_2025() {  
    // Declaraciones  
    const int a = 0, b = 1;  
    var int e56,y;  
    const int c=10;  
    var int z;  
    // Sentencias  
    e56=(a+b*20-(b+19/2))/2; //e56=5  
    print ("Comienza \t simulación \\ junio 2025\n");  
    if (a*1000) print ("Esto no va bien con a*1000=",a*1000,"\\n");  
    if (b-1) print ("Esto no va bien con b-1=\n",b-1);  
    else while (e56-2) { // Entra 3 veces en el bucle  
        e56 = e56-1;  
        print ("Introduce valores de 'y' y 'z' 3 veces\n");  
        read (y,z);  
        print ("e56=",e56,", y=", (y+0)/1,", z=", (z-0)*1,"\\n");  
    }  
    print ("Termina correctamente con e56=",e56,"\\n",-----,"\\n"); //e56 debe ser 2  
    print ("Expresion triple a 1: -----", (e56?1:0),-----"\n");  
}
```



```
.data  
_a: .word 0  
_b: .word 0  
_e56: .word 0  
_y: .word 0  
_c: .word 0  
_z: .word 0  
$str0: .asciz "Comienza \t simulación \\ junio 2025\n"  
$str1: .asciz "Esto no va bien con a*1000="  
$str2: .asciz "\n"  
$str3: .asciz "Esto no va bien con b-1=\n"  
$str4: .asciz "Introduce valores de 'y' y 'z' 3 veces\n"  
$str5: .asciz "e56="  
$str6: .asciz ", y="  
$str7: .asciz ", z="  
$str8: .asciz "\n"  
$str9: .asciz "Termina correctamente con e56="  
$str10: .asciz "\n"  
$str11: .asciz "-----"  
$str12: .asciz "\n"  
$str13: .asciz "Expresion triple a 1: -----"  
$str14: .asciz "-----\n"
```

```
.text
.globl main
main:
    li $t0,_0
    sw $t0,_a
    li $t0,_1
    sw $t0,_b
    li $t0,_10
    sw $t0,_c
    lw $t0,_a
    lw $t1,_b
    add $t0,$t0,$t1
    li $t1,_20
    mul $t0,$t0,$t1
    lw $t1,_b
    li $t2,_19
    add $t1,$t1,$t2
    li $t2,_2
    div $t1,$t1,$t2
    sub $t0,$t0,$t1
    li $t1,_2
    div $t0,$t0,$t1
    sw $t0,_e56
    li $v0,_4
    la $a0,$str0
    syscall
    lw $t0,_a
    li $t1,_1000
    mul $t0,$t0,$t1
    beqz $t0,$t0
    li $v0,_4
    la $a0,$str1
    syscall
    lw $t1,_a
    li $t2,_1000
    mul $t1,$t1,$t2
    li $v0,_1
    move $a0,$t1
    syscall
    li $v0,_4
    la $a0,$str2
    syscall
$t0
    lw $t0,_b
    li $t1,_1
    sub $t0,$t0,$t1
    bnez $t0,$t1
$t1
    lw $t1,_e56
    li $t2,_2
    sub $t1,$t1,$t2
    beqz $t1,$t2
    lw $t2,_e56
    li $t3,_1
    sub $t2,$t2,$t3
    sw $t2,_e56
    li $v0,_4
    la $a0,$str4
    syscall
    li $v0,_5
    syscall
    sw $v0,_y
    li $v0,_5
    syscall
    sw $v0,_z
    li $v0,_4
    la $a0,$str5
    syscall
    lw $t0,_e56
    li $v0,_1
    move $a0,$t0
    syscall
    li $v0,_4
    la $a0,$str6
    syscall
    lw $t0,_y
    li $t1,_0
    add $t0,$t0,$t1
    li $t1,_1
    div $t0,$t0,$t1
    li $v0,_1
    move $a0,$t0
    syscall
    li $v0,_4
    la $a0,$str7
    syscall
    lw $t0,_z
    li $t1,_0
    sub $t0,$t0,$t1
    li $t1,_1
    mul $t0,$t0,$t1
    li $v0,_1
    move $a0,$t0
    syscall
    li $v0,_4
    la $a0,$str8
    syscall
    b $t1
$t2
    b $t4
$t3
    li $v0,_4
    la $a0,$str3
    syscall
    lw $t1,_b
    li $t2,_1
    sub $t1,$t1,$t2
    li $v0,_1
    move $a0,$t1
    syscall
$t4
    li $v0,_4
    la $a0,$str9
    syscall
    lw $t0,_e56
    li $v0,_1
    move $a0,$t0
    syscall
    li $v0,_4
    la $a0,$str10
    syscall
    li $v0,_4
    la $a0,$str11
    syscall
    li $v0,_4
    la $a0,$str12
    syscall
    li $v0,_4
    la $a0,$str13
    syscall
    lw $t0,_e56
    bnez $t0,$t5
    li $t2,_0
    move $t3,$t2
    b $t6
$t5
    li $t1,_1
    move $t3,$t1
$t6
    li $v0,_1
    move $a0,$t3
    syscall
    li $v0,_4
    la $a0,$str14
    syscall
#   Fin
    li $v0, 10
    syscall
```

5.3 Pruebas comentarios

Voy a hacer tres pruebas que le he pedido a la inteligencia artificial, el primero para que no funcione, el segundo que “juegue” con los comentarios pero que no funcione, y el tercero que lo haga de “manera salvaje”

5.3.1 Prueba sin final

El código no termina tiene un primer comentario y luego otro que no llega a terminar a mitad de la función:

```
/* Fichero de prueba: comentario sin cerrar
 *
main() {
    var int x;
    x = 5;
    /* Este comentario nunca termina...
    x = x + 1;
    print("Esto nunca debería verse\n");
}
```

```
[sega@archlinux compilador-miniC]$ ./minic pruebas/prueba_nofincom.mc > salidas/nofin.s
Error sintáctico en línea 11 cerca de ''': syntax error
```

5.3.2 Prueba con muchos comentarios pero mal

El código tiene muchos comentarios pero mal puestos y mal colocados a cosa hecha para que falle:

```
/* Fichero de prueba: comentarios mal anidados
 *
main() {
    var int y;
    y = 10;
    /* Comentario que /* intenta anidar */ pero no debe */
    y = y + 2; // Suma dos
    /* Comentario abierto
    y = y + 3; // Esto está dentro del comentario
    // Pero aquí no se cierra nunca
    print("¿Esto se imprime?\n");
}
```

```
[sega@archlinux compilador-miniC]$ ./minic pruebas/salvaje.mc > salidas/salvaje.s
Error sintáctico en línea 8 cerca de 'no': syntax error
```

Como podemos observar, el código falla cerca del “no” es decir cuando de repente encuentra “pero”.

5.3.3 Prueba con muchos comentarios pero bien

En este código podemos encontrar diversos comentarios pero todos usados de manera correcta, con esto podemos concluir la prueba de comentarios:

```
*****  
* Fichero de prueba: comentarios bien usados  
*****  
  
main() {  
    var int a, b;  
    a = 1; // Inicializa a  
    b = 2; /* Inicializa b */  
    /* Suma a y b */  
    a = a + b; // a = 3  
    print("El valor de a es: ", a, "\n"); /* Imprime el  
    resultado */  
    /*  
     * Bloque de comentario  
     * que abarca varias líneas  
     */  
    b = b * 2; // b = 4  
    print("El valor de b es: ", b, "\n");  
    // Fin del programa  
}
```

```
.data  
.a:    .word 0  
.b:    .word 0  
$str0:  .asciiz "El valor de a es: "  
$str1:  .asciiz "\n"  
$str2:  .asciiz "El valor de b es: "  
$str3:  .asciiz "\n"  
.text  
.globl main  
main:  
    li $t0,1  
    sw $t0,_a  
    li $t0,2  
    sw $t0,_b  
    lw $t0,_a  
    lw $t1,_b  
    add $t0,$t0,$t1  
    sw $t0,_a  
    li $v0,4  
    la $a0,$str0  
    syscall  
    lw $t0,_a  
    li $v0,1  
    move $a0,$t0  
    syscall  
    li $v0,4  
    la $a0,$str1  
    syscall  
    lw $t0,_b  
    li $t1,2  
    mul $t0,$t0,$t1  
    sw $t0,_b  
    li $v0,4  
    la $a0,$str2  
    syscall  
    lw $t0,_b  
    li $v0,1  
    move $a0,$t0  
    syscall  
    li $v0,4  
    la $a0,$str3  
    syscall  
#   Fin  
    li $v0, 10  
    syscall
```

5.4 Prueba de registros

Conforme iba probando el compilador me dí cuenta que no gestionaba bien los registros entonces hice ligeras modificaciones como no generar un registro temporal cada vez que hiciese una operación si no reutilizar el primero, entonces le pedí a la IA que me generase un ejemplo para mi gramática que pasase una prueba de estrés en cuanto a registros y que hiciese un caso en el que tuviese el máximo de temporales en uso a la vez para comprobar la liberación y la utilización de registros:

```
● ● ●
testRegistrosTemporales() {
    // Declaraciones
    var int a, b, c, d, e, f, g, h, i, j, k;
    const int uno = 1, dos = 2, tres = 3, cuatro = 4, cinco = 5, seis = 6, siete = 7, ocho = 8, nueve = 9;
    var int resultado;

    // Expresión anidada al máximo (9 temporales)
    resultado = (((uno + dos) * tres) - cuatro) / cinco +
               (((seis * siete) + ocho) - nueve) / uno;

    print("Resultado = ", resultado);

    // Ternario que usa el resultado anterior
    k = (resultado ? (a + b) : (c - d));
    print("k = ", k);

    // Bucle sencillo
    a = 0;
    while (a) {
        a = a + 1;
    }
    print("a = ", a);

    // Operación final
    resultado = (k * a) + (b - c);
    print("Resultado final = ", resultado);
}
```

Esto genera un código de 581 líneas, de las cuales para no saturar la memoria voy a poner unas cuantas líneas que demuestran que el código se genera (de todas formas está en el directorio “pruebas” con el nombre “prueba_9temporales.mc”.

```
● ● ●
.data
_a:
_b:
_c:
_d:
//[...]
$str17:
    .asciiz "RESULTADO FINAL EXTREMO: "
$str18:
    .asciiz "Ternario final: "
$str19:
    .asciiz "== FIN TEST EXTREMO =="
.text
.globl main
main:
    li $t0,1
    sw $t0,_uno
    li $t0,2
//[...]
$L23
    sw $t3,_temp
    li $v0,4
    la $a0,$str18
    syscall
    lw $t0,_temp
    li $v0,1
    move $a0,$t0
    syscall
    li $v0,4
    la $a0,$str19
    syscall
#   Fin
    li $v0, 10
    syscall
```

6. Conclusiones finales

Este trabajo me ha servido para aplicar la teoría estudiada, más centrada en la lógica a la práctica.

Debido a la situación de la convocatoria de Junio me he visto obligado a hacer un repositorio en [github](#) para ir actualizando el proyecto con el control de versiones, además así si fastidiaba algo en algún momento poder volver para atrás de manera muy sencilla, era una tecnología que no controlaba y que he aprendido a usar mientras hacía el proyecto.

Para “debugear” los problemas numerosos de violación de segmento he usado la aplicación Valgrind.

Como he dicho anteriormente, para la generación de casos de prueba he usado Claude Sonnet 4.

En la memoria para la poner el código me he apoyado en un proyecto llamado [Carbon](#) que permite básicamente crear imágenes personalizadas de código muy personalizable.