Memoria Compilador MiniC

Integrantes:
Daniel Pérez Gómez
Víctor Carrillo Gil

ÍNDICE

Explicación del proyecto

El compilador desarrollado cuenta de diferentes partes. **miniC.l:** Contiene el **análisis léxico**, es decir, define los **tokens** del lenguaje miniC, todos los operadores que se usarán más adelante en el **miniC.y**, junto a las declaraciones de NUM e ID con sus respectivos rangos máximos. También identificadores, palabras clave, etc.

Lo más destacable del léxico en nuestro caso podría tratarse de las declaraciones de:
El modo pánico `([^A-Za-z0-9_\+\-\/\$\*{}()<>!=;,"\[\?\:\n])* error();`

Esta línea llama a la función error() siempre que encuentre un carácter que **no se encuentre** en la lista especificada. La función error() muestra por pantalla el carácter que da el error: `printf("Error en el carácter: %s\n", yytext);`

Otras expresiones de gran importancia son: `\"([^"\n\\]|\\[ntr"])*\" { yylval.cadena = strdup(yytext); return STRING;}`
Esta expresión **reconoce cadenas de texto válidas** encerradas entre comillas dobles (“”) aceptando secuencias de escape válidas como `\n, \t, \r.`

En el documento miniC.y, donde se incluyen tanto el análisis sintáctico como el semántico. En cuanto al sintáctico, se puede destacar las siguientes funciones: **Definición de la gramática** del lenguaje (reglas sintácticas), **declaración de variables y constantes**, **análisis de expresiones aritméticas y asignaciones**, **control de flujo con estructuras if y while**, **entrada/salida con read y print**, **manejo de errores semánticos** (por ejemplo, variables no declaradas o intento de modificación de constantes) y **generación de código intermedio**, usando una estructura llamada Operacion.

Estructuras de datos

1. **Tabla de símbolos (`tablaSimbolos`)**
   Estructura que almacena información sobre los identificadores declarados (nombre, tipo, si es constante o variable, etc.).

   Sirve para almacenar los elementos necesarios para llevar a cabo las operaciones, ya que es la encargada de almacenar lo anteriormente mencionado.

2. **Lista de código (`LC`)**
   Almacena el código ensamblador generado para cada sentencia, como asignaciones, operaciones aritméticas, saltos condicionales, etc.

   Se trata de la estructura de datos más importante del código ya que es la encargada de ir almacenando el ensamblador resultado de cada elemento del código a traducir.

WARNING DESPLAZAMIENTO/REDUCCIÓN

miniC.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
Estamos al tanto de este *warning*, es aceptable ya que sabemos el motivo por el que se produce, que es:
Este conflicto ocurre cuando Bison encuentra una estructura if-else anidada, y no puede determinar de forma inmediata a cuál if corresponde el else.

Funciones principales

obtenerEtiqueta()

```
char *obtenerEtiqueta() {
    char aux[32];
    sprintf(aux,"$label%d",contadorEtiquetas++);
    return strdup(aux);
}
```

Genera una etiqueta única (por ejemplo: `$label0, $label1,` etc.) que puede ser usada para marcar lugares como destinos de saltos.

- Usa un contador global `contadorEtiquetas` para asegurar que cada etiqueta sea única.

siguienteRegistro()

```
void siguienteRegistro(char *registro) {
    for (int i = 0; i < 10; i++) {
        if (tablaRegistros[i] == 0) {       /* Busca un registro libre
*/
            tablaRegistros[i] = 1;          /* Marca el registro como
ocupado */
            sprintf(registro, "$t%d", i);   /* Asigna el nombre del
registro a la variable 'registro' */
            return;                         /* Salir después de asignar
el primer registro libre */
        }
    }
    // Si llegamos aquí es porque no hay registros libres
    printf("No hay registros libres.\n");
```

```
}
```

Lleva un control de disponibilidad usando la tabla `tablaRegistros`.

- Si encuentra un registro libre, lo marca como ocupado y lo devuelve.
- Si no hay ninguno disponible, informa con un mensaje de error.

liberarRegistro()

```
void liberarRegistro(char * registro) {
    int indice = atoi(registro + 2);
    tablaRegistros[indice] = 0;}
```

Libera un registro temporal que ya no se necesita, marcándolo como disponible en la tabla.

- Usa `atoi(registro + 2)` para extraer el número del registro, asumiendo el formato `$tX`.

inicializarTablaReg()

```
void inicializarTablaReg(){
    for(int i = 0; i < 10; i++ ){
        tablaRegistros[i] = 0;              /* Se inicializa a cero
porque está vacía obviamente */
    }
}
```

Inicializa la tabla de registros, marcando todos como libres `(0)`.

yyerror()

```
void yyerror(const char *s){
    printf("Error sintático en el token %s y linea %d\n",yytext,
yylineno);
}
```

Manejador de errores de sintaxis, imprime un mensaje indicando el token actual `(yytext)` y la línea del error `(yylineno)`.

imprimeLC() //En listaCodigo.c

```
/* Para imprimir y asi ver si hay fallos */
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");
}
```

Función encargada de mostrar por pantalla la traducción que ha realizado el compilador. Imprime el código mips correspondiente a la entrada miniC proporcionada

imprimeLS() // En listaSimbolos.c

```
void imprimeLS(Lista lista){
  printf(".data\n");
  PosicionLista pos_actual = inicioLS(lista);
    while (pos_actual != finalLS(lista))
    {
```

```
        Simbolo sim = recuperaLS(lista, pos_actual);
        if(sim.tipo != CADENA) printf("_%s:\n\t .word 0\n",sim.nombre);
        else printf("$str%d:\n\t .asciiz %s\n",sim.valor, sim.nombre);
        pos_actual = siguienteLS(lista,pos_actual);
    }

}
```

Se encarga de imprimir la **sección de datos** `(.data)` del código ensamblador generado por el compilador.

Ejemplos de funcionamiento del programa

Entrada para probar los **whiles y el Do while:**

```
PruebaWhile() {

var int c, d, e;
d = 10;
c = 5;
while(d) {
   d = d - 1;
   print ("d = ",d,"\n");
   }

do {
    c = c - 1;
    read(e);
    print(e+e,"\n");
    print("e = ",e,"\n");
    } while (c > 0);
}
```

Salida obtenida tras ejecutar el miniC:

```
.data
_c:                                     .word 0

_d:                                     .word 0

_e:                                     .word 0

$str1:
        .asciiz "d = "
$str2:
        .asciiz "\n"
$str3:
        .asciiz "\n"
$str4:
        .asciiz "e = "
$str5:
        .asciiz "\n"
.text
.globl
main
main:
        li $t0,10
        sw
        $t0,_d li
        $t0,5 sw
        $t0,_c
$label1 :
        lw $t0,_d
        beqz
        $t0,$label2 lw
        $t1,_d
        li $t2,1
        sub
        $t1,$t1,$t2 sw
        $t1,_d
        la
        $a0,$str1 li
        $v0,4
        syscall
        lw $t0,_d
        move
        $a0,$t0 li
        $v0,1 syscall
        la
        $a0,$str2 li
        $v0,4
        syscall
        b $label1
$label2 :
```

```
$label3 :
        lw $t0,_c
```

```
        li $t1,1
        sub
        $t0,$t0,$t1 sw
        $t0,_c
        li
        $v0,5
        syscall
        sw
        $v0,_e lw
        $t0,_e lw
        $t1,_e
        add
        $t0,$t0,$t1
        move $a0,$t0
        li $v0,1
        syscall
        la
        $a0,$str3 li
        $v0,4
        syscall
        la
        $a0,$str4 li
        $v0,4
        syscall
        lw $t0,_e
        move
        $a0,$t0 li
        $v0,1 syscall
        la
        $a0,$str5 li
        $v0,4
        syscall
        lw
        $t0,_c li
        $t1,0
        sgt $t0,$t0,$t1
        bnez
        $t0,$label3
###############
# Fin
        li $v0, 10
        syscall
```

Salida tras ejecutar el ensamblador:

```
d = 9
d = 8
d = 7
d = 6
d = 5
d = 4
d = 3
d = 2
```

```
d = 1
d = 0
4
8
e = 4
```

```
6
12
e = 6
7
14
e = 7
8
16
e = 8
9
18
e = 9
```

Explicación: Con esta entrada estamos probando el funcionamiento del While del read() y por último del Do While.

Entrada para probar los **if y los else if**

```
Pruebaif() {
const int a=2, b=0;
    if (b) print ("b","\n");
    else if (a) print ("a\n");

   if (a) print ("Final","\n");



}
```

Salida obtenida tras ejecutar el miniC:

```
.data
_a:
        .word 0
_b:
        .word 0
$str1:
        .asciiz "b"
$str2:
        .asciiz "\n"
```

```
$str3:
        .asciiz "a\n"
$str4:
        .asciiz "Final"
$str5:
        .asciiz "\n"
.text
.globl
main
main:
        li $t0,2
        sw
        $t0,_a li
        $t0,0 sw
        $t0,_b lw
        $t0,_b
        beqz
        $t0,$label2 la
        $a0,$str1
        li
        $v0,4
        syscall
        la
        $a0,$str2 li
        $v0,4
        syscall
        b $label3
$label2 :
        lw $t1,_a
        beqz
        $t1,$label1 la
        $a0,$str3
        li
        $v0,4
        syscall
$label1 :
$label3 :
        lw $t0,_a
        beqz
        $t0,$label4 la
        $a0,$str4
        li
        $v0,4
        syscall
        la
        $a0,$str5 li
        $v0,4
        syscall
$label4 :
###############
# Fin
        li $v0, 10
```

```
syscall
```

Salida tras ejecutar el ensamblador:

a
Final

Entrada para probar los **el For y más compleja en general**

```
PruebaFor() {
var int n, suma, sumando, i;
suma = 0;
print("Introduce el numero de sumandos: \n");
read(n);
for (i = 0; i < n; i = i + 1) {
    print("Introduce un sumando:\n");
    read(sumando);
    suma = sumando + suma;

    }
    print( "La suma total es: ", suma ,"\n" );
}
```

Salida obtenida tras ejecutar el miniC:

```
.data
_n:
        .word 0
_suma:
        .word 0
_sumando:
        .word 0
_i:
        .word 0
$str1:
        .asciiz "Introduce el numero de sumandos: \n"
$str2:
        .asciiz "Introduce un sumando:\n"
$str3:
        .asciiz "La suma total es: "
$str4:
        .asciiz "\n"
.text
.globl
main
main:
        li $t0,0
        sw
        $t0,_suma la
        $a0,$str1
```

```
        li
        $v0,4
        syscall
        li
        $v0,5
        syscall
        sw
        $v0,_n    li
        $t0,0
        sw $t0,_i
$label1 :
        lw $t1,_i
        lw
        $t2,_n
        slt $t1,$t1,$t2
        beqz
        $t1,$label2 la
        $a0,$str2
        li
        $v0,4
        syscall
        li
        $v0,5
        syscall
        sw
        $v0,_sumando lw
        $t2,_sumando lw
        $t3,_suma
        add
        $t2,$t2,$t3 sw
        $t2,_suma lw
        $t1,_i
        li $t2,1
        add
        $t1,$t1,$t2 sw
        $t1,_i
        b $label1
$label2 :
        la
        $a0,$str3 li
        $v0,4
        syscall
        lw $t0,_suma
        move
        $a0,$t0 li
        $v0,1 syscall
        la
        $a0,$str4 li
        $v0,4
        syscall
###############
# Fin
```

```
    li $v0, 10
    syscall
```

Salida tras ejecutar el ensamblador:

```
Introduce el número de sumandos:
6
```

```
Introduce un sumando:
2
Introduce un sumando:
3
Introduce un sumando:
4
Introduce un sumando:
5
Introduce un sumando:
6
Introduce un sumando:
7
La suma total es: 27
```

Entrada propuesta en el Aula Virtual

```
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");
                  print ("const b = ",b,"\n");
                  c = c-2+1;


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

salida generada al ejecutar miniC:

```
.data
_a:
                                    .word 0
_b:
                                    .word 0
_c:
                                    .word 0
```

```
$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 "const b = "
$str8:
        .asciiz "\n"
$str9:
        .asciiz "Final"
$str10:
        .asciiz "\n"
.text
.globl
main
main:
        li $t0,0
        sw
        $t0,_a li
        $t0,0 sw
        $t0,_b
        la
        $a0,$str1 li
        $v0,4
        syscall
        li
        $t0,5
        li
        $t1,2
        add
        $t0,$t0,$t1 li
        $t1,2
        sub
        $t0,$t0,$t1 sw
        $t0,_c
        lw $t0,_a
        beqz
        $t0,$label5 la
        $a0,$str2
        li
        $v0,4
        syscall
        la
        $a0,$str3 li
```

```
        $v0,4
        syscall
        b $label6
$label5 :
        lw $t1,_b
        beqz $t1,$label3
```

```
        la
        $a0,$str4 li
        $v0,4
        syscall
        b $label4
$label3 :
$label1 :
        lw $t2,_c
        beqz
        $t2,$label2 la
        $a0,$str5
        li $v0,4
        syscall
        lw
        $t3,_c
        move
        $a0,$t3 li
        $v0,1 syscall
        la
        $a0,$str6 li
        $v0,4
        syscall
        la
        $a0,$str7 li
        $v0,4
        syscall
        lw $t3,_b
        move
        $a0,$t3 li
        $v0,1 syscall
        la
        $a0,$str8 li
        $v0,4
        syscall
        lw
        $t3,_c li
        $t4,2
        sub
        $t3,$t3,$t4 li
        $t4,1
        add
        $t3,$t3,$t4 sw
        $t3,_c
        b $label1
$label2 :
$label4 :
$label6 :
        la
        $a0,$str9 li
        $v0,4
        syscall
        la
```

```
        $a0,$str10 li
        $v0,4
        syscall
###############
# Fin
        li $v0, 10
        syscall
```

Salida al ejecutar el ensamblador:

```
Inicio del programa
c = 5
const b = 0
c = 4
const b = 0
c = 3
const b = 0
c = 2
const b = 0
c = 1
const b = 0
Final
```

Ampliaciones desarrolladas

Do While:

Lo primero que hicimos para completar esta ampliación fue declarar el token “DO” en el miniC.l ya que al contrario que el del “While” no lo teníamos aún.

do **return DO;**

Posteriormente en el miniC.y añadimos a la lista de tokens el que acabamos de generar:

%token INT IF ELSE WHILE **DO** FOR PRINT READ COMMA CONST LKEY RKEY VAR

Por último, había que crear la regla de producción y la lógica para producir código:

DO statement WHILE PARI expression PARD FIN {

```
 ListaC lc = creaLC();
        char *etiquetaInicio = obtenerEtiqueta();
         Operacion op;
        op.op = etiquetaInicio;
        op.arg1 = NULL;
        op.arg2 = NULL;
        op.res = ":";
        insertaLC(lc, finalLC(lc), op);
```

```
        // 2. Código del bloque (statement)

        concatenaLC(lc, $2);


        // 3. Código de la condición (expression)

        concatenaLC(lc, $5);


        // 4. Salto condicional (si la condición es verdadera, vuelve a inicio)
        Operacion opJump;
        opJump.op = "bnez"; // Saltar si NO es cero
        opJump.arg1 = etiquetaInicio;
        opJump.arg2 = NULL;
        opJump.res = recuperaResLC($5);
        insertaLC(lc, finalLC(lc), opJump);

        // 5. Liberar registro de la condición
        liberarRegistro(recuperaResLC($5));


        $$ = lc;
    }
```

Este código realiza lo siguiente:

Una etiqueta de inicio.

Código del bloque de instrucciones.

Evaluación de la condición.

Salto condicional hacia la etiqueta si la condición se cumple.

Ampliación de los operadores relacionales: (<, >, <=, >=, ==, !=)

Para este caso al igual que en el DO WHILE, hace falta añadir los tokens para cada operador:

```
"<"          return LT;
">"          return GT;
"=="         return EQ;
"!="         return DIFF;
"<="         return LTE;
"=<"         return LTE;
">="         return GTE;
"=>"         return GTE;
```

En cuanto añadir los tokens al miniC.y también tenemos que tener en mente la precedencia que han de tener estos operadores por lo que los declaramos a esta altura para que el resto de operaciones se realicen siempre antes de realizar la comparación:

```
%left GT LT GTE LTE DIFF EQ
%left QUESTION DPTOS
%left MAS MENOS
%left POR DIV
%left UMENOS
```

Por último añadimos las reglas de producción de cada operador, resulta sencillo ya que siguen una lógica muy similar a los operadores ya implementados:

```
| expression GT expression {
      concatenaLC($1, $3);

        ListaC lc = $1;

        Operacion
        op;  op.op  =
        "sgt";
        op.arg1                          =
        recuperaResLC($1);   op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);

        $$ = lc;
  }
  | expression GTE expression {
      concatenaLC($1, $3);

        ListaC lc = $1;

        Operacion
        op; op.op =
        "sge";
```

```
        op.arg1                         =
        recuperaResLC($1);    op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);

        $$ = lc;
}
| expression LT expression {
    concatenaLC($1, $3);

        ListaC lc = $1;

        Operacion
        op;   op.op   =
        "slt";
        op.arg1                         =
        recuperaResLC($1);    op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);

        liberarRegistro(op.arg2); // Liberar el registro de 'a'


        $$ = lc;
}
| expression LTE expression {
    concatenaLC($1, $3);

        ListaC lc = $1;

        Operacion
        op;   op.op   =
        "sle";
        op.arg1                         =
        recuperaResLC($1);    op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);
```

```
liberarRegistro(op.arg2); // Liberar el registro de 'a'
```

```
        $$ = lc;
    }
    | expression DIFF expression {
        concatenaLC($1, $3);
        ListaC lc = $1;

        Operacion op;

        op.op = "sne"; /*set not equal*/
        op.arg1                         =
        recuperaResLC($1);    op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);

        $$ = lc;
    }
    | expression EQ expression {
        concatenaLC($1, $3);
        ListaC lc = $1;

        Operacion op;

        op.op = "seq"; /*set equal*/
        op.arg1                         =
        recuperaResLC($1);    op.arg2
        =  recuperaResLC($3);  op.res
        = op.arg1;

        liberarRegistro(op.arg2);

        guardaResLC(lc, op.res);
        insertaLC(lc, finalLC(lc), op);

        $$ = lc;
    }
```

Las reglas de producción contienen una nueva operación dependiendo cual sea su equivalencia con las de MIPS

La implementación de una sentencia for:

Para la última ampliación como en las anteriores lo primero es declarar el nuevo token “For” en el miniC.l:
for return FOR;

Una vez hecho esto debemos añadir el token en el miniC.y a la lista de tokens:

%token INT IF ELSE WHILE DO **FOR** PRINT READ COMMA CONST LKEY RKEY VAR

Por último queda lo más complicado, diseñar su regla de producción y el código encargado de generar el ensamblador:
| FOR PARI ID ASIG expression FIN expression FIN ID ASIG expression PARD statement {

```
ListaC lc = creaLC();

        PosicionLista s = buscaLS(tablaSimbolos, $3);
        if (s == finalLS(tablaSimbolos)) {printf("Variable %s no declarada \n",$3);
        errores = errores + 1;}
        else if (recuperaLS(tablaSimbolos, s).tipo == CONSTANTE){errores = errores + 1;
        printf("Asignación a constante\n");}

        PosicionLista s2 = buscaLS(tablaSimbolos, $9);
        if (s2 == finalLS(tablaSimbolos)) {printf("Variable %s no declarada\n", $9);
        errores = errores + 1;}
        else if (recuperaLS(tablaSimbolos, s2).tipo == CONSTANTE) {errores = errores + 1;
        printf("Asignación a constante\n");}

        // 2. Código de inicialización
        concatenaLC(lc, $5);
        char nombreVar[20];
        sprintf(nombreVar, "_%s", $3);
        Operacion opInit;
        opInit.op = "sw";
        opInit.arg1 =strdup(nombreVar);
        opInit.arg2 = NULL;
        opInit.res = recuperaResLC($5);
        insertaLC(lc, finalLC(lc), opInit);

        liberarRegistro(recuperaResLC($5));

        // 3. Etiqueta de inicio del bucle
        char *etiquetaInicio = obtenerEtiqueta();
        Operacion etInicio;
        etInicio.op = etiquetaInicio;
        etInicio.arg1 = NULL;
        etInicio.arg2 = NULL;
```

```
etInicio.res = ":";
insertaLC(lc, finalLC(lc), etInicio);

// 4. Código de la condición
concatenaLC(lc, $7);


// 5. Etiqueta de fin del bucle
char *etiquetaFin = obtenerEtiqueta();

// 6. Salto si condición es
falsa Operacion opCond;
opCond.op = "beqz";
opCond.arg1 = etiquetaFin;
opCond.arg2 = NULL;
opCond.res = recuperaResLC($7);
insertaLC(lc, finalLC(lc), opCond);

liberarRegistro(recuperaResLC($7));

// 7. Código del cuerpo del bucle
concatenaLC(lc, $13);

// 8. Código del incremento
concatenaLC(lc, $11);
char nombreVar1[20];
sprintf(nombreVar1, "_%s",
$9); Operacion opInc;
opInc.op = "sw";
opInc.arg1 =
strdup(nombreVar1); opInc.arg2
= NULL;
opInc.res =
recuperaResLC($11);
insertaLC(lc, finalLC(lc), opInc);

liberarRegistro(recuperaResLC($11));

// 9. Salto al inicio
Operacion opSalto;
opSalto.op = "b";
opSalto.arg1 = NULL;
opSalto.arg2 = NULL;
opSalto.res =
etiquetaInicio;
insertaLC(lc, finalLC(lc), opSalto);

// 10. Etiqueta de fin
Operacion etFin;
etFin.op =
etiquetaFin;
etFin.arg1 = NULL;
```

```
etFin.arg2 = NULL;
etFin.res = ":";
insertaLC(lc, finalLC(lc), etFin);

$$ = lc;
```

EXPLICACIÓN

Se crea una lista vacía para almacenar todo el código generado para el `for`.

```
PosicionLista s = buscaLS(tablaSimbolos, $3);
if (s == finalLS(tablaSimbolos)) {  }
else if (recuperaLS().tipo == CONSTANTE) { }

PosicionLista s2 = buscaLS(tablaSimbolos,

$9);
```

Se verifica que las variables estén declaradas.

También se asegura que no sean constantes, ya que no pueden recibir asignación.
Si hay errores, se notifica y se incrementa el contador `errores`.

```
concatenaLC(lc, $5);
sprintf(nombreVar, "_%s", $3);
opInit.op = "sw";
opInit.arg1 = strdup(nombreVar);
opInit.res = recuperaResLC($5);
```

Se concatena el código de la expresión de inicialización
Se genera la instrucción `sw` para **almacenar el valor** resultante en la variable `$3`.

```
char *etiquetaInicio = obtenerEtiqueta();
```

Se genera una etiqueta para el inicio.

```
opCond.op = "beqz";     // salto si condición == 0
(false) opCond.res = recuperaResLC($7);
opCond.arg1 = etiquetaFin;
```

Se crea una etiqueta de salida del bucle.

Si la condición da falso `(== 0)`, se hace un salto a esa etiqueta.

```
concatenaLC(lc, $13);
```

Se concatena el código del bloque de instrucciones dentro del bucle.

```
concatenaLC(lc, $11);
opInc.op = "sw";
opInc.arg1 = strdup(nombreVar1); // nombre de $9
opInc.res = recuperaResLC($11);
```

Se concatena el código que evalúa la expresión de incremento.
Luego se genera una instrucción `sw` para guardar el resultado en la variable `$9`.

```
opSalto.op = "b";
opSalto.res = etiquetaInicio;
```

Salto al inicio del bucle.

```
etFin.op = etiquetaFin;
etFin.res = ":";
```

Se marca el fin del bucle con una etiqueta, para que `beqz` salte aquí si la condición falla.

Manual de usuario.

Para hacer uso del compilador primero comprobamos tener todos los archivos necesarios en una misma carpeta:

```
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$ ls
PruebaFor.mc    colores.cpp   lex.yy.c       listaSimbolos.c  miniC         miniC.tab.c  miniC_main.c
PruebaWhile.mc  entrada.mc    listaCodigo.c  listaSimbolos.h  miniC.l       miniC.tab.h  salidafor.txt
Pruebaif.mc     entrada2.txt  listaCodigo.h  makefile         miniC.output  miniC.y
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$
```

ejecutar por terminal el comando make y posteriormente ejecutar el programa miniC junto al nombre de la entrada que se le quiera pasar (en este caso de ejemplo):

```
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$ make
make: 'miniC' is up to date.
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$ ./miniC PruebaWhile.mc
.data
_a:
        .word 0
_c:
        .word 0
_d:
        .word 0
_e:
        .word 0
_r:
        .word 0
$str1:
        .asciiz "a tras el question da: "
$str2:
        .asciiz "\n"
$str3:
        .asciiz "d = "
$str4:
        .asciiz "\n"
$str5:
        .asciiz "\n"
$str6:
        .asciiz "e = "
$str7:
        .asciiz "\n"
$str8:
        .asciiz "r = "
$str9:
        .asciiz "\n"
.text
.globl main
main:
        li $t0,10
        sw $t0,_d
```

(el código no aparece entero ya que no cabe en una misma captura de pantalla)
Una vez ejecutado como se puede ver en la imagen: el programa nos generará, si la entrada es válida, un código ensamblador correspondiente a la traducción del código c pasado por la entrada. La cual en este caso era esta:

```
PruebaWhile() {

var int a, c, d, e, r;
d = 10;
c = 5;

a = ( d > c ? c + 2 :  c + 1);
print ("a tras el question da: ",a,"\n");
while(d) {
   d = d - 1;
   print ("d = ",d,"\n");
   }
 do {
    c = c - 1;
    read(e,r);
    print(e+e,"\n");
    print("e = ",e,"\n");
    print("r = ",r,"\n");
    } while (c > 0);
}
```

Hay dos formas de ejecutar el código ensamblador, la primera copiando la salida y pegando en mars para ejecutar, o también redirigiendo la salida de la terminal a un archivo, y ya elegir ejecutarlo usando spim, o el mismo mars abriendo este nuevo archivo:

```
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$ ./miniC PruebaWhile.mc > salida.txt
danie@Daniel:~/Escritorio/Compiladores/miniC(Semantica)/miniC(Semantica)/Proyecto$ spim -file salida.txt
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
a tras el question da: 7
d = 9
d = 8
d = 7
d = 6
d = 5
d = 4
d = 3
d = 2
d = 1
```

CONCLUSIÓN

La práctica nos ha parecido una forma muy útil de aplicar lo visto en la teoría y así mismo dejar de ver un compilador como algo abstracto, para comprender qué está sucediendo gracias a este. Además sirve para comprender en mayor manera cómo aprovechar la memoria y cómo funcionan las operaciones por debajo incluso del código ensamblador, viendo su léxico, las sintaxis y la semántica, hasta poder llegar a la generación de código.

Como única pega, por decir alguna, es que la programación del curso hace que coincidan el desarrollo de 3 proyectos a la vez lo que puede resultar, y lo hace, en una carga que a veces te obliga a dejar un poco de lado la teoría, en cuanto al proyecto estamos muy satisfechos del desarrollo de este, y de ver que el trabajo obtiene resultados tan interesantes como este compilador.