Traducción de miniC a código ensamblador de MIPS

Alumnos:
Rodríguez López, Enrique (Grupo 3.3)

Contacto:
enrique.rodriguezl@um.es (Enrique Rodríguez López)

Asignatura: Compiladores

Curso: 2022/2023

Convocatoria: Junio

Fecha de Entrega: 1 may 2023

2º Grado en Ingeniería Informática

1. Introducción

Este informe tiene cómo objetivo documentar el proyecto de un compilador de miniC a código ensamblador MIPS realizado en el contexto de la parte de prácticas de la asignatura de Compiladores de 2º de Ingeniería Informática en la UMU.

Se trata de un programa al que le introduces cómo input el código de un programa escrito en el lenguaje miniC y a partir de este te genere una salida en ensamblador MIPS correcta. En caso de fallos de sintaxis y otros problemas, el compilador te avisará de los diferentes errores.

Este documento se va a organizar siguiendo las diferentes fases que se implementan en un compilador: análisis léxico, análisis sintáctico, análisis semántico y generación de código.

En cada una de las fases se comentará la implementación particular y las diferentes decisiones de diseño que se han tomado. Además, comentaré algunas observaciones y problemas con los que me he ido encontrando.

Por último, veremos las dos mejores opciones que hemos implementado además de unas conclusiones generales de la práctica.

2. Análisis de léxico

El análisis léxico ha sido realizado usando el software Flex que nos permite especificar los diferentes tokens de nuestro lenguaje.

Lo primero que hemos hecho es identificar los diferentes tokens que van a aparecer en nuestro lenguaje. Incluye palabras reservadas, carácteres especiales, identificadores, números y cadenas.

```
void                    return VOID;
var                     return VAR;
const                   return CONST;
if                      return IF;
else                    return ELSE;
while                   return WHILE;
do                      return DO;
print                   return PRINT;
read                    return READ;
for                     return FOR;
from                    return FROM;
to                      return TO;
by                      return BY;
```

Cada uno de estos tokens está referenciado en Bison (análisis sintáctico) para poder usarlo más adelante.

El objetivo del análisis léxico será encontrar errores léxicos en el programa. Por ejemplo: cuando un carácter no está permitido o cuando un identificador es demasiado largo. Además de comprobar estos errores, Flex va a tokenizar el código para facilitar la tarea al análisis sintáctico.

Muchas de estos tokens son simples palabras reservadas o carácteres, pero en otros casos son mucho más complejos y habrá que diseñar expresiones regulares que hagan “match” con estos tokens. Por ejemplo, para el caso de los comentarios se tendrá que diseñar una expresión regular que sea capaz de agrupar todo el comentario.

Nuestro compilador también tiene implementado la recuperación de errores en modo pánico, consiste en recoger todos los caracteres que no hayan sido definidos previamente con el objetivo de detectar todos aquellos caracteres inválidos que no están recogidos por las reglas anteriores.

Para la verificación hemos usado los ficheros proporcionados para su uso. Verificando que se tokeniza correctamente y el compilador avisa en los escenarios de error.

Las mayores dificultades que hemos tenido en esta fase han sido con el diseño de la expresión regular para los comentarios multilínea y la implementación del modo pánico.

3. Análisis sintáctico

El análisis sintáctico ha sido realizado usando el software Bison que nos permite reconocer sintácticamente ficheros generados por la gramática de miniC. El fichero de Bisón trabaja con los tokens de Flex (análisis léxico) . En él hay que incluir las diferentes reglas sintácticas además prioridades, tipos de datos de los diferentes tokens…

Nos hemos basado en está gramática de miniC para construir en Bison las diferentes reglas necesarias en el análisis sintáctico.

program	→	`void` *id* `(` `)` `{` declarations statement_list `}`
declarations	→	declarations `var` identifier_list `;`
	|	declarations `const` identifier_list `;`
	|	λ
identifier_list	→	asig
	|	identifier_list `,` asig
asig	→	*id*
	|	*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 `)`
	|	*id*
	|	*num*

La fase de análisis sintáctico nos permite comprobar si el código es válido sintácticamente en el lenguaje miniC. En caso de tener errores sintácticos el compilador avisará de la existencia de errores. Por ejemplo, cuando se te olvida poner un “;” después de una instrucción.

Esta gramática tiene un conflicto de desplazamiento/reducción en el caso de if/if-else que puede ser solucionado especificando la precedencia haciendo uso de *"%prec".*

Bison te proporciona un mecanismo que te informa de todos los conflictos de precedencia y te dice cual es la acción por defecto. Hemos revisado esta información para comprobar que se resuelven todos los conflictos correctamente.

Para la verificación hemos usado los ficheros proporcionados para su uso. Comprobando que la salida coincide con la esperada.

Las mayores dificultades en esta fase las hemos tenido en la parte de especificar las prioridades y precedencias.

4. Análisis semántico

El análisis semántico se encarga de crear una tabla de símbolos donde están los diferentes símbolos (variables, constantes y cadenas) y de gestionar los diferentes errores que puedan haber en el código relacionados con ellos. Por ejemplo: cuando se usa una variable sin ser declarada.

Para ello hacemos uso de un TDA (tipo de dato abstracto) que representa la tabla de símbolos. En esta fase se va analizando el código y se van insertando los diferentes símbolos cuando se declaran, comprobando que no están en uso. Al mismo tiempo, cuando se debe usar un símbolo se comprueba que este haya sido declarado anteriormente.

Para el caso de los strings, hemos creado un contador de strings que va asignando un número único a cada string autoincrementandose.

Hemos definido funciones auxiliares cómo *esCostante(símbolo)* que nos ayudan en el proceso de verificación del código. Por ejemplo: detectar cuando se redefine una constante.

A la hora de insertar en la tabla de símbolos, lo hemos programado para que se inserte de manera ordenada y agrupada los símbolos según sean variables/constantes o cadenas, siendo estas las primeras en aparecer, de manera que luzca de forma más limpia al imprimir la tabla de símbolos

Las mayores dificultades en esta fase las hemos tenido en la parte de usar la estructura de la tabla de símbolos correctamente.

5. Generación de código

Por último, la generación de código es capaz de generar el código de ensamblador MIPS en base a las fases anteriores. Para ello, utilizamos una estructura llamada lista de código donde se van insertando las diferentes instrucciones en ensamblador MIPS que correspondería al código en miniC.

Este es un proceso recursivo, donde se van generando las diferentes instrucciones en nodos hojas y se van concatenando en los niveles superiores para formar la lista de código completa. Posterior al concatenado, se libera la lista de código que se une a la lista principal con el fin de liberar memoria.

En este proceso, debemos ir gestionando también los registros temporales ($t0-$t9) manteniendo un array de booleanos que indica si cada uno de los registros está disponible o no. Se debe hacer también la gestión de esta estructura, liberando registros cuando ya no sean necesarios.

También se mantiene un contador de etiquetas que ayuda a ir asignando un número único a cada etiqueta que se autoincrementa.

En esta fase hemos tenido que implementar: traducción de expresiones, asignaciones, print, read, saltos (if,if-else,while).

La traducción de expresiones (suma, resta, multiplicación, división, carga de registro…) son uno de los “nodo hoja” que generan instrucciones que mandan a niveles superiores.

Uno de los problemas más importantes que hemos tenido es en la impresión de la lista de código final, que hemos entendido al final de todo el proceso una vez hemos tenido una alta comprensión de cómo funciona el proceso recursivo completo.

Como resultado final se envía a la salida la lista de símbolos y la lista de código con el formato ensamblador.

6. Mejoras opcionales

6.1. Mejoras opcionales: do-while

Para hacer el do-while hemos decidido usar una sintaxis muy similar a la versión de do-while en c.

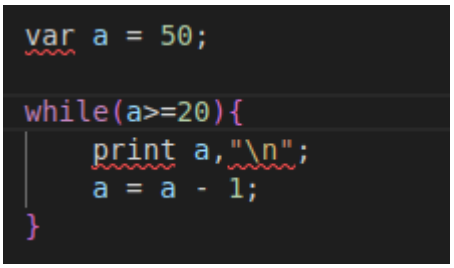
```
var a=1;

do{
    print "Hola";
}while(a);
```

En este caso, el funcionamiento es muy similar al while lo único que hacemos es la comprobación del salto después de ejecutar las instrucciones. (en vez de antes).

6.2. Mejoras opcionales: Operadores relacionales

Se han implementado los siguientes operadores relacionales

Operador relacional	Funcionamiento
==	Comprueba que dos expresiones son iguales
!=	Comprueba que dos expresiones son distintas
<	Comprueba que una expresión es menor a otra
<=	Comprueba que una expresión es menor o igual a otra
>	Comprueba que una expresión es mayor a otra
>=	Comprueba que una expresión es mayor o igual a otra



```
var a = 50;

while(a>=20){
    print a,"\n";
    a = a - 1;
}
```

Ejemplo de código

Para implementar esta funcionalidad, se han utilizado las siguientes instrucciones en MIPS.

Operador relacional	Funcionamiento
==	xor + sltiu
!=	xor
<	slt
<=	sle
>	sgt
>=	sge

Funcionan realmente cómo una expresión, donde 1 representa al valor true y 0 representa al valor false, de esta manera en los operadores de control (if, while…) sigue funcionando ya que internamente funcionan cómo un bez o un bnez (comprueban si el valor es 0 o distinto de 0)

6.3. Mejoras opcionales: for

El caso del for es un poco más complejo, esta es la sintaxis que hemos propuesto. Para usar un for primero tienes que declarar una variable que utilizará el for como iterador, en este caso es la variable j.

Existen tres alternativas para el for:

- For estandar, con límite inferior y superior e indicando el paso de actualización
- For con límite inferior y superior sin indicar paso de actualización (por defecto, 1)
- For con límite inferior y con condición de parada.

En la parte de “from 1 to 20” se establecerán el inicio y el fin del for. El límite superior es inclusivo por lo que en este caso iría de 1 a 20 (ambos inclusives).

Por último en la parte del “by 1” se le especifica cómo se va a incrementar el iterador. En este caso se incrementará de uno en uno (cómo si fuera un j++).

```
var j;
for j from 1 to 20 by 1{
    print "Hola ",j;
}
```

Version en miniC

```
for(int j=1;j<=20;j++){
    printf("Hola %d",j);
}
```

Versión equivalente en C

En la segunda alternativa, la sintaxis es exactamente igual pero se ignora el by, teniendo cómo paso de actualización 1.

```
var j;
for j from 1 to 10{
    print "Hola ",j;
}
```

Segunda alternativa: Sin paso de actualización

En la tercera alternativa, se incluye la palabra reservada “until” que determina la expresión correspondiente a la condición de parada, si se cumple, se detendrá el bucle for.

```
var a;
for a from 10 until a>25{
    print a,"\n";
    a = a + 1;
}
```

Para la implementación de esta mejora hemos empezado con una versión de for muy simplificada (solo con el from-to) y hemos añadido posteriormente la mejora del by, de la variable del iterador y por último la versión con operadores relacionales. Hemos utilizado la instrucción “bgt” de MIPS para poder obtener mayor funcionalidad.

7. Manual de usuario

Para poder ejecutar el programa primero tenemos que compilarlo usando “make”. Una vez tengamos el ejecutable tenemos que ejecutarlo de la siguiente manera

```
$ make
```

```
$ ./minic programa > programa_salida
```

Donde “programa” es el programa escrito en miniC y “programa_salida” es el programa generado por el compilador en ensamblador MIPS.

8. Ejemplo funcionamiento

8.1. Hola mundo

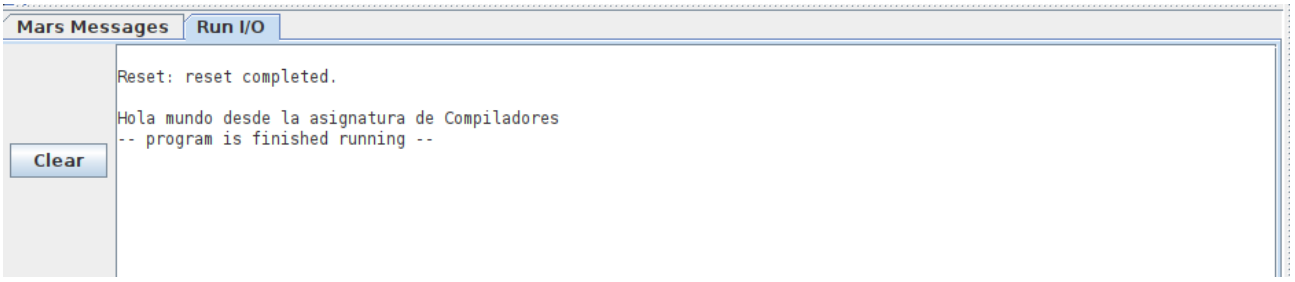
Código en MiniC

```
void main(){
   print "Hola mundo", " desde la asignatura ", "de Compiladores";
}
```

Código en MIPS

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

$str1:
   .asciiz "Hola mundo"
$str2:
   .asciiz " desde la asignatura "
$str3:
   .asciiz "de Compiladores"

###################
# Seccion de codigo
   .text
   .globl main
main:
   li $v0, 4
   la $a0, $str1
   syscall
   li $v0, 4
   la $a0, $str2
   syscall
   li $v0, 4
   la $a0, $str3
   syscall

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.2. Operadores aritméticos

Código en MiniC

```
void main(){
   var a = 5
   var b = 2;
   print "Suma: ", a + b, "\n";
   print "Resta: ", a - b, "\n";
   print "Multiplicacion: ", a * b, "\n";
   print "Division: ", a / b, "\n";
}
```

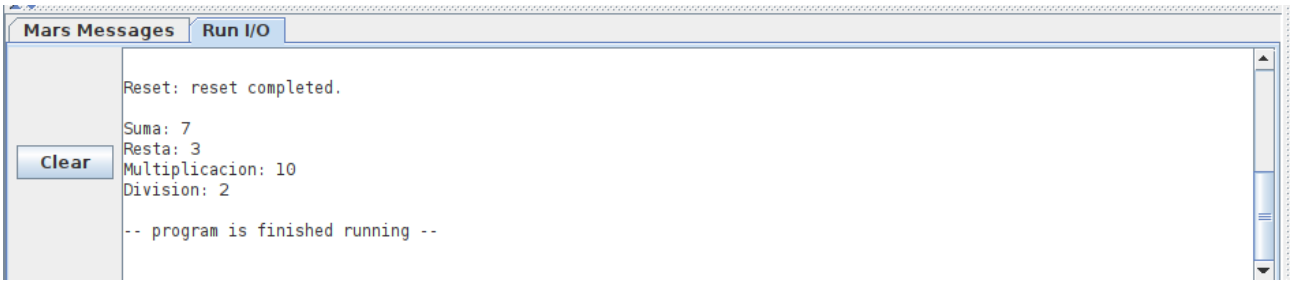
Código en MIPS

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

$str1:
   .asciiz "Suma: "
$str2:
   .asciiz "\n"
$str3:
   .asciiz "Resta: "
$str4:
   .asciiz "\n"
$str5:
   .asciiz "Multiplicacion: "
$str6:
   .asciiz "\n"
$str7:
   .asciiz "Division: "
$str8:
   .asciiz "\n"
_a:
```

```
    .word 0
_b:
    .word 0

###################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 5
    sw $t0, _a
    li $t0, 2
    sw $t0, _b
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _a
    lw $t1, _b
    add $t0, $t0, $t1
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str2
    syscall
    li $v0, 4
    la $a0, $str3
    syscall
    lw $t0, _a
    lw $t1, _b
    sub $t0, $t0, $t1
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str4
    syscall
    li $v0, 4
    la $a0, $str5
    syscall
    lw $t0, _a
    lw $t1, _b
    mul $t0, $t0, $t1
    li $v0, 1
```

```
   move $a0, $t0
   syscall
   li $v0, 4
   la $a0, $str6
   syscall
   li $v0, 4
   la $a0, $str7
   syscall
   lw $t0, _a
   lw $t1, _b
   div $t0, $t0, $t1
   li $v0, 1
   move $a0, $t0
   syscall
   li $v0, 4
   la $a0, $str8
   syscall

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.3. IF-ELSE

Código en MiniC

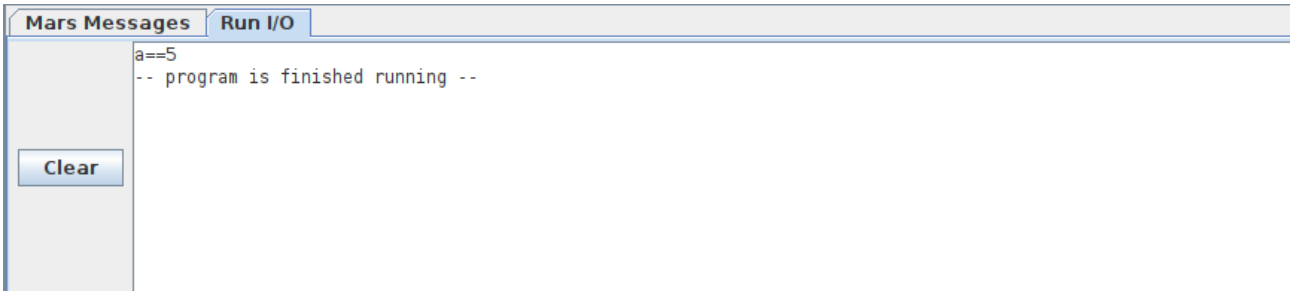
```
void main(){
   var a = 5;

   if(a==6){
       print "a=6";
   }else if(a==4){
       print "a==4";
   }else if(a==5){
       print "a==5";
   }else{
       print "Lo siento, no conozco el valor de la variable A.";
   }
}
```

Código en MIPS

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

$str1:
   .asciiz "a=6"
$str2:
   .asciiz "a==4"
$str3:
   .asciiz "a==5"
$str4:
   .asciiz "Lo siento, no conozco el valor de la variable A."
_a:
   .word 0

##################
# Seccion de codigo
   .text
   .globl main
main:
   li $t0, 5
   sw $t0, _a
   lw $t0, _a
   li $t1, 6
   xor $t0, $t0, $t1
```

```
   sltiu $t0, $t0, 1
   beqz $t0, $l5
   li $v0, 4
   la $a0, $str1
   syscall
   b $l6
$l5:
   lw $t1, _a
   li $t2, 4
   xor $t1, $t1, $t2
   sltiu $t1, $t1, 1
   beqz $t1, $l3
   li $v0, 4
   la $a0, $str2
   syscall
   b $l4
$l3:
   lw $t2, _a
   li $t3, 5
   xor $t2, $t2, $t3
   sltiu $t2, $t2, 1
   beqz $t2, $l1
   li $v0, 4
   la $a0, $str3
   syscall
   b $l2
$l1:
   li $v0, 4
   la $a0, $str4
   syscall
$l2:
$l4:
$l6:

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.4. WHILE

Código en MiniC

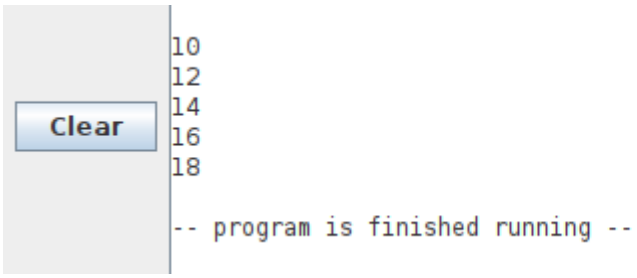
```
void main(){
   var a = 10;
   while(a<20){
       print a,"\n";
       a = a+2;
   }
}
```

Código en MIPS

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

$str1:
   .asciiz "\n"
_a:
   .word 0

###################
# Seccion de codigo
   .text
   .globl main
main:
   li $t0, 10
   sw $t0, _a
$l1:
   lw $t0, _a
   li $t1, 20
   slt $t0, $t0, $t1
   beqz $t0, $l2
```

```
   lw $t1, _a
   li $v0, 1
   move $a0, $t1
   syscall
   li $v0, 4
   la $a0, $str1
   syscall
   lw $t1, _a
   li $t2, 2
   add $t1, $t1, $t2
   sw $t1, _a
   b $l1
$l2:

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.5. DO-WHILE

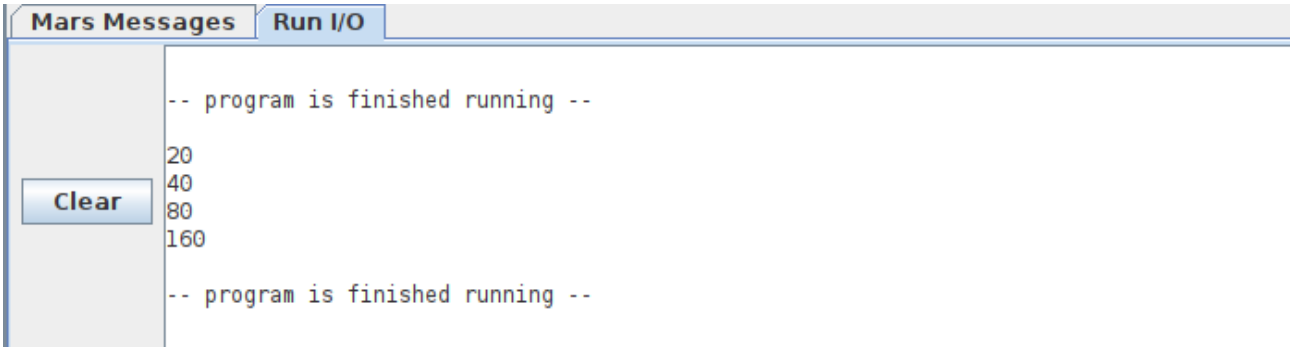
Código en MiniC

```
void main(){
   var a = 10;

   do{
       a = a * 2;
       print a,"\n";
   }while(a<100);
}
```

Código en MIPS

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

$str1:
    .asciiz "\n"
_a:
    .word 0

###################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 10
    sw $t0, _a
$l1:
    lw $t0, _a
    li $t1, 2
    mul $t0, $t0, $t1
    sw $t0, _a
    lw $t0, _a
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _a
    li $t1, 100
    slt $t0, $t0, $t1
    bnez $t0, $l1

##################
# Final: exit
    li $v0, 10
    syscall
```

Ejecución del programa en MARS



8.6. OPERADORES RELACIONALES

Código en MiniC

```
void main(){
   var a = 4;

   if (a!=5){
       if(a<5){
           while(a<5){
               print a,"\n";
               a=a+1;
           }
       }else if(a>5){
           while(a>5){
               print a,"\n";
               a=a-1;
           }
       }
   }else{
       print "a es cinco";
   }
   print "Valor final de a:",a;
}
```

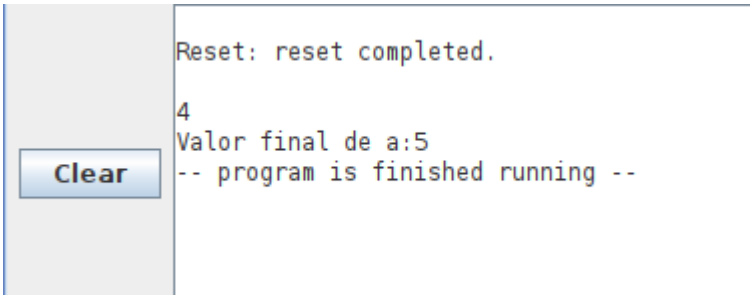
Código en MIPS

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

$str1:
   .asciiz "\n"
$str2:
```

```
   .asciiz "\n"
$str3:
   .asciiz "a es cinco"
$str4:
   .asciiz "Valor final de a:"
_a:
   .word 0

###################
# Seccion de codigo
   .text
   .globl main
main:
   li $t0, 4
   sw $t0, _a
   lw $t0, _a
   li $t1, 5
   xor $t0, $t0, $t1
   beqz $t0, $l8
   lw $t1, _a
   li $t2, 5
   slt $t1, $t1, $t2
   beqz $t1, $l6
$l1:
   lw $t2, _a
   li $t3, 5
   slt $t2, $t2, $t3
   beqz $t2, $l2
   lw $t3, _a
   li $v0, 1
   move $a0, $t3
   syscall
   li $v0, 4
   la $a0, $str1
   syscall
   lw $t3, _a
   li $t4, 1
   add $t3, $t3, $t4
   sw $t3, _a
   b $l1
$l2:
   b $l7
$l6:
   lw $t2, _a
```

```
   li $t3, 5
   sgt $t2, $t2, $t3
   beqz $t2, $l5
$l3:
   lw $t3, _a
   li $t4, 5
   sgt $t3, $t3, $t4
   beqz $t3, $l4
   lw $t4, _a
   li $v0, 1
   move $a0, $t4
   syscall
   li $v0, 4
   la $a0, $str2
   syscall
   lw $t4, _a
   li $t5, 1
   sub $t4, $t4, $t5
   sw $t4, _a
   b $l3
$l4:
$l5:
$l7:
   b $l9
$l8:
   li $v0, 4
   la $a0, $str3
   syscall
$l9:
   li $v0, 4
   la $a0, $str4
   syscall
   lw $t0, _a
   li $v0, 1
   move $a0, $t0
   syscall

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.7. FOR

Código en MiniC

```
void main(){
   var a;
   var b;
   var c;

   // For estandar
   for a from 0 to 10 by 2{
       print "a=",a,"\n";
   }

   // For con by 1 por defecto
   for b from 0 to 5{
       print "b=",b,"\n";
   }

   // For con condición de parada
   for c from 3 until c>20{
       print "c=",c,"\n";
       c = c*3;
   }
}
```

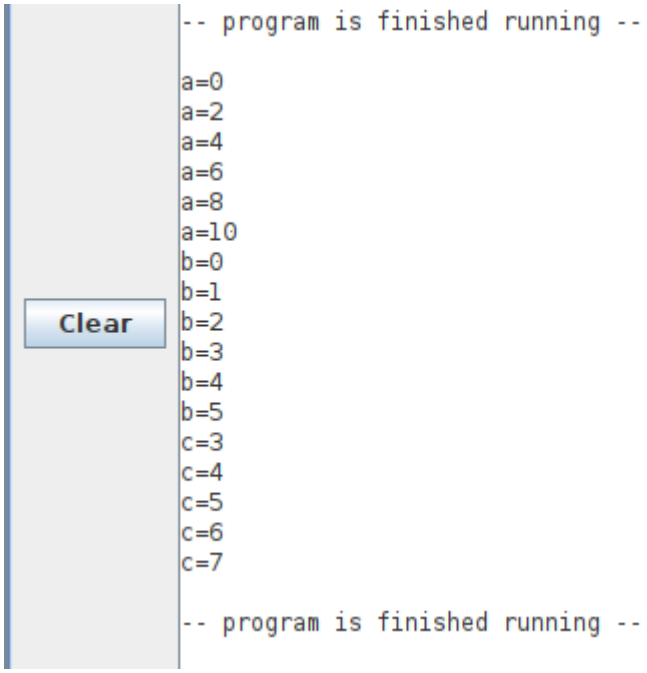
Código en MIPS

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

$str1:
    .asciiz "a="
$str2:
    .asciiz "\n"
$str3:
    .asciiz "b="
$str4:
    .asciiz "\n"
$str5:
    .asciiz "c="
$str6:
    .asciiz "\n"
_a:
    .word 0
_b:
    .word 0
_c:
    .word 0

###################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 0
    li $t1, 10
    li $t2, 2
$l1:
    bgt $t0, $t1, $l2
    sw $t0, _a
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t3, _a
    li $v0, 1
    move $a0, $t3
    syscall
    li $v0, 4
    la $a0, $str2
```

```
   syscall
   add $t0, $t0, $t2
   b $l1
$l2:
   li $t0, 0
   li $t1, 5
$l3:
   bgt $t0, $t1, $l4
   sw $t0, _b
   li $v0, 4
   la $a0, $str3
   syscall
   lw $t2, _b
   li $v0, 1
   move $a0, $t2
   syscall
   li $v0, 4
   la $a0, $str4
   syscall
   addi $t0, $t0, 1
   b $l3
$l4:
   li $t0, 3
   lw $t1, _c
   li $t2, 20
   sgt $t1, $t1, $t2
$l5:
   lw $t1, _c
   li $t2, 20
   sgt $t1, $t1, $t2
   bnez $t1, $l6
   sw $t0, _c
   li $v0, 4
   la $a0, $str5
   syscall
   lw $t2, _c
   li $v0, 1
   move $a0, $t2
   syscall
   li $v0, 4
   la $a0, $str6
   syscall
   lw $t2, _c
   li $t3, 3
```

```
   mul $t2, $t2, $t3
   sw $t2, _c
   addi $t0, $t0, 1
   b $l5
$l6:

#################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.8. Suma de primeros 500 números pares

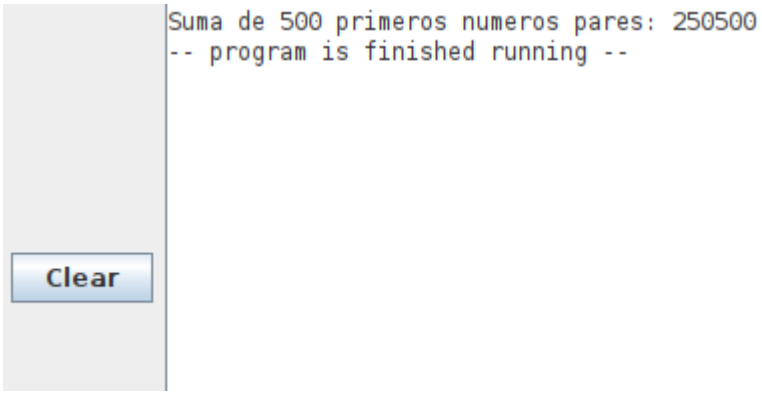
Código en MiniC

```
void main(){

   var suma;
   var j;
   for j from 0 to 1000 by 2{
       suma = suma + j;
   }
   print "Suma de 500 primeros numeros pares: ", suma;
}
```

Código en MIPS

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

$str1:
    .asciiz "Suma de 500 primeros numeros pares: "
_suma:
    .word 0
_j:
    .word 0

###################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 0
    li $t1, 1000
    li $t2, 2
$l1:
    bgt $t0, $t1, $l2
    sw $t0, _j
    lw $t3, _suma
    lw $t4, _j
    add $t3, $t3, $t4
    sw $t3, _suma
    add $t0, $t0, $t2
    b $l1
$l2:
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _suma
    li $v0, 1
    move $a0, $t0
    syscall

##################
# Final: exit
    li $v0, 10
    syscall
```

Ejecución del programa en MARS



8.9. Cálculo del factorial de un número

Código en MiniC

```
void main() {

   var n = 5; // Calcularemos el factorial de 5

   var fact = 1;
   var i = 1;

   while(i<=n){
       fact = fact * i;
       i=i+1;
   }

   print "Fact(",n,") = ", fact;
}
```

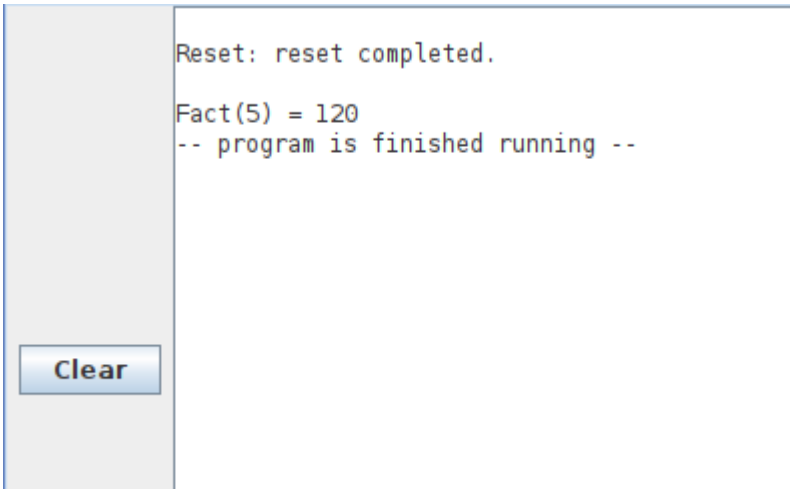
Código en MIPS

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

$str1:
   .asciiz "Fact("
$str2:
   .asciiz ") = "
_n:
```

```
    .word 0
_fact:
    .word 0
_i:
    .word 0

###################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 5
    sw $t0, _n
    li $t0, 1
    sw $t0, _fact
    li $t0, 1
    sw $t0, _i
$l1:
    lw $t0, _i
    lw $t1, _n
    sle $t0, $t0, $t1
    beqz $t0, $l2
    lw $t1, _fact
    lw $t2, _i
    mul $t1, $t1, $t2
    sw $t1, _fact
    lw $t1, _i
    li $t2, 1
    add $t1, $t1, $t2
    sw $t1, _i
    b $l1
$l2:
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _n
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str2
    syscall
    lw $t0, _fact
    li $v0, 1
```

```
   move $a0, $t0
   syscall

##################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



8.10. Algoritmo de Fibonnaci

Código en MiniC

```
void main(){

   var n = 20; // Calcularemos 20 primeros numeros de fibonnaci

   var fib = 1;
   var prevFib = 1;
   var i;
   var temp;

   if (n<=1) {
       print "1";
   }else{
       for i from 2 to n{
           temp = fib;
           fib = fib + prevFib;
           prevFib = temp;
           print fib,"\n";
       }
   }
}
```

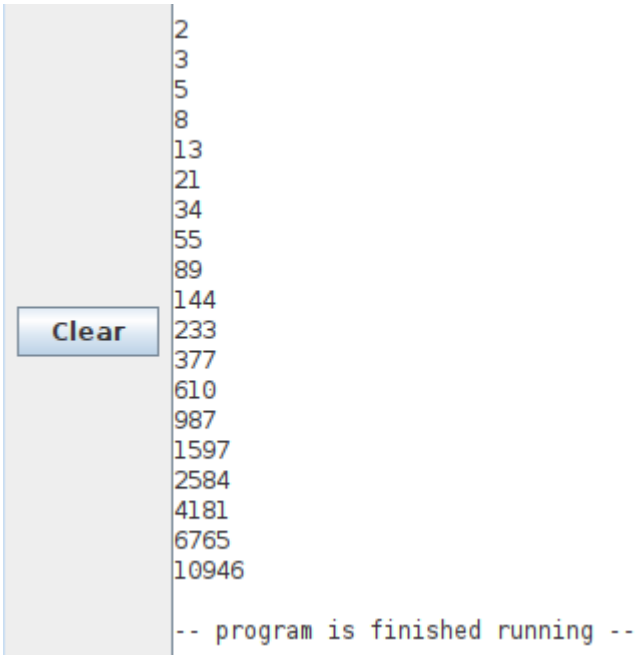
Código en MIPS

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

$str1:
   .asciiz "1"
$str2:
   .asciiz "\n"
_n:
   .word 0
_fib:
   .word 0
_prevFib:
   .word 0
_i:
   .word 0
_temp:
   .word 0
```

```
##################
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 20
    sw $t0, _n
    li $t0, 1
    sw $t0, _fib
    li $t0, 1
    sw $t0, _prevFib
    lw $t0, _n
    li $t1, 1
    sle $t0, $t0, $t1
    beqz $t0, $l3
    li $v0, 4
    la $a0, $str1
    syscall
    b $l4
$l3:
    li $t1, 2
    lw $t2, _n
$l1:
    bgt $t1, $t2, $l2
    sw $t1, _i
    lw $t3, _fib
    sw $t3, _temp
    lw $t3, _fib
    lw $t4, _prevFib
    add $t3, $t3, $t4
    sw $t3, _fib
    lw $t3, _temp
    sw $t3, _prevFib
    lw $t3, _fib
    li $v0, 1
    move $a0, $t3
    syscall
    li $v0, 4
    la $a0, $str2
    syscall
    addi $t1, $t1, 1
    b $l1
$l2:
```

```
$l4:

#################
# Final: exit
   li $v0, 10
   syscall
```

Ejecución del programa en MARS



9. Conclusiones

El proyecto en general ha sido muy interesante y me ha ayudado a comprender el funcionamiento por dentro de un compilador. La parte que más tiempo nos ha llevado probablemente ha sido la generación de código, pero una vez que hemos entendido cómo funciona hemos podido avanzar a buen ritmo.

Una vez acabada la práctica, pese a las dificultades que nos han surgido, nos ha resultado muy didáctica y enriquecedora.

En cuanto a las mejoras opcionales, la parte que más ha costado ha sido la de los operadores relacionales, además de su integración en los bucles for.