MiniC Compiler Memory

Members:
Daniel Pérez Gómez
Victor Carrillo Gil

INDEX

Project explanation

The developed compiler consists of different parts.**miniC.l:** Contains the lexical **analysis**, that is, defines the tokens of the miniC language, all the operators that will be used later in the**miniC.y**, along with declarations of NUM and ID with their respective maximum ranges. Also identifiers, keywords, etc.

The most notable thing about the lexicon in our case could be the statements of:
Panic mode `([^A-Za-z0-9_\+\-\/\$\*{}()<>!=;,"\[\?\:\n])*`
`error();`

This line calls the error() function whenever it encounters a character that is **not found in** the specified list. The error() function displays the character that is causing the error:`printf("Error in character: %s\n", yytext);`

Other expressions of great importance are:`\"([^"\n\\]|\\[ntr"])*\" {`
`yylval.cadena = strdup(yytext); return STRING;}`
This expression recognizes **valid text strings enclosed** in double quotes (“”) accepting valid escape sequences such as`\n, \t, \r.`

In the miniC.y document,which includes both syntactic and semantic analysis. Regarding syntactic analysis, the following functions can be highlighted:**Definition of grammar of** language (syntactic rules),**declaration of variables and constants**, **analysis of arithmetic expressions and assignments**, **flow control with if and while structures**, **input/output with read and print**, **semantic error handling**(for example, undeclared variables or attempted modification of constants) and intermediate **code generation**, using a structure called Operation.

Data structures

1. **Table of symbols (`tableSymbols`)**
   Structure that stores information about the declared identifiers (name, type, whether it is constant or variable, etc.).

   It is used to store the elements necessary to carry out operations, since it is responsible for storing the aforementioned.

2. **Code list (`LC`)**
   Stores the assembly code generated for each statement, such as assignments, arithmetic operations, conditional jumps, etc.

   This is the most important data structure in the code since it is responsible for storing the assembler result of each element of the code to be translated.

WARNING SHIFT/REDUCTION

miniC.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
We are aware of this warning, it is acceptable since we know the reason why it occurs, which is:
This conflict occurs when Bison encounters a nested if-else structure, and cannot immediately determine which if corresponds to the else.

Main functions

getLabel()

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

Generates a unique label (for example: `$label0`, `$label1`, etc.) which can be used to mark places as jump destinations.

- Use a global counter`counterTags`to ensure that each label is unique.

nextRecord()

```
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");
```

```
}
```

Keep track of availability using the table`tablaRegistros`.

- If it finds a free register, it marks it as busy and returns it.
- If none are available, it reports with an error message.

releaseRegister()

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

Frees a temporary record that is no longer needed, marking it as available in the table.

- deer `atoi(register + 2)` to extract the number from the record, assuming the format`$tX`.

initializeRegTable()

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

Initializes the record table, marking all records as free.`(0)`.

yyerror()

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

Syntax error handler, prints a message indicating the current token `(yytext)` and the line of error `(yylneno)` .

imprimeLC() //In listCode.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");
}
```

Function responsible for displaying on screen the translation performed by the compiler. Prints the mips code corresponding to the provided miniC input

imprimeLS() // In listSymbols.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);
    }

}
```

It is responsible for printing the data **section** `(.data)` from the assembly code generated by the compiler.

Examples of program operation

Entrance to try the**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);
}
```

Output obtained after running miniC:

```
.data
_c:                                     .word 0

_d:                                     .word 0

_and:                                   .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
```

```
        that $t1,1
        sub
        $t0,$t0,$t1 sw
        $t0,_c
        like
        $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
##############
# END
        like $v0, 10
        syscall
```

Output after running the assembler:

```
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
and 6
=
7
14
and 7
=
8
16
and 8
=
9
18
and 9
=
```

Explanation: With this entry we are testing the operation of the While of read() and finally of the Do While.

Entrance to try the**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");



}
```

Output obtainedafter running 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 to
       $a0,$str1
       like
       $v0,4
       syscall
       la
       $a0,$str2 li
       $v0,4
       syscall
       b $label3
$label2 :
       lw $t1,_a
       beqz
       $t1,$label1 to
       $a0,$str3
       like
       $v0,4
       syscall
$label1 :
$label3:
       lw $t0,_a
       beqz
       $t0,$label4 to
       $a0,$str4
       like
       $v0,4
       syscall
       la
       $a0,$str5 li
       $v0,4
       syscall
$label4 :
#############
# END
       like $v0, 10
```

```
syscall
```

Output after running the assembler:

a
Final

Entrance to try **the For and more complex in 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" );
}
```

Output obtained after running miniC:
.data
_n:
    .word 0
_addition:
    .word 0
_adding:
    .word 0
_i:
    .word 0
$str1:
    .asciiz "Enter the number of addends: \n"
$str2:
    .asciiz "Enter a summand:\n"
$str3:
    .asciiz "The total sum is: "
$str4:
    .asciiz "\n"
.text
.globl
main
main:
    li $t0,0
    sw $t0,_sum
    to $a0,$str1

```
        as
        $v0,4
        syscall
        as
        $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
        as
        $v0,4
        syscall
        as
        $v0,5
        syscall
        sw $v0,_sumand
        lw $t2,_sumand
        lw $t3,_sum
        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
#############
# END
        like $v0, 10
```

syscall

Output after running the assembler:`Enter the number of addends: 6`

```
Enter an addend:
2
Enter an addend:
3
Enter an addend:
4
Enter an addend:
5
Enter an addend:
6
Enter an addend:
7
The total sum is: 27
```

Proposed entry in the Virtual Classroom

```
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");
}
```

output generated when running miniC:

.data

_a:

.word 0

_b:

.word 0

_c:

.word 0

```
$str1:
        .asciiz "Program Start\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 to
       $a0,$str2
       like
       $v0,4
       syscall
       la
       $a0,$str3 li
```

```
	$v0,4
	syscall
	b $label6
$label5 :
	lw $t1,_b
	frog $t1,$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
##############
# END
    like $v0, 10
    syscall
```

Output when running the assembler:

```
Start of program 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
```

Extensions developed

Do While:

The first thing we did to complete this extension was to declare the “DO” token in the miniC.l since, unlike the “While” token, we did not have it yet.

do return DO;

Later in miniC.y we add the token we just generated to the list of tokens:

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

Finally, the production rule and logic to produce code had to be created:

DO statement WHILE PARI expression PARD FIN {

```
 ListC lc = creaLC();
        char *startLabel = getLabel(); Operation op;
        op.op = etiquetaInicio;
        op.arg1 = NULL;
        op.arg2 = NULL;
        op.res = ":";
        insertaLC(lc, finalLC(lc), op);
```

```
    // 2. Block code (statement)

    concatenaLC(lc, $2);


    // 3. Condition code (expression)

    concatenaLC(lc, $5);


    // 4. Conditional jump (if the condition is true, return to start) Operation
    opJump;
    opJump.op = "bnez"; // Jump if NOT zero opJump.arg1
    = startLabel;
    opJump.arg2 = NULL;
    opJump.res = recuperaResLC($5);
    insertaLC(lc, finalLC(lc), opJump);

    // 5. Release record from the condition
    releaseRecord(recoverResLC($5));


    $$ = lc;
}
```

This code does the following:

A start tag.

Instruction block code. Condition

evaluation.

Conditional jump to the label if the condition is met.

Extension of relational operators: (<, >, <=, >=, ==, !=)

For this case, as in DO WHILE, it is necessary to add the tokens for each operator:

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

When adding the tokens to miniC.y we also have to keep in mind the precedence that these operators must have, so we declare them at this point so that the rest of the operations are always performed before making the comparison:

```
%left GT LT GTE LTE DIFF EQ
%left QUESTION DPTOS
%left PLUS LESS
%left POR DIV
%left UMENOS
```

Finally, we add the production rules for each operator. This is simple since they follow a logic very similar to the operators already implemented:

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

      ListC 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);

      ListC lc = $1;

      Operation on;
      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);

        ListC 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);

        releaseRegister(op.arg2); // Release the register 'a'


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

        ListC 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);
```

```
releaseRegister(op.arg2); // Release the register 'a'
```

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

        Op operation;

        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);
        ListC lc = $1;

        Op operation;

        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;
    }
```

The production rules contain a new operation depending on their equivalence with those of MIPS.

The implementation of a for statement:

For the last extension, as in the previous ones, the first thing is to declare the new “For” token in the miniC.l:

for return FOR;

Once this is done we must add the token to the miniC.y token list:

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

Finally, the most complicated part remains: designing the production rule and the code responsible for generating the assembler:

| FOR PARI ID ASIG expression FIN expression FIN ID ASIG expression PARD statement {

```
ListC lc = creaLC();

    ListPosition s = searchLS(symbolsTable, $3);
    if (s == finalLS(tablaSimbolos)) {printf("Undeclared variable %s \n",$3); errors
    = errors + 1;}
    else if (recuperaLS(tablaSimbolos, s).tipo == CONSTANTE){errors = errors + 1;
    printf("Assignment to constant\n");}

    ListPosition s2 = searchLS(symbolsTable, $9);
    if (s2 == finalLS(tablaSimbolos)) {printf("Undeclared variable %s\n", $9); errors
    = errors + 1;}
    else if (recuperaLS(tablaSimbolos, s2).tipo == CONSTANTE) {errors = errors + 1;
    printf("Assignment to constant\n");}

    // 2. Initialization code concatenaLC(lc,
    $5);
    char varName[20]; sprintf(varName,
    "_%s", $3); Operation opInit;
    opInit.op = "sw";
    opInit.arg1 = strdup(numberVar);
    opInit.arg2 = NULL;
    opInit.res = recuperaResLC($5);
    insertaLC(lc, finalLC(lc), opInit);

    freeRegistry(recoverLCRes($5));

    // 3. Loop start label
    char *startLabel = getLabel(); etStart
    operation;
    etInicio.op = etiquetaInicio;
    etInicio.arg1 = NULL;
    etInicio.arg2 = NULL;
```

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

// 4. Condition code concatenatedLC(lc,
$7);


// 5. End of loop label
char *endLabel = getLabel();

// 6. Jump if condition is false
Operation opCond;
opCond.op = "beqz";
opCond.arg1 = EndLabel;
opCond.arg2 = NULL;
opCond.res = recuperaResLC($7);
insertaLC(lc, finalLC(lc), opCond);

freeRegistry(recoverLCRes($7));

// 7. Loop body code concatenaLC(lc,
$13);

// 8. Increment code concatenaLC(lc,
$11);
char varName1[20];
sprintf(varName1, "_%s", $9);
Operation opInc;
opInc.op = "sw";
opInc.arg1 = strdup(VarName1);
opInc.arg2 = NULL;
opInc.res =
recuperaResLC($11);
insertaLC(lc, finalLC(lc), opInc);

freeRegistry(recoverLCRes($11));

// 9. Jump to start
Operation opJump;
opJump.op = "b";
opJump.arg1 = NULL;
opJump.arg2 = NULL;
opJump.res = startLabel;
insertaLC(lc, finalLC(lc), opSalto);

// 10. End label
Operation etFin;
etFin.op =
etiquetaFin;
etFin.arg1 = NULL;
```

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

$$ = lc;

EXPLANATION
An empty list is created to store all the code generated for the`for`.`ListPosition s = searchLS(symbolsTable, $3);`
```
if (s == finalLS(tableSymbols)) { }
else if (recuperaLS().tipo == CONSTANTE) { }

PosicionLista s2 = buscaLS(tablaSimbolos,

$9);
```
It is verified that the variables are declared.

It also ensures that they are not constant, since they cannot be assigned.
If there are errors, they are notified and the counter is incremented.`errores`.
```
concatenaLC(lc, $5);
sprintf(nombreVar, "_%s", $3);
opInit.op = "sw";
opInit.arg1 = strdup(Var number);
opInit.res = retrieveResLC($5);
```

The code of the initialization expression is concatenated
The instruction is generated `sw` to **store the value resulting** in the variable`$3`.

```
char *startLabel = getLabel();
```

A label is generated for the start.

```
opCond.op = "beqz";     // jump if condition == 0 (false)
opCond.res = recuperaResLC($7);
opCond.arg1 = EndLabel;
```
A loop exit label is created.

If the condition is false`(== 0)`, a jump is made to that label.

```
concatenaLC(lc, $13);
```
The code of the instruction block is concatenated within the loop.

```
concatenaLC(lc, $11);
opInc.op = "sw";
opInc.arg1 = strdup(numberVar1); // number of $9
opInc.res = retrieveResLC($11);
```

The code that evaluates the increment expression is concatenated.
Then an instruction is generated `sw` to save the result in the variable`$9`. `opSalto.op = "b";`

```
opSalto.res = startLabel;
```

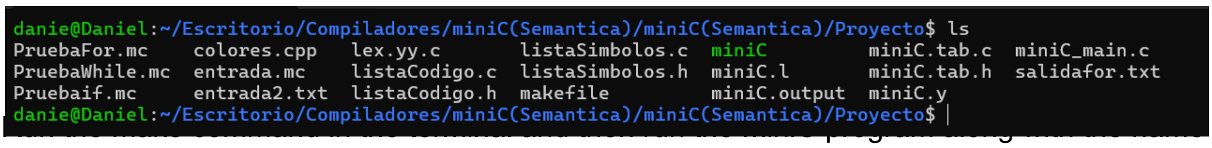
Jump to the beginning of the loop.

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

The end of the loop is marked with a label, so that `frog` jump here if the condition fails.

User manual.

To use the compiler, we first check that we have all the necessary files in the same folder:



of the input you want to pass to it (in this example case):

```
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
```

(the code does not appear in its entirety since it does not fit in a single screenshot)
Once executed, as can be seen in the image, the program will generate, if the input is valid, an assembly code corresponding to the translation of the C code passed in as input. In this case, it was:

```
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);
}
```

There are two ways to run the assembly code, the first is by copying the output and pasting it into mars to run, or also by redirecting the terminal output to a file, and then choosing to run it using spim, or mars itself by opening this new file:

```
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
```

CONCLUSION

We found this practical exercise to be a very useful way to apply what we've learned in theory and, at the same time, to stop viewing a compiler as something abstract, and instead understand what's happening thanks to it. It also helps us better understand how to leverage memory and how operations work even beneath the assembly code, examining its vocabulary, syntax, and semantics, all the way to code generation.

The only drawback, if I may say one, is that the course schedule requires the development of three projects at the same time, which can result in, and does result in, a workload that sometimes forces you to leave the theory aside a bit. As for the project, we are very satisfied with its development and with seeing that the work is achieving results as interesting as this compiler.