COMPILADORES

Departamento de
Ingeniería de la Información
y las Comunicaciones

CONVOCATORIA DE JULIO
2017

ALUMNO: MOURAD ABBOU AAZAZ

E-MAIL: mourad.abbou@um.es

GRUPO: 2

PROFESOR : EDUARDO MARTINEZ GRACIA

ÍNDICE GENERAL

DEFINICIÓN INTRODUCTORIA DE LA PRÁCTICA

ANÁLISIS LÉXICO

ANÁLISIS SINTÁCTICO

ANÁLISIS SEMÁNTICO

GENERACIÓN DE CÓDIGO

ESTRUCTURAS DE DATOS

**EXPLICACIÓN DE FUNCIONAMIENTO
DE LOS BUCLES Y SENTENCIAS ANIDADAS**

ESTRUCTURAS DE DATOS
- **TABLA DE SÍMBOLOS**
- **CUADRUPLAS**
- **LISTA DE CÓDIGO**

MANUAL DE USUARIO

MEJORAS

DEFINICIÓN INTRODUCTORIA DE LA PRÁCTICA

El objetivo de esta práctica es el diseño y la implementación de un compilador con una sintaxis reducida del lenguaje C, mediante una serie de fases y herramientas para los distintos análisis que se realizan en el proceso de compilación.

El compilador , realizado en las sesiones de prácticas y durante el trabajo autónomo, comprende una serie de fases que van desde la definición de identificadores y lexemas que constituyen la sintaxis y hasta la traducción de sentencias al lenguaje ensamblador.

La realización de esta práctica tiene las siguientes fases que se especificarán más adelante, el resto del informe:

- ***ANÁLISIS LÉXICO*****:** Definición de lexemas, tokens, identificadores, expresiones regulares que definen letras, dígitos y cadenas, y palabras claves para las sentencias.

- ***ANALISIS SINTÁCTICO*****:** Reconocimiento sintáctico de las sentencias y orden de precedencia, así como asociatividad y eliminación de las sentencias ambiguas. También se tratan conflictos y errores.

- ***ANÁLISIS SEMÁNTICO*****:** Comprobación de las variables declaradas, así como constantes y cadenas declaradas en el programa. También el procesamiento y coherencia de las variables, y la comprobación de que las variables no se declaren mas de una vez.

- ***TRADUCCIÓN DE CÓDIGO*****:** Generar el código en ensamblador de las sentencias tras completar los análisis.

Para llevar a cabo todo el proceso de análisis y traducción vamos a necesitar estructuras de datos para el almacenamiento de variables en el análisis semántico y para la traducción de sentencias en ensamblador.
La especificación de las estructuras de datos, se harán más adelante.

ANÁLISIS LÉXICO

Para el diseño del compilador, es necesario declarar y especificar un conjunto de lexemas conocidos como tokens, identificadores y palabras claves.

En esta práctica se usa un fichero cuya extensión es .l, y se llamará ***miniC.l.***
En ella se especifican cada una de las palabras clave, tokens, expresiones regulares y funciones que controlen los errores.

Para llevar a cabo el análisis se hace uso de la herramienta Flex, necesaria para este análisis.

El código definido en el fichero fuente .l se compone de tres partes:

- **Cabereca**: Consiste en el nombre del programa que se realiza.
  Esta incluye el tipo de programa y las funciones que requiere o que necesita el tipo que devuelve.
  Definimos un dígito que es número cuyo rango está entre 0 y 9.
  A continuación definimos un carácter que puede ser una letra de la A a la Z independientemente de si es mayúscula o minúscula.

- **Declaraciones**: Se declaran variable, constantes, palabras reservadas, símbolos y operadores que se utilizaran para el compilador. También se incluyen comentarios, recolector de basura, y métodos que se especifican en la parte de sentencias en C.

- **Sentencias**: Aquí se implementan los métodos y funciones que devuelve algunas variables para el control y notificación de errores.

cabecera

```
"//".*    {}

({letra}|_)({letra}|_|{digito})*  {

      if (yyleng > 16){
          printf("Error: EL identificador es demasiado grande \n");
          yytext[16] = '\0';
       }
      yylval.str = strdup(yytext);
      return ID;
}

[^a-zA-A0-9_/\- \n\t\r(){}**/=,;"]+ { printf("Carácter no válido: %s\n",yytext); }



"/*"              BEGIN comentario;
<comentario>[^*]*   ;
<comentario>[^]+[^/^]*     ;
<comentario><<EOF>>   {return comentario_retorno();}
<comentario>[*]+[/]   BEGIN 0;
```

Declaraciones

```
%{
// Código C
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
// Deficnición de códigos de tokens
#include "miniC.tab.h"
#include "traductor.h"
void comprobar_longitud();
int comentario_retorno();
%}
digito [0-9]
letra [a-zA-Z]
entero [0-9]+



%x comentario
%x cadena
%%
```

sentencias

```
func   return FUNC;
var    return VAR;
let    return LET;
if     return IF;
else   return ELSE;
while  return WHILE;
print  return PRINT;
read   return READ;
do     return DO;
for    return FOR;
";"    return PUNTOCOMA;
","    return COMA;
"+"    return SUMA;
"-"    return MENOS;
"*"    return ASTERISCO;
"/"    return CONTRABARRA;
"="    return IGUAL;
"("    return PARIZQUIERDO;
")"    return PARDERECHO;
"{"    return LLAVEIZQUIERDA;
"}"    return LLAVEDERECHA;
"&&"   return AND;
"||"   return OR;
"!="   return DISTINTO;
">"    return GREATERTHAN;
"<"    return LESSERTHAN;
">="   return GTOREQ;  // greater or equal
"<="   return LSSROREQ;  // lesser or equal
"=="   return IGTG;      // equalequal
\"([^\n"]|\\\")*\" {

  yylval.str = strdup(yytext);
  return CADENA;
}

{entero} {
  comprobar_longitud();
  return NUM;
}
```

```
void comprobar_longitud() {
 double l = atol(yytext);
 if (l > pow(2,31)) {
  printf("el numero %f es demasiado grande", l);
 }
 yylval.str = strdup(yytext);
}


int comentario_retorno(){
  return 0;
}
```

ANALIZADOR SINTÁCTICO

En esta fase de análisis, como se vió en teoría, se crea un árbol sintáctico ascendente donde cada nodo hoja es un token o una variable no terminal. Para llevar a cabo este proceso utilizamos la herramienta Bison, un programa que construye el árbol a partir de los nodos hoja que hayamos definido en en nuestras reglas de producción. El análisis se implementa en un fichero llamado miniC.y
En primer lugar declaramos los tokens y las expresiones regulares que hemos generado en el
analizador léxico.

Después construimos un conjunto de procedimientos, uno por cada no terminal.
Para cada procedimiento se determina las acciones a aplicar.
A continuación implementamos las reglas de producción que establece la practica. En parte superior del fichero, se ve como se declaran los tipos de cada regla producción, algunos son de tipo lista o `listaCodigo`, que son las estructuras de datos que se han definido y se especifican más adelante en el informe. En la herramienta de **Bison**, se generan dos ficheros *miniC.tab.h* y *miniC.tab.c* que contienen el código del ***parsing***.

ANÁLISIS SEMÁNTICO

En el análisis sintáctico, hacemos la construcción del árbol sintáctico.
En el análisis semántico tenemos que tener todas nuestras variables declaradas almacenadas en una estructura de datos. Para ello vamos a crear una tabla de símbolos que en la que se inserten todas las variables, contantes, identificadores y cadenas que vayamos a usar en nuestro programa. Así se puede comprobar que no se usa variable no declarada y que una variable no se declare más de una vez. En el análisis sintáctico, al llegar a la regla de producción *VAR* y *LET*, se insertan las variables declaradas en la tabla. Los variables tipo LET no pueden ser modificadas una vez establecidas.

GENERACIÓN DE CÓDIGO (FASE OPCIONAL)

La última fase consiste en ir recorriendo el árbol creado en la fase anterior para ir generando el código ensamblador como resultado de la traducción de las sentencias. En esta fase se usa la semántica de los atributos para ir conectando las entradas y salidas de las instrucciones generadas. Cada instrucción de tres direcciones contiene por lo menos un operador, además de la asignación. Cuando se generan estas instrucciones, el traductor tiene que decidir el orden en que deben
efectuarse las operaciones. Para ello hemos creado un fichero que usa una estructura de datos llamada **cuádrupla**, en la que se
recogen el tipo de operación que se quiere realizar y los registros utilizados de cada instrucción. Si para una instrucción se necesita menos de tres registros, los campos restantes quedarán vacíos, inicializados en la cuádrupla a NULL. Un claro ejemplo de esto son las cuádruplas que generan etiquetas, ya que solo necesitamos el tipo de operación (primer atributo) por lo que los otros tres restantes los inicializamos a NULL.
También hacemos uso de la tabla de símbolos del análisis semántico para imprimir las variables involucradas. En el fichero de análisis sintáctico ***miniC.y*** se implementan, por cada regla de producción, las declaraciones y manejo de las listas de código.

ESTRUCTURAS DE DATOS

En esta práctica se han creado dos estructuras de datos:

- **TABLA DE SÍMBOLOS :** Es una lista enlazada que va insertando los símbolos y variables del análisis semántico.
  Está formado por un nombre, un valor que indica si es una cadena, una variable o una constante, y una cadena ***str*** que almacena un la etiqueta que va a ser utilizada en el lenguaje ensamblador.
Se ha creado un fichero llamado lista.c y otro fichero cabecera lista.h, ambos proporcionados por el profesor.

```
struct listaRep {
    char *nombre;
    char *str;          // contiene la variable en ensamblador
    int valor;
    struct listaRep *sig;
};

typedef struct listaRep * lista;
```

`lista crearLista();` Incializa la lista.

`void borrarLista(lista l);` Elimina la lista y libera memoria.

`void insertarVariable(lista *l, char *nombre, int valor);`
Esta función inserta la variable en la lista. Se le pasa como parámetro un valor que indica si es una cadena, una variable o una constante.
Si la variable no existía anteriormente, se recorre la lista hasta encontrar una posición nula donde insertarla. Si el tipo es igual a 3, se trata de una cadena y se inserta en el campo ***str*** la variable de la cadena, en caso contrario se inserta el identificador de la variable o constante. El campo ***str*** se utilizará para el código ensamblador.

`int consultarVariable(lista l, char *nombre);` Devuelve un número mayor que 0 en caso de que una variable nombre, se encuentre dentro de una lista l.

`char* getLabel(lista t, char *cadena)`: Retorna el campo ***str*** que contiene la etiqueta de una variable en una lista.

`char* barra(char *cadena)`: Devuelve el identificador del nombre de una variable pero con una barra baja insertada al principio.

`void imprimirVariables(lista t)`: Imprime todas variables que hay dentro de una lista t.

`char* str_label()`: Crea una cadena con una cadena “$str” con el número de etiqueta disponible.

- CUÁDRUPLAS

Para la traducción de sentencias a ensamblador, vamos a crear una estructura de datos llamada cuádrupla, que representa una operación con registros en lenguaje ensamblador. Esta estructura contiene:

- `op` : una operación en ensamblador.
- `destino`: registro destino donde se almacena la operación.
- `arg1`: registro implicado.
- `arg2`: registro implicado.

Existen sentencias en ensamblador en las que no participan todos los registro, por lo que pueden haber argumentos nulos.

- LISTA DE CÓDIGOS

Para hacer la traducción de todas las sentencias, enlazarlas y unirlas con las variable de la tabla, vamos a crear una estructura de datos que es una lista enlazada con dos cuádruplas principales, una llamada primera que contiene la primera sentencia en ensamblador y otra última que contiene la última sentencia.
Se ha creado un fichero llamado traductor.c y otro traductor.h cabecera.
El fichero contiene la estructura de datos de una cuádrupla y una estructura de una lista de cuádruplas.
En este fichero también se han incluido algunas variables como un array de registro y un contador que es utilizado para crear etiquetas.

La funcionalidad de la lista de código es:

`char* crear_etiqueta()`: Crea una etiqueta para ensamblador.

`void insertarCuadrupla(Cuadrupla cuádrupla, ListaCodigo listaCodigo)`: Esta función inserta el una cuádrupla que representa una sentencia dentro de una lista de cuádruplas. Si la última cuádrupla a la que apunta la lista es nula, entonces se insertará en ella la cuádrupla que se pasa como parámetro.

`void enlazarlistaCodigos(ListaCodigo principal, ListaCodigo t)`: Esta función sirve para enlazar dos listas de código. Si la lista principal está inicialmente vacía, apuntará a lista t. En otro caso, la última cuádrupla de la lista principal apuntará a la lista t.

`void liberarlistaCodigo(ListaCodigo t)`: Esta función elimina una lista t, y libera la memoria ocupada.

`void imprimirlistaCodigo(ListaCodigo t)`: Imprima todo el contenido de una lista t.

`void imprimirlistaCodigos(ListaCodigo t1, ListaCodigo t2)`: Imprime todo el contenido de una lista t1, y de una lista t2.

`char *registro_actual()`: Esta función devuelve el registro actual disponible en la lista de código.

`void eliminarRegistro(ListaCodigo listaCodigo)`: Libera el registro de la última cuádrupla de una lista de códigos.

EXPLIACIÓN DE FUNCIONAMIENTO DE LOS BUCLES Y SENTENCIAS ANIDADAS

En el fichero miniC.y hemos implementado las sentencias anidadas IF, IF-ELSE y WHILE establecidas en la practica. A continuación vamos a explicar como se han traducido estas sentencias en ensamblador, para aclarar su funcionamiento.

Sentencia IF:

Comenzamos creando una etiqueta que indique el comienzo de la sentencia IF . Creamos una cuádrupla que va a contener la etiqueta y otra que contiene la sentencia de salto *beqz*.
Inicializamos nuestra lista de código y la enlazamos con la lista de código del símbolo no terminal. Tras insertar todo el código de la expresión insertamos la cuádrupla de la sentencia de salto salto. Después insertamos el código de statement y la cuádrupla de la etiqueta. Eliminamos el registro que ha tomado la expresión y lo dejamos libre.

```
IF PARIZQUIERDO expresion PARDERECHO statement {
       char *etiqueta = crear_etiqueta();
       Cuadrupla c1 = nueva_cuadrupla(etiqueta, NULL, NULL, NULL);
       Cuadrupla c2 = nueva_cuadrupla("beqz ", NULL, $3->primera->destino, c1->op);
       $$ = nueva_listaCodigo();
       enlazarlistaCodigos($$, $3);                                    // expresion
       insertarCuadrupla(c2, $$);                                      // BEQZ
       enlazarlistaCodigos($$, $5);                                    // statement
       insertarCuadrupla(c1, $$);                                      // ETIQUETA
       eliminarRegistro($3);
}
```

Sentencia IF-ELSE:

Esta sentencia es igual que la la sentencia IF solo que tiene una etiqueta a la que salta la ejecución si no se cumple la sentencia *beqz*. También se ha creado una cuádrupla con la sentencia branch, que salta a la parte del código en la que no se cumple la sentencia de la expresión.

```
IF PARIZQUIERDO expresion PARDERECHO statement ELSE statement {

Cuadrupla c1 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL);
Cuadrupla c2 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL);
Cuadrupla c3 = nueva_cuadrupla("beqz ", NULL, $3->primera->destino, c1->op);
Cuadrupla c4 = nueva_cuadrupla("b ", NULL, c2->op, NULL);
$$ = nueva_listaCodigo();        // creamos la lista de codigo
```

```
enlazarlistaCodigos($$, $3);                                    // enlazamos con la expresion
insertarCuadrupla(c3, $$);                                      // insertamos la cuadrupla de
beqz en $$
enlazarlistaCodigos($$, $5);                                    // enlazamos  $$ con el
primer STATEMENT
insertarCuadrupla(c4, $$);                                      // insertamos el branch en $$
insertarCuadrupla(c1, $$);                                      // insertamos la cuadrupla 1
en $$
enlazarlistaCodigos($$, $7);                                    // enlazamos
insertarCuadrupla(c2, $$);                                      // insertamos
eliminarRegistro($3);            // eliminamos registro
}
```

SENTENCIA WHILE:

Las cuádruplas necesarias para implementar el bucle while, son las mismas que en la sentencia IF-ELSE.
Insertamos la primera etiqueta, enlazamos con la expresion, insertamos la cuadrupla de beqz, enlazamos con statement, insertamos la sentencia branch e insertamos la etiqueta de salida del bucle while.

```
WHILE PARIZQUIERDO expresion     PARDERECHO statement {
Cuadrupla c1 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL);
Cuadrupla c2 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL);
Cuadrupla c3 = nueva_cuadrupla("beqz ",  NULL, $3->primera->destino, c2->op);
Cuadrupla c4 = nueva_cuadrupla("b ", NULL, c1->op, NULL);

$$ = nueva_listaCodigo();
insertarCuadrupla(c1, $$);                                          // ETIQUETA
enlazarlistaCodigos($$, $3);                                        // expresion
insertarCuadrupla(c3, $$);                                          // BEQZ
enlazarlistaCodigos($$, $5);                                        // statement
insertarCuadrupla(c4, $$);                                          // B
insertarCuadrupla(c2, $$);                                          // ETIQUETA
eliminarRegistro($3);                                                      //
}
```

MANUAL DE USUARIO

El compilador implementado tiene una sintaxis reducida del lenguaje C, en la que solo existe una función, que es ***func*** donde se encuentra el programa principal. Debemos tener en cuenta que este lenguaje (miniC) carece de operadores lógicos.

La declaración de variables **var** y constantes **let**, se hacen al principio del programa donde donde se tienen que inicializar. Recordemos que las constantes no se pueden modificar. Permite solamente valores primitivos de tipo int, y boolean. Este último es una mejora implementada que se especifica en el apartado de Mejoras.
Dentro del cuerpo del programa podrás inicializar tus variables, implementar bucles IF, IF-ELSE, y WHILE. Ademas, se han implementado la sentencia DO WHILE que se especifica en el apartado de Mejoras.
Se pueden imprimir variables declaradas y texto con la sentencia `print` “`<cadena>`”.

Tras crear nuestro codigo, hacemos lo siguiente:

- Abrimos la terminal.
- Vamos al directorio donde tenemos nuestro compilador y escribimos make en caso de que no esté construido el compilador.
- escribimos *`./scanner <nombreFichero>.c > nombreFichero.s`*
- Compilamos el fichero.s en algún IDE de MIPS.
- Ejecutamos.

```
func prueba () {
let a=6, b=0;
var c=5+7-2;
print "Inicio del programa\n";
if (a) print "a","\n";
  else if (b) print "No a y b\n";
      else while (c) {
          print "c = ",c,"\n";
          c = c-2+1;
      }
  print "Final","\n";
}
```

```
eduroam_alu-83-187:miniC morcd$ make
make: `scanner' is up to date.
eduroam_alu-83-187:miniC morcd$ ./scanner prueba.c > fileprueba.s
eduroam_alu-83-187:miniC morcd$
```

```
# Seccion de datos
        .data

$str6:
        .asciiz "Final"
$str5:
        .asciiz "c = "
$str4:
        .asciiz "No a y b\n"
$str3:
        .asciiz "\n"
$str2:
        .asciiz "="
$str1:
        .asciiz "Inicio del programa\n"
_c:
        .word 0
_b:
        .word 0
_a:
        .word 0

#########################
# Seccion de codigo
    .text

    .globl main
main:
    li  $t0, 0
    sw  $t0, _c
    li  $t0, 0
    sw  $t0, _b
    li  $t0, 3
    li  $t1, 2
    add  $t2, $t0, $t1
    li  $t0, 2
    sub  $t1, $t2, $t0
    sw  $t1, _c
    la  $a0 , $str1
    li  $v0 , 4
    syscall
    lw  $t0, _c
    beqz  $t0, $15
    la  $a0 , $str2
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
    b  $16
$15:
    lw  $t1, _b
    beqz  $t1, $13
    la  $a0 , $str4
    li  $v0 , 4
    syscall
    b  $14
$13:
$11:
    lw  $t2, _c
    beqz  $t2, $12
    la  $a0 , $str5
    li  $v0 , 4
    syscall
    lw  $t3, _c
    move  $a0 , $t3
    li  $v0 , 1
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
    lw  $t3, _c
    li  $t4, 2
    sub  $t5, $t3, $t4
    li  $t3, 1
    add  $t4, $t5, $t3
    sw  $t4, _c
    b  $11
$12:
$14:
$16:
    la  $a0 , $str6
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall

#########################
# Fin
        jr $ra
```

Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
Final

MEJORAS

Tras hacer implementar los tres análisis, se han hecho una serie de mejoras opcionales para el compilador tales como:

- **TRADUCCIÓN DE SENTENCIAS A CÓDIGO ENSAMBLADOR**: La explicación está arriba.

- **IMPLEMENTACIÓN DE OPERADORES LÓGICOS Y BOOLEANOS:**

Se han dispuesto a implementar operadores lógicos y booleanos. Para ello tenemos que definir nuevos tipos en nuestro análisis léxico y crear nuevas reglas de producción para el análisis sintáctico.
En nuestro fichero miniC.l hemos implementado los operadores:

"&&" AND:
"||" OR:
"!=" DISTINTO:
">" GREATERTHAN:
"<" LESSERTHAN:
">=" GTOREQ: Greater or Equal
"<=" LSSROREQ: Lesser or Equal
"==" IGIG: Igual Igual

En el fichero miniC.y hemos implementado la variable no terminal **boolean**, que deriva en las siguientes regla de producción:

```
boolean:    expresion IGIG expresion {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("seq", registro, $1->ultima->destino, $3-
      >ultima->destino);
      eliminarRegistro($1);
      eliminarRegistro($3);

}
| expresion GTOREQ expresion{
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("sge", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
```

```
      eliminarRegistro($3);
}

| expresion LSSROREQ expresion {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("sle", registro,$1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
}
| expresion GREATERTHAN expresion {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("sgt", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
 }
| expresion LESSERTHAN expresion {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("slt", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
 }
| expresion DISTINTO expresion {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("sne", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
 }
| boolean AND boolean {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
```

```
      Cuadrupla c1 = nueva_cuadrupla("and", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
 }
| boolean OR boolean {
      $$ = nueva_listaCodigo();
      char* registro = registro_actual();
      enlazarlistaCodigos($$, $1);
      enlazarlistaCodigos($$, $3);
      Cuadrupla c1 = nueva_cuadrupla("or", registro, $1->ultima->destino, $3-
      >ultima->destino);
      insertarCuadrupla(c1, $$);
      eliminarRegistro($1);
      eliminarRegistro($3);
 }
| PARIZQUIERDO boolean PARDERECHO {
      $$ = nueva_listaCodigo();
      enlazarlistaCodigos($$, $2);
      $$->primera->destino = $2->primera->destino;
      $$->ultima->destino = $2->ultima->destino;
}
| expresion {$$ = $1;}
```

- Implementación de un bucle DO-WHILE.

```
DO statement WHILE expresion PUNTOCOMA {

      Cuadrupla c1 = nueva_cuadrupla(crear_etiqueta(),NULL,NULL,NULL);
      Cuadrupla c2 = nueva_cuadrupla("bnez", $4->primera->destino, c1-
      >op, NULL);
      $$ = nueva_listaCodigo();
      insertarCuadrupla(c1, $$);                //etiqueta
      enlazarlistaCodigos($$, $2);              //statement
      enlazarlistaCodigos($$, $4);              //expresion
      insertarCuadrupla(c2, $$);                //bnez
      eliminarRegistro($4);                     //eliminar exp
}
```

La implementación del bucle do while trata de crear una etiqueta y una cuádrupla de sentencias. A diferencia del bucle WHILE, en el bucle DO-WHILE se inserta primero el código de statement, y después se inserta el código de la sentencia de salto.

Ahora vamos a hacer una prueba con la mejor de boolean.
Ejecutamos la pruebaBoolean.c

```
eduroam_alu-78-35:miniC morad$ cat pruebaBoolean.c
func prueba () {
let a=0, b=0;
var c=5+2-2;
print "Inicio del programa\n";
if (a == 0) print "a","\n";
  else if (b) print "No a y b\n";
       else do  {
       print "c = ",c,"\n";
       c = c-2+1;
    } while (c != 0);
  print "Final","\n";
}
```

```
eduroam_alu-78-35:miniC morad$ make
bison -d  -t miniC.y
flex --yylineno miniC.l
gcc -g main.c lex.yy.c  miniC.tab.c lista.c traductor.c -ll -ln -o scanner
eduroam_alu-78-35:miniC morad$ ./scanner pruebaBoolean.c
```

fileBoolean.s

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

$str6:
        .asciiz "Final"
$str5:
        .asciiz "c = "
$str4:
        .asciiz "No c y b\n"
$str3:
        .asciiz "\n"
$str2:
        .asciiz "a"
$str1:
        .asciiz "Inicio del programa\n"
_c:
        .word 0
_b:
        .word 0
_a:
        .word 0

#####################################
# Seccion de codigo
    .text

    .globl main
main:
    li  $t0, 0
    sw  $t0, _a
    li  $t0, 0
    sw  $t0, _b
    li  $t0, 5
    li  $t1, 2
    add  $t2, $t0, $t1
    li  $t0, 2
    sub  $t1, $t2, $t0
    sw  $t1, _c
    la  $a0 , $str1
    li  $v0 , 4
    syscall
    lw  $t0, _a
    li  $t1, 0
    beqz  $t0, $l4
    la  $a0 , $str2
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
    b  $l5
```

Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
Final

Ahora probamos la mejora del DO-WHILE
Ejecutamos pruebaDOWHILE.c

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

$str6:
        .asciiz "Final"
$str5:
        .asciiz "c = "
$str4:
        .asciiz "No a y b\n"
$str3:
        .asciiz "\n"
$str2:
        .asciiz "a"
$str1:
        .asciiz "Inicio del programa\n"
_c:
        .word 0
_b:
        .word 0
_a:
        .word 0

################################
# Seccion de codigo
    .text

    .globl main
main:
    li  $t0, 0
    sw  $t0, _a
    li  $t0, 0
    sw  $t0, _b
    li  $t0, 5
    li  $t1, 2
    add  $t2, $t0, $t1
    li  $t0, 2
    sub  $t1, $t2, $t0
    sw  $t1, _c
    la  $a0 , $str1
    li  $v0 , 4
    syscall
    lw  $t0, _a
    li  $t1, 0
    beqz  $t0, $l4
    la  $a0 , $str2
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
```

```
    li  $t1, 0
    beqz  $t0, $l4
    la  $a0 , $str2
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
    b  $L5
$l4:
    lw  $t0, _b
    beqz  $t0, $l2
    la  $a0 , $str4
    li  $v0 , 4
    syscall
    b  $L5
$l2:
$l1:
    la  $a0 , $str5
    li  $v0 , 4
    syscall
    lw  $t1, _c
    move  $a0 , $t1
    li  $v0 , 1
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall
    lw  $t1, _c
    li  $t3, 2
    sub  $t4, $t1, $t3
    li  $t1, 1
    add  $t3, $t4, $t1
    sw  $t3, _c
    lw  $t1, _c
    li  $t3, 0
    sne $t4, $t1, $t3
    bnez $t1, $l1
$l3:
$l5:
    la  $a0 , $str6
    li  $v0 , 4
    syscall
    la  $a0 , $str3
    li  $v0 , 4
    syscall

#####################
# Fin
       jr $ra

aduroom_clu-7&-35:miniC morad$
```

Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
Final

- **IMPLEMENTACION DE LA SENTENCIA BUCLE-FOR**

Para ejecutar una sentencia FOR, se ha decidido hacerlo por medio de esta sintaxis:

```
for (id = a; boolean ; a := a + 1) {
    statement_list;
}
```

Consiste en un contador previamente definido que es inicializado. Mientras que no se cumpla la condición, se ejecuta la lista de sentencias, y se incrementa el contador hasta que se cumpla la condición satisfactoriamente. Para construir el cuerpo de un bucle FOR, vamos a tener en cuenta los siguientes campos:

```
for( id = a ; id < b; id := id + c) {
      statement_list
}
```

$5 : Va a ser el valor de inicio.
$7: Va a ser la condición. Es 1 la condición se cumple, y 0 en caso contrario.
$9: El valor de incremento.
$11: El valor de incremento.
$14: La lista de sentencias.

Primeramente vamos a introducir a nuestra lista de código, el valor de $5

```
enlazarlistaCodigos($$, $5);
```

Luego creamos una cuádrupla con la operación sw, para almacenar nuestro valor inicial de $5 en un identificador.

```
Cuadrupla sw1 = nueva_cuadrupla("sw", $5->ultima->destino,
barra($3), NULL);

insertarCuadrupla(sw1, $$);
```

Insertamos una etiqueta que indique que se entra en el bucle.
Enlazamos el código de nuestra condición booleana a nuestra lista de códigos y creamos nuevas cuádruplas para que comprueben el resultado de la condición, y así determinar si se hace un salto a la etiqueta de salida o se continúa con la ejecución del bucle.

```
enlazarlistaCodigos($$, $7);
insertarCuadrula(Label1, $$);
Cuadrupla bnez = nueva_cuadrupla(“bnez”, NULL, $7->primera
->destino, Label2->op);
insertarCuadrupla(bnez, $$);
```

Enlazamos el código del incremento y de la lista de sentencias.

```
enlazarlistaCodigos($$, $11);
enlazarlistaCodigos($$, $14);
```

Insertamos una operación de salto hacia la primera etiqueta.

```
Cuadrupla Label1 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL);

Cuadrupla Label2 = nueva_cuadrupla(crear_etiqueta(), NULL, NULL, NULL):

Cuadrupla sw1 = nueva_cuadrupla("sw", $5->ultima->destino, barra($3),
NULL);
Cuadrupla sb = nueva_cuadrupla("sb",$11->ultima->destino, barra($9),
NULL);
Cuadrupla beq = nueva_cuadrupla("bnez",NULL,  $7->primera          -
>destino, Label2->op);
Cuadrupla sw2 = nueva_cuadrupla("sw", $11->ultima->destino, barra($9),
NULL);
Cuadrupla b = nueva_cuadrupla("b",NULL , NULL, Label1->op);

$$=nueva_listaCodigo();
enlazarlistaCodigos($$,$5);
insertarCuadrupla(sw1, $$);
enlazarlistaCodigos($$,$7);
insertarCuadrupla(Label1, $$);
insertarCuadrupla(sb, $$);
insertarCuadrupla(beq, $$);
enlazarlistaCodigos($$,$14);
enlazarlistaCodigos($$,$11);
insertarCuadrupla(sw2, $$);
insertarCuadrupla(b, $$);
insertarCuadrupla(Label2, $$);
eliminarRegistro($5);
eliminarRegistro($7);
eliminarRegistro($11);
```

PruebaFOR.c

```
func prueba() {

let a = 0, b = 0;
var c = 6 -2 + 2;
var d = 0;
for(d = 0; d < 4; d := d + 1) {
     c := c + 1;
     print “c =”, c, “\n”;
}
}
```

salidaFOR.s

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

$str2:
        .asciiz "\n"
$str1:
        .asciiz "c = "
_a:
        .word 0
_c:
        .word 0
_b:
        .word 0
_d:
        .word 0

#########################
# Seccion de codigo
     .text

     .globl main
main:
     li  $t0, 0
     sw  $t0, _a
     li  $t0, 0
     sw  $t0, _b
     li  $t0, 6
     li  $t1, 2
     add  $t2, $t0, $t1
     li  $t0, 2
     sub  $t1, $t2, $t0
     sw  $t1, _c
     li  $t0, 0
     sw  $t0, _d
     li  $t0, 0
     sw $t0, _d
     lw  $t1, _d
     li  $t2, 4
     slt $t3, $t1, $t2
 Label1:
     nb $t4, _d
     beqz $t1, Label2
     lw  $t1, _c
     li  $t2, 1
     add  $t5, $t1, $t2
     sw  $t5, _c
     la  $a0 , $str1
     li  $v0 , 4
     syscall
     lw  $t1, _c
     move  $a0 , $t1
     li  $v0 , 1
     syscall
     la  $a0 , $str2
     li  $v0 , 4
     syscall
     lw  $t1, _d
     li  $t2, 1
     add  $t4, $t1, $t2
     sw $t4, _d
     b, Label1
 Label2:

#########################
# fin
```

SALIDA:

```
c = 7
c = 8
```