

MEMORIA PRÁCTICAS COMPILADORES

CONVOCATORIA JUNIO 23/24

Alfredo Marquina Meseguer
49445345Z
alfredo.m.m@um.es
Grupo 2.2
Universidad De Murcia

ÍNDICE

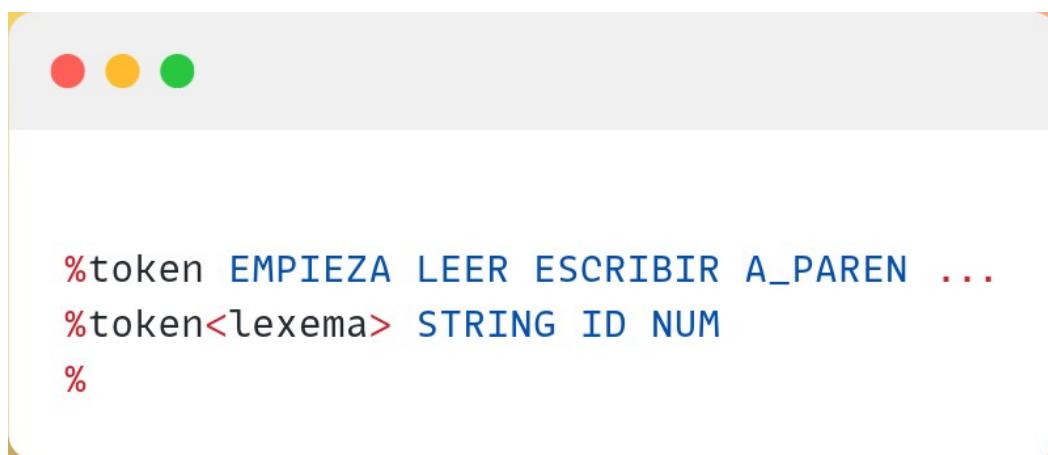
1 Análisis Léxico.....	4
1.1 Macros.....	5
1.2 Palabras Reservadas y Símbolos.....	6
1.3 Comentarios.....	6
1.4 IDs y Tipos.....	7
1.5 Errores.....	9
2 Análisis Sintáctico.....	10
2.1 Conflictos Desplazamiento/Reducción.....	10
2.2 Precedencia Operadores.....	11
3 Análisis Semántico.....	11
3.1 Resolución de Nombres.....	12
3.2 Funciones Añadidas a la Lista de Símbolos.....	13
4 Generación de Código.....	15
4.2 Manual del Usuario: Compilar un Programa MiniC.....	19
4.3 Pruebas.....	20
4.3.1 Prueba Original.....	20
4.3.2 Prueba Junio 2023.....	22
4.3.3 Prueba Bucle Do-While.....	26
4.3.4 Prueba Operadores Relacionales.....	30
4.3.5 Prueba Bucle For.....	35
5 Extensiones.....	39
5.1 Bucle Do-While.....	39
5.1.1 Edición Léxico.....	40
5.1.2 Edición Sintáctico.....	40
5.1.3 Generación de Código.....	41
5.2 Operadores Relacionales.....	42
5.2.1 Edición Léxico.....	42
5.2.2 Edición Sintáctico.....	42
5.2.3 Generación de Código.....	43
5.3 Bucle For.....	49
5.3.1 Edición Léxico.....	50
5.3.2 Edición Sintáctico.....	51
5.3.3 Edición Semántico.....	52

5.3.4 Generación de Código.....52

1 ANÁLISIS LÉXICO

El análisis léxico se trata de una subrutina del análisis sintáctico que se encarga de tokenizar la entrada del programa. El análisis sintáctico necesita que esta entrada se encuentre expresada en tokens, así que llama al léxico cada vez que quiere obtener de la entrada una sentencia comprensible.

En nuestro código, especificaremos los tokens que tenemos al inicio del fichero `miniC.y` antes de la definición del sintáctico. Estos tienen dos tipos, unos normales definidos con la etiqueta `%token` al principio de la línea y se parados entre ellos por los espacios; y el segundo son tipos de tokens cuyo valor importa, esto se ve reflejado por la etiqueta `%token<lexema>` al inicio, siendo `lexema` una variable tipo cadena de caracteres que guardará el valor de los tokens una vez reconocidos.



```
%token EMPIEZA LEER ESCRIBIR A_PAREN ...
%token<lexema> STRING ID NUM
%
```

A partir de estas dos líneas se creará una enumeración de tokens en el fichero `miniC.tab.h`, además, se crearán los tokens especiales `YYEMPTY`, `YYEOF`, `Yyerror` e `YYUNDEF` para usados durante la compilación. Bison es el programa que se encarga, entre otras cosas, de generar los tokens y del semántico y sintáctico. Sin embargo, para realizar el análisis léxico, vamos a necesitar una herramienta llamada Flex, basada en el reconocimiento de cadenas con expresiones regulares, aunque más potente por el uso de banderines y demás mecanismos. El reconocimiento del léxico se realiza en el fichero `miniC.l`.

```
/* Token kinds. */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
enum yytokentype
{
    YYEMPTY = -2,
    YYEOF = 0, /* "end of file" */
    YYerror = 256, /* error */
    YYUNDEF = 257, /* "invalid token" */
    EMPIEZA = 258, /* EMPIEZA */
    LEER = 259, /* LEER */
    ESCRIBIR = 260, /* ESCRIBIR */
    A_PAREN = 261, /* A_PAREN */
    C_PAREN = 262, /* C_PAREN */
    ...
}
```

1.1 MACROS

Una macro o macroinstrucción, es una serie de instrucciones que se ejecutan con una sola llamada, es este caso se trata de una expresión regular que se encuentra repetidas veces en el código, o es muy importante y se separa para saber cuál es su utilidad.

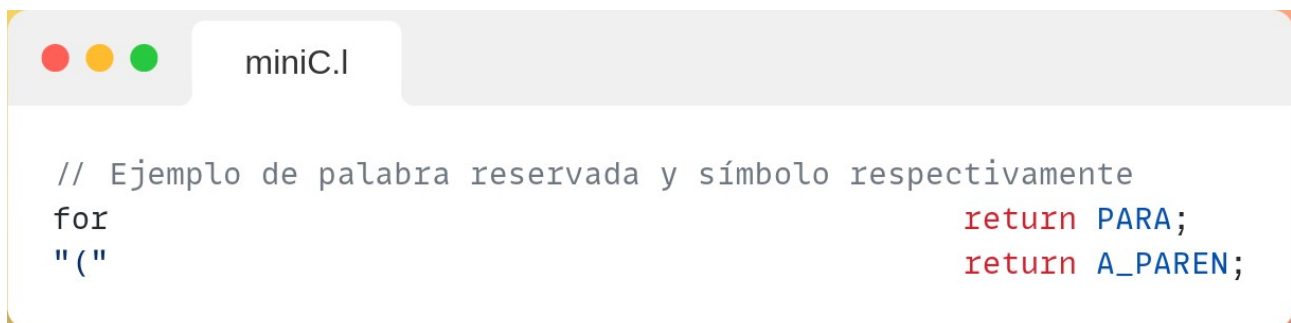
En nuestro código hemos utilizado macros para reconocer las partes de los lexemas, mencionados en el apartado anterior, con dígito y letra. Mientras que reconocemos los números permitidos con entero. La expresión más importante aquí es `panico` que se utiliza para reconocer todos los caracteres no permitidos en la gramática. La manera más simple de hacer esto es reconociendo el conjunto de todos los caracteres que pertenecen a la gramática y negándolos.

dígito	<code>[0-9]</code>
letra	<code>[a-zA-Z]</code>
entero	<code>{dígito}+</code>
panico	<code>[^A-Za-z0-9();,=+*\-{}_<>!"']</code>

1.2 PALABRAS RESERVADAS Y SÍMBOLOS

Una palabra reservada es un término que tienen un comportamiento propio, razón por la cual ninguna otra variable o función debería compartir nombre con ella. Especificar las palabras reservadas es una parte sencilla del léxico ya que como siempre tienen la misma forma no es necesario reconocer ningún patrón mas que el mismo nombre. Flex analiza todas las posibles combinaciones al mismo tiempo, pero da prioridad de aceptación según el orden de aparición en el código. Por lo tanto, para evitar esta coincidencia, vamos a especificarlas lo antes posible en el código.

De la misma manera, se obtienen los símbolos. Sin embargo, como muchos tienen funciones dentro de comillas, para que Flex “sepa” que lo que queremos utilizar es el carácter en sí y no utilizarlo como modificador de la expresión regular.



```
// Ejemplo de palabra reservada y símbolo respectivamente
for                                     return PARA;
"("                                   return A_PAREN;
```

1.3 COMENTARIOS

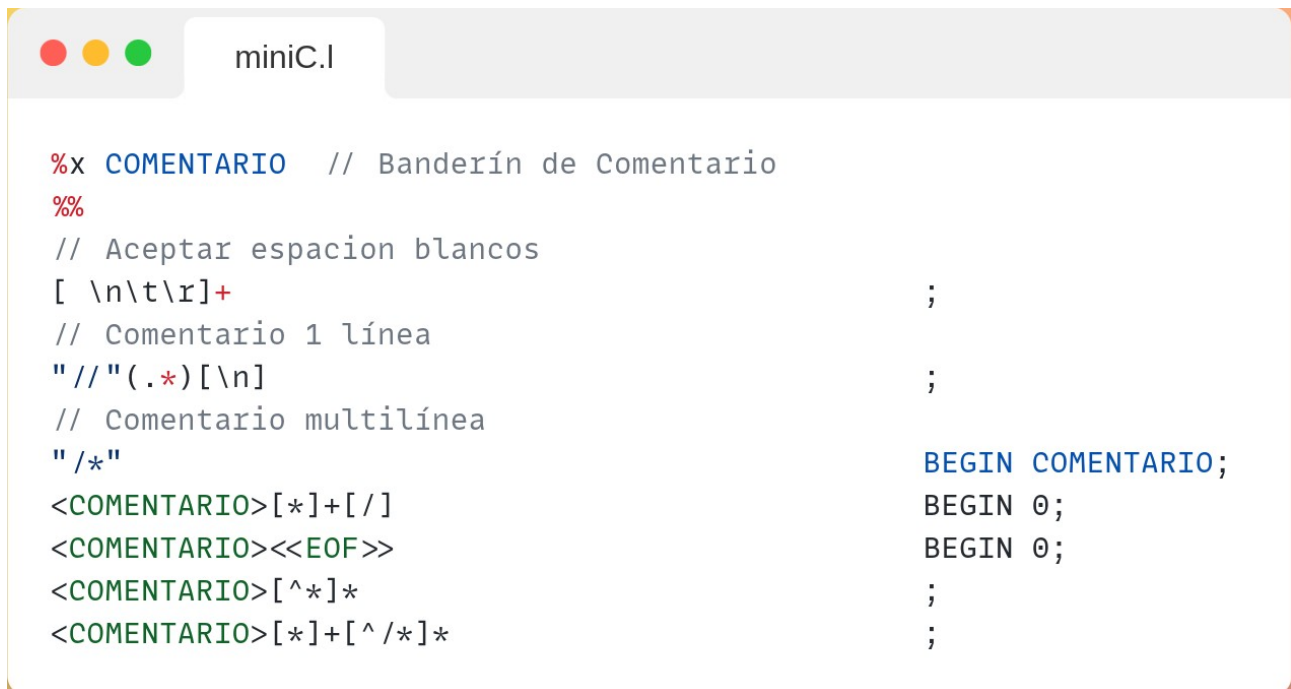
Para realizar comentario, partes del código que no se compilan, se han creado dos formas de hacerlo, siguiendo el estilo dado por C: el de una sola línea y el multilínea.

El primero es sencillo, pues se puede decir que un comentario de estos empieza por dos barras (/) seguidas y terminan con un salto de línea.

El segundo tipo, el comentario multiínea, resulta más complicado porque se trata de reconocer una cadena tal que los dos primeros caracteres son “/*” y los dos últimos “*/” o se acabe el fichero y entre estos dos puede contener cualquier otro, inclusive los que componen la combinación de salida, aunque sin tener el mismo orden. Para conseguir reconocer estos comentarios vamos a hacer uso de los banderines.

En Flex, existe una variable que contiene como el “modo” en el que se encuentra en cierto momento. Cada estado de estos es conocido como un banderín. El estado inicial se llama INITIAL o 0, pero puede ser cambiado con la sentencia: BEGIN banderín. Para cada sentencia se puede especificar un banderín con el que se puede leer de la forma: <banderín>expresión. Existen dos tipos de banderines: los inclusivos, que permiten reconocer expresiones que no especifican banderín y la que especifican el suyo propio; y los exclusivos, que solo permiten leer aquellos mensajes tengan el banderín especificado.

Utilizando un banderín exclusivo para los comentarios multilínea podemos asegurarnos de que no aceptemos accidentalmente un comentario como un símbolo, palabra reservada o identificador. Iniciamos el banderín cuando se reconoce “/*” de cadena de inicio y vamos reconociendo cualquier carácter hasta que encontremos una combinación de “*/” o se termine el fichero. A todo esto se le han añadido ciertas mejoras para que se tengan que hacer menos pasadas y se reconozca antes el comentario.



```
%X COMENTARIO // Banderín de Comentario
%%
// Aceptar espacio blancos
[ \n\t\r]+ ;
// Comentario 1 línea
"//"(.*)[\n] ;
// Comentario multilínea
"/*" BEGIN COMENTARIO;
<COMENTARIO>[*]+[/] BEGIN 0;
<COMENTARIO><<EOF>> BEGIN 0;
<COMENTARIO>[^*]* ;
<COMENTARIO>[*]+[^/*]* ;
```

1.4 IDs Y TIPOS

Una variables es una cadena que hace referencia a una parte de memoria reservada donde se puede guardar información. En el análisis léxico no se pueden declarar variables, sin embargo, sí que se tienen que reconocer sus nombres o identificadores.

Cuando hablamos de tipos nos referimos a los valores que pueden tomar dichas variables o se utilizar para realizar cálculos. En MiniC, solo existen dos tipos de valores: los enteros y las cadenas de texto. Todos los valores expresados en pertenecen a uno de esos dos tipos, hasta los operadores relacionales devuelven enteros no booleanos.

Tanto tipos como identificadores tienen en común que no solo importa el valor de su token, sino también el contenido que formó este, el cual se guarda en una variable lexema que contiene dicha cadena de texto. La variable yyval que contiene lexema, nos viene dada por Bison, nos permite pasar este valor al sintáctico. Mientras tanto, las variable yytext es de Flex y contiene la cadena de texto leída. Utilizamos strdup para pasar la copia y asegurarnos que nos es borradas por alguna casualidad.

Además, para los identificadores tenemos que controlar que su longitud no se pase de los treinta y dos caracteres, sin embargo, si lo hiciésemos con llaves de repetición solo

conseguiríamos que para un edificador de 32 letras, se “contara” como dos identificadores, uno de 31 letras y otro de una. Esto provocaría dos errores incorrecto, uno por no seguir el orden correcto de tokens y el otro por ser dos identificadores no declarados, cuando el verdadero error es el id muy largo. Por esta razón, se comprueba con una sentencia if. La variable `yyleng` también pertenece a Flex y contiene el tamaño de la cadena de caracteres `yytext` mencionada anteriormente. Si no cumple la condición y se muestra como un error léxico.

De la misma manera, tenemos que comprobar el valor de una cadena que representa a un entero, pues el valor de este no puede superar `INT_MAX` o se desbordaría el número llevando a comportamiento indefinido. Por eso, convertimos a `long` y comparamos con `INT_MAX`. Si no cumple la condición y se mostrará como un error léxico.

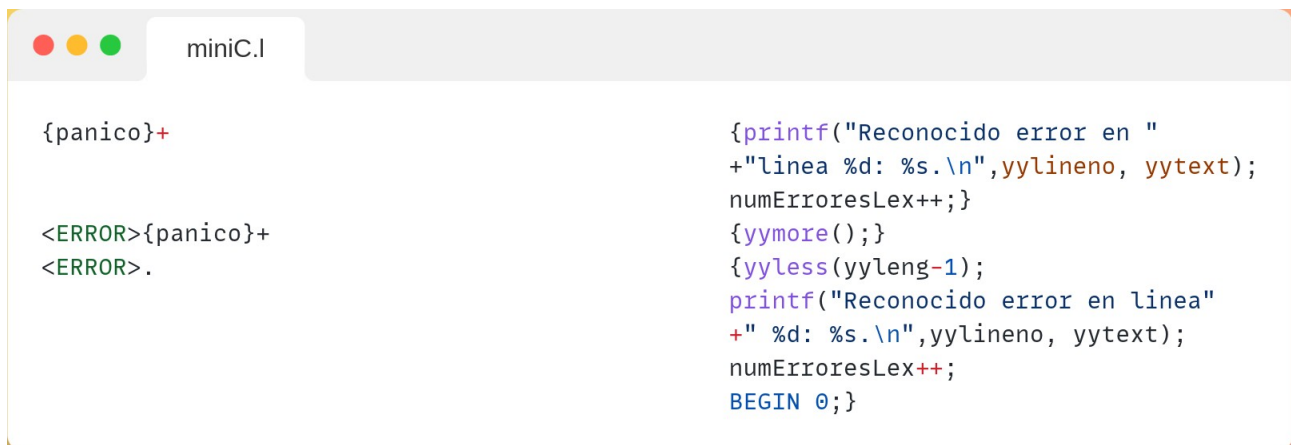


```
(_{letra})({letra}|{digito}|_)*  
  
{entero}  
  
\"([^\n]|\\.)*(\n|\\n)  
  
{if (yyleng < 32){  
    yylval.lexema = strdup(yytext);  
} else {  
    yymore();  
    BEGIN ERROR;  
}return ID;}  
{if (atoll(yytext) < INT_MAX){  
    yylval.lexema = strdup(yytext);  
}else{  
    yymore();  
    BEGIN ERROR;  
}return NUM;}  
{yylval.lexema = strdup(yytext);  
return STRING;}
```


1.5 ERRORES

Cuando ocurre un error léxico porque no se ha reconocido un carácter no válido, este entra sin banderín a la primera entrada, la cual reconoce todos los caracteres que no pertenecen a la gramática.

Por otra parte, cuando uno de los dos casos del apartado anterior utiliza habilita el banderín ERROR. El programa actúa de forma similar a como lo hace sin este. La razón por la que se añade este banderín es para poder realizar las llamadas desde la parte de reconocimiento de números e identificadores.



```
{panico}+

<ERROR>{panico}+
<ERROR>.

{printf("Reconocido error en "
+"línea %d: %s.\n",yylineno, yytext);
numErroresLex++;}
{yymore();}
{yyless(yyleng-1);
printf("Reconocido error en línea"
+" %d: %s.\n",yylineno, yytext);
numErroresLex++;
BEGIN 0;}
```

2 ANÁLISIS SINTÁCTICO

El sintáctico es la parte del compilador que se encarga de validar y comprender la estructura gramatical del programa según las reglas de nuestro lenguaje de programación (MiniC) y construir un árbol sintáctico que represente la estructura y las relaciones que existen en el programa. El análisis sintáctico se realiza con Bison, por lo que se realiza en el fichero del mismo con extensión *.y*, en nuestro caso *miniC.y*.

Como casi todo el sintáctico viene dado por el enunciado del trabajo no se va a explicar mucho. Por separado sí que se especificará como se han añadido las extensiones al sintáctico.

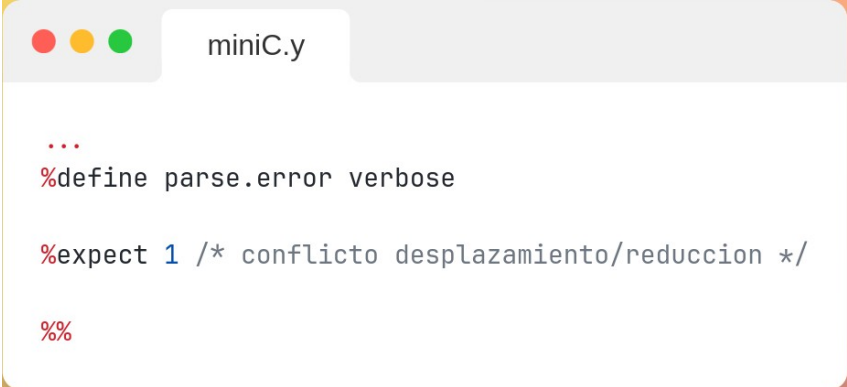
2.1 CONFLICTOS DESPLAZAMIENTO/REDUCCIÓN

Si bien nuestra gramática está plagada de conflictos es no es un problema que nos impida crear nuestro lenguaje de programación.

Durante la ejecución del compilador, si Bison se encuentra con un conflicto desplazamiento-reducción tiende a favorecer a desplazar ya que este enfoque suele ser más seguro en términos de la estructura sintáctica del análisis.

Además, también podemos encontrarnos con conflictos reducción-reducción, es decir que hay dos posibles reglas de tipo reducción a aplicar. En esta caso Bison simplemente la primera regla presente según el orden en el que se han especificado en el fichero *.y*. Sin embargo, esto no siempre resulta en el comportamiento deseado, por lo que existe la posibilidad de especificar de forma explícita el comportamiento de estos como se verá en el siguiente apartado.

Finalmente, si no se puede resolver un conflicto de automáticamente o si la resolución elegido puede no seguir nuestras intenciones, Bison genera mensajes de advertencia o errores para informar sobre la situación. Es por esto que en nuestro código tenemos que añadir una clausula *%expect 1*, porque o sino se provocaría una de estos y no nos estaría permitido compilar. Al establecer el *expect* con valor a 1, estamos indicando que esperando que haya un conflicto de reducción en la gramática.



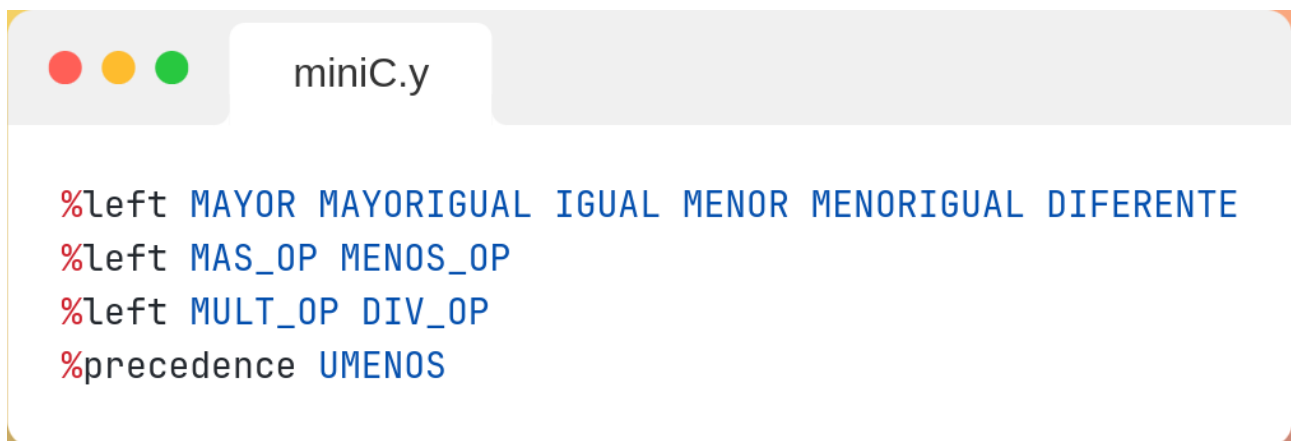
```
...  
%define parse.error verbose  
  
%expect 1 /* conflicto desplazamiento/reduccion */  
  
%%
```

2.2 PRECEDENCIA OPERADORES

Para evitar la ambigüedad causadas por los conflictos reducción-reducción cuando tratamos con los operadores matemáticos y relacionales tenemos que establecer el orden de precedencia de estos, de no establecerlo provocaríamos que el único orden fuera de izquierda a derecha.

El orden a establecer será por grupos, los grupos tendrán precedencia uno sobre otro, pero dentro de los grupo no habrá, solo el orden de aparición. Además, vamos a tener que crear un nuevo token UMENOS, que comparta lexema con MENOS_OP, es decir, ambos tienen de lexema “-”, pero se asignará uno o el otro según el contexto de la aparición de este símbolo, el primero para números negativos y el segundo para operaciones de resta.

Los grupos creados son el de UMENOS, multiplicación y división, suma y resta y operadores relacionales, siendo UMENOS el de mayor precedencia y los operadores relacionales el de menor.

A screenshot of a code editor window titled 'miniC.y'. The window has a standard macOS-style title bar with red, yellow, and green buttons. The code inside is as follows:

```
%left MAYOR MAYORIGUAL IGUAL MENOR MENORIGUAL DIFERENTE
%left MAS_OP MENOS_OP
%left MULT_OP DIV_OP
%precedence UMENOS
```

3 ANÁLISIS SEMÁNTICO

El análisis semántico se encarga de atribuir significado a las estructuras generadas en la etapa anterior. Esto implica verificar que las operaciones y demás construcciones sean coherentes y válidas según las reglas. En esta etapa se suelen realizar verificaciones de tipo, resolución de nombres, conversiones implícitas, análisis de control de flujo, etc.

Sin embargo, como nuestra gramática resulta muy simple y solo tenemos un tipo posible para las variables, solo realizaremos la resolución de nombre y guardaremos cadenas en la sección de datos.

Se añadirán los identificadores según se declaren y se guardará si son variables o constantes, se comprueba que los identificadores usados hallan sido declarados y que aquellos cuyo valor sea editado no sean constantes.

También veremos las funciones que han sido implementadas en los ficheros listaSimbolos para que la lista de símbolos sea funcional.

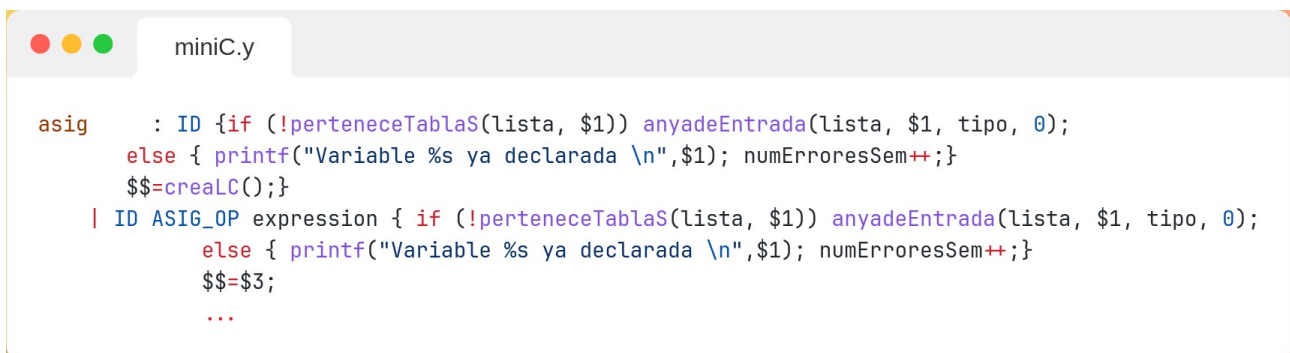
3.1 RESOLUCIÓN DE NOMBRES

Los errores semánticos posibles ya se han sido definidos por el código proporcionado de miniC.y estos se tratan de los casos:

1. Un identificador ya se ha declarado, por lo que no se puede volver a declarar.
2. Un identificador no ha sido declarado, por lo que no se puede llamar.
3. Un identificador se ha declarado como constante y se está intentando modificar.

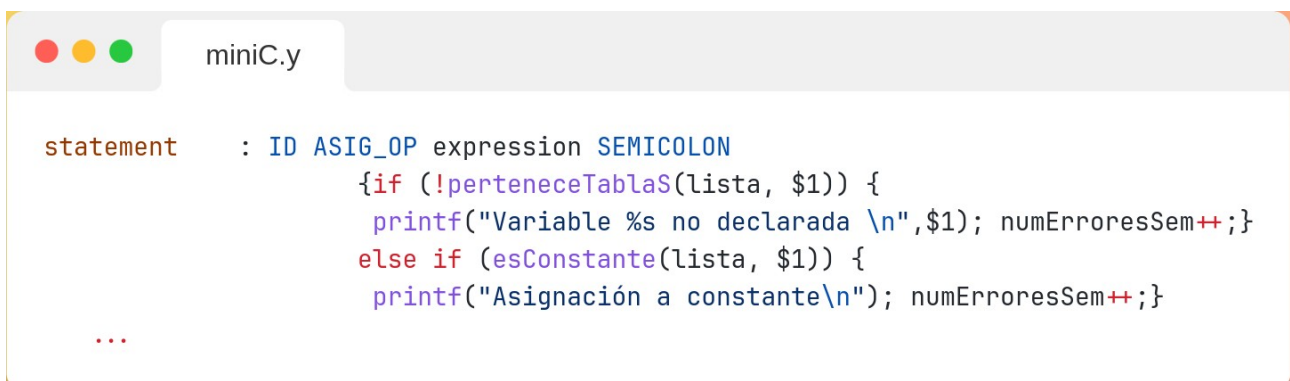
El primer punto solo es posible durante la sección de datos. Para alcanzar este comportamiento tenemos que tener en cuenta la tabla de símbolos. Esta estructura de datos se trata de una lista, llamada lista, de un tipo de *struct* llamado Nodo. Cada uno de los nodos de esta lista tiene definido un nombre, tipo (es un *enum* de los diferentes tipos) y un valor que es un entero.

En código que vemos abajo podemos observar como al declarar un nuevo identificador, se comprueba que este no haya sido declarado previamente, añadiendo un error semántico si lo ha sido o añadiéndolo a la lista de símbolos si no lo ha hecho.



```
asig      : ID {if (!perteneceTablaS(lista, $1)) anyadeEntrada(lista, $1, tipo, 0);
             else { printf("Variable %s ya declarada \n", $1); numErroresSem++;}
             $$=crealC();}
| ID ASIG_OP expression { if (!perteneceTablaS(lista, $1)) anyadeEntrada(lista, $1, tipo, 0);
                           else { printf("Variable %s ya declarada \n", $1); numErroresSem++;}
                           $$=$3;
                           ...
```

Finalmente, el punto dos se comprueba cada vez que se utiliza un identificador y el punto tres cada vez que se asigna. Por esta razón, cuando se asigne un nuevo valor a un identificador, se comprobarán ambos puntos. Como podemos ver en el código, primero se comprueba que esté declarada y de estarlo, se comprueba si es constante. Es importante que las comprobaciones se realicen en este orden o podría haber un error.



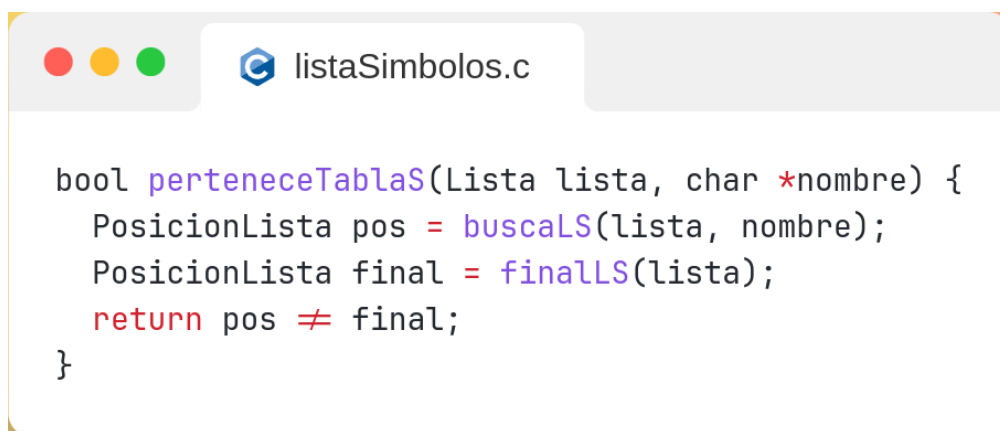
```
statement : ID ASIG_OP expression SEMICOLON
           {if (!perteneceTablaS(lista, $1)) {
               printf("Variable %s no declarada \n", $1); numErroresSem++;}
           else if (esConstante(lista, $1)) {
               printf("Asignación a constante\n"); numErroresSem++;}
           ...
```

3.2 FUNCIONES AÑADIDAS A LA LISTA DE SÍMBOLOS

Se han implementado una serie de funciones para completar el código proporcionado listaSimbolos. Estas funciones son: perteneceTablaS, anyadeEntrada, esConstante e imprimeTabla.

PERTENECETABLAS

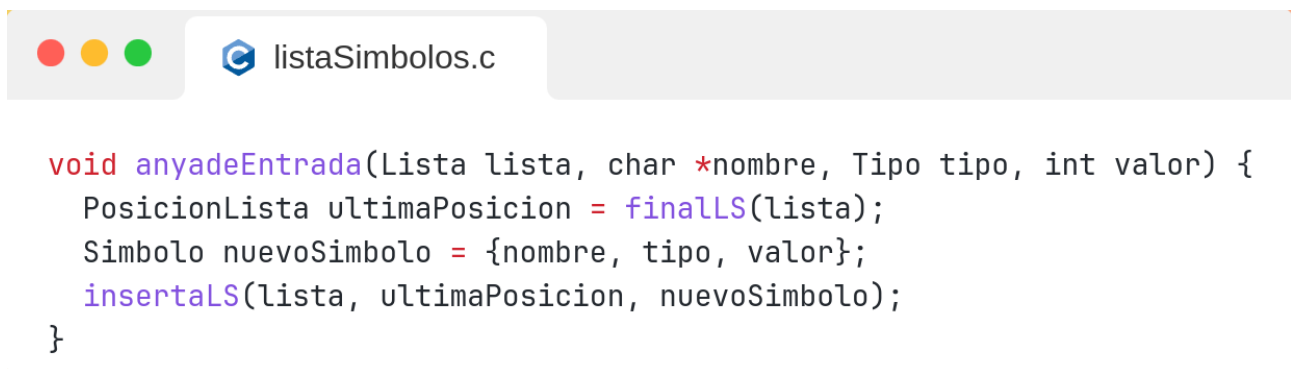
Esta primera función trata de comprobar la pertenencia de un elemento a la tabla de símbolos. Para ello, basta con identificar su posición en la lista y comprobar que no se encuentra en la última, pues es el valor devuelto en caso de no estar presente en la estructura de datos.



```
bool perteneceTablaS(Lista lista, char *nombre) {
    PosicionLista pos = buscaLS(lista, nombre);
    PosicionLista final = finalLS(lista);
    return pos != final;
}
```

anyadeEntrada

En la segunda, se añade una entrada a la lista de símbolos, para ello, debemos insertar un struct Simbolo al final de la lista.



```
void anyadeEntrada(Lista lista, char *nombre, Tipo tipo, int valor) {
    PosicionLista ultimaPosicion = finalLS(lista);
    Simbolo nuevoSimbolo = {nombre, tipo, valor};
    insertaLS(lista, ultimaPosicion, nuevoSimbolo);
}
```

ESCONSTANTE

En la tercera comprobamos que un elemento sea constante, esta función se utiliza para que cada vez que se asigna o edita una variable podamos saber si se trata de una constante. La forma de comprobarlo es buscando en la lista la posición de este elemento. De encontrarse, se debería poder comprobar el valor del tipo el cual puede o no ser CONSTANTE.

```
bool esConstante(Lista lista, char *nombre) {
    Simbolo s;
    PosicionLista pos = buscaLS(lista, nombre);
    if (pos != finalLS(lista)) {
        s = recuperaLS(lista, pos);
        return (s.tipo == CONSTANTE);
    }
    return false;
}
```

IMPRIME TABLA

Finalmente la función que imprime la declaración de los datos a partir de la información guardada en una lista de símbolos. Se encuentra compuesta por la impresión de la cabecera y un par de bucles, donde el primero declara las cadenas y en el segundo, el resto de tipos que resultan ser solo números.

```
void imprimirTablaS(Lista lista) {
    printf("#####\n");
    printf("# Seccion de datos\n");
    printf("\t.data\n");
    printf("\n");
    PosicionLista pos = inicioLS(lista); // Obtenemos la posición del primer elemento
    Simbolo simbolo;

    while (pos != finalLS(lista)) { // Recorremos la lista hasta llegar al final
        simbolo = recuperaLS(lista, pos); // Obtenemos el simbolo de la posición actual

        // Imprimimos la información del simbolo con formato según su tipo
        if (simbolo.tipo == CADENA) {
            printf("$str%d:\n", simbolo.valor);
        }
    }
}
```

```
        printf("\t.asciiz %s\n", simbolo.nombre);
    }

    pos = siguienteLS(lista, pos); // Avanzamos a la siguiente posición
}

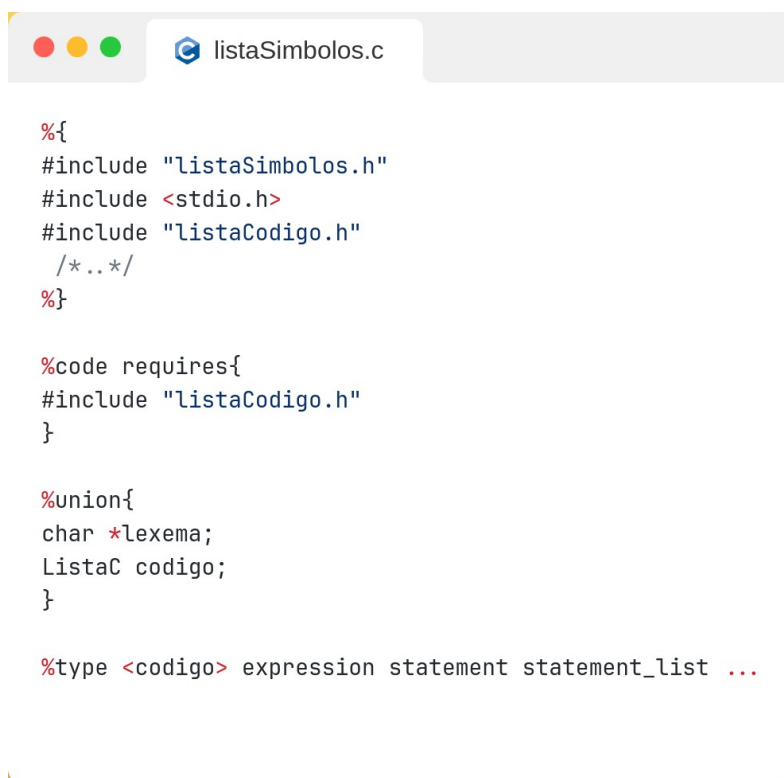
pos = inicioLS(lista);
while (pos != finalLS(lista)) {
    simbolo = recuperaLS(lista, pos);
    if (simbolo.tipo != CADENA) {
        printf("_%s:\n", simbolo.nombre);
        printf("\t.word 0\n");
    }
    pos = siguienteLS(lista, pos);
}
printf("\n");
}
```

4 GENERACIÓN DE CÓDIGO

Finalmente, generaremos el programa sin la sección de datos, pues fue generada en el semántico. Esta etapa, así como la sección de datos no serán impresas si el programa encuentra algún error en alguno de los tres análisis anteriores.

Las instrucciones se generarán durante la ejecución del compilador y se irán guardando en el orden adecuado en una lista específica para el código. Esta lista está formada por nodos que simulan la estructura normal de una instrucción de ensamblador y se convertirán en una durante la impresión del código. Cada nodo cuenta con cuatro campos y todos son cadenas de texto son: oper, res, arg1 y arg2. Siendo el primero donde se encuentra la instrucción o la etiqueta. Res, arg1 y arg2 están pensados para escribir nombres de registros, direcciones de memoria o inmediatos que se quieran utilizar. Los nombres de estos atributos están basados en la estructura de una instrucción matemática como *add* o *div* siendo la estructura: oper res, arg1, arg2.

Para empezar tenemos que importar la lista en el fichero de Bison, el miniC.y. La generación se realiza dentro de este fichero una vez identificada una regla sintáctica



```
%{
#include "listaSimbolos.h"
#include <stdio.h>
#include "listaCodigo.h"
/*..*/
}%

%code requires{
#include "listaCodigo.h"
}

%union{
char *lexema;
ListaC codigo;
}

%type <codigo> expression statement statement_list ...
```

A lo largo de la generación de código, vamos a tener que obtener registros para realizar las operaciones y cuando terminemos con estos liberarlos. Para ello tenemos una estructura de datos en listaCodigo que nos sirve para realizar un seguimiento de qué registros estamos utilizando y cuáles no. Esta estructura se llama registros y es un array de booleanos.

INIREG

Inicializa todos los registros a 0 mediante el array de booleanos registros[]. Previamente reservamos memoria.

```
void iniReg() {
    registros = malloc(sizeof(int)*NUM_REGISTROS);
    for (int i = 0; i < NUM_REGISTROS; i++) registros[i] = 0;
}
```

OBTENERREG

recorre la lista de registros y para el primer registro libre lo pone a ocupado. Posteriormente devuelve el nombre (junto con el índice) del primero libre.

```
char *obtenerReg() {
    char registro[4];
    for (int i = 0; i < NUM_REGISTROS; i++) {
        if (registros[i] == 0) {
            registros[i] = 1;
            sprintf(registro, "$t%d", i);
            return strdup(registro);
        }
    }
}
```

NUEVAETIQUETA

Es una función que crea nuevas etiquetas no utilizadas y las devuelve.

Aumentando el contador global numEtiqu nos aseguramos la unicidad de cada etiqueta.



```
char *nuevaEtiqueta() {
    char aux[16];
    sprintf(aux, "$l%d", numEtiqu);
    numEtiqu++;
    return strdup(aux);
}
```

IMPRIMIRLISTAC

Es la función final que imprime el código generado de forma ordenada y correcta. Se compone de una parte inicial predefinida donde se inicia el programa main, un bucle donde se recorre LC y una parte final también predefinida que nos saca del problema, así como la liberación de los registros. A pesar de ser una de las funciones más largas se trata también de las más simples. La mayor complejidad se encuentra en el bucle que recorre LC. En este bucle se imprimen las instrucciones y las etiquetas de forma diferente, por lo que se debe comprobar el campo op de operación si tiene "etiq" o no.



```
void imprimirListaC(ListaC listaCodigo) {
    printf("#####\n");
    printf("# Seccion de codigo\n");
    printf("\t.text\n");
    printf("\t.globl main\n");
    printf("main:\n");

    Operacion oper;
    PosicionListaC pos = inicioLC(listaCodigo);

    while (pos != finalLC(listaCodigo)) {
        oper = recuperaLC(listaCodigo, pos);
        if (strcmp(oper.op, "etiq")) {
            printf("\t%s", oper.op);
            if (oper.res) printf(" %s", oper.res);
            if (oper.arg1) printf(", %s", oper.arg1);
            if (oper.arg2) printf(", %s", oper.arg2);
        }
    }
}
```

```
        else printf("%s:", oper.res);  
        printf("\n");  
        pos = siguienteLC(listaCodigo, pos);  
    }  
  
    printf("\n#####\n");  
    printf("# Fin\n");  
    printf("\tli $v0, 10\n");  
    printf("\tsyscall\n");  
    printf("\n");  
  
    free(registros);  
}
```

Esta función (imprimirListaC) se ejecuta tras el análisis semántico del programa si no se encuentra ningún error.

4.2 MANUAL DEL USUARIO: COMPILAR UN PROGRAMA MINIC

Para la ejecución del programa seguiremos los siguientes pasos:

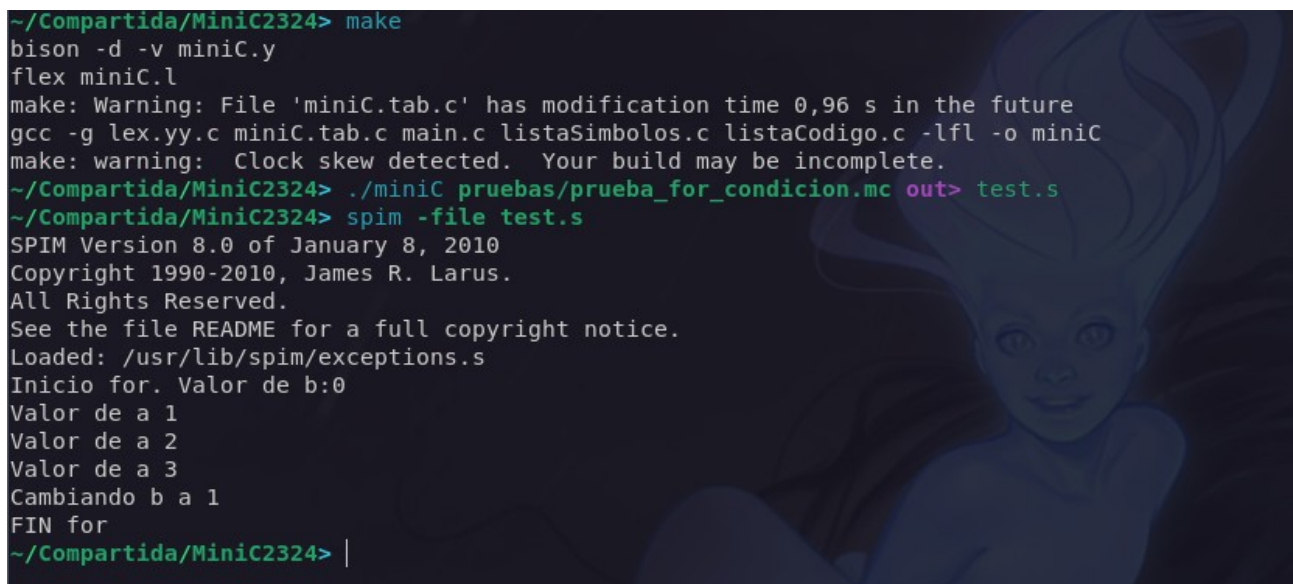
1. Nos colocaremos en el mismo directorio en el que se encuentran los ficheros del programa y ejecutaremos el comando *make* para compilar.
2. Con esto conseguiremos un ejecutable llamado *miniC*.
3. Para poder compilar cualquier programa de *miniC* ejecutaremos este archivo junto al nombre del fichero que se desea compilar. La salida del programa es por consola, así que para guardarla en un fichero se debe redireccionar. La estructura resultante es la siguiente:

```
./miniC [fichero miniC] > [fichero salida]
```

4. Una vez obtenido el programa en ensamblador MIPS, podemos ejecutarlo utilizando IDE de desarrollo como Mars o programas de consola como *spim*. Por simplicidad se recomienda el uso de *spim*, la forma de ejecutarlo es la siguiente:

```
spim -file [fichero]
```

Se recomienda que los ficheros de *miniC* tengan la extensión *.mc*, aunque solo sean texto plano, a su vez, se recomienda que la compilación del programa se guarde en un fichero con extensión *.s*.



```
~/Compartida/MiniC2324> make
bison -d -v miniC.y
flex miniC.l
make: Warning: File 'miniC.tab.c' has modification time 0,96 s in the future
gcc -g lex.yy.c miniC.tab.c main.c listaSimbolos.c listaCodigo.c -lfl -o miniC
make: warning: Clock skew detected. Your build may be incomplete.
~/Compartida/MiniC2324> ./miniC pruebas/prueba_for_condicion.mc out> test.s
~/Compartida/MiniC2324> spim -file test.s
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
Inicio for. Valor de b:0
Valor de a 1
Valor de a 2
Valor de a 3
Cambiando b a 1
FIN for
~/Compartida/MiniC2324> |
```

Aquí tenemos un ejemplo de compilación para la prueba *prueba_for_condición.mc*

4.3 PRUEBAS

4.3.1 PRUEBA ORIGINAL

CÓDIGO MINIC

```
main() {
    var x = 30, y = -10;
    var b = 3;
    var z;
    z = x + y;
    print ("hola \"esto es\" una prueba", "\n");
    /* Esto es
       un comentario
       multilinea
    */

    // Esto es un comentario de una sola linea

    if (b) print ("b es true porque su valor es distinto de 0\n");
    while(b) {
        b = b-1;
        print("El valor de b va disminuyendo en", "cada iteración\n");
    }
    if(b) print ("hola");
    else print ("El valor de b es 0\n");
}
```

CÓDIGO GENERADO MIPS

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

$str1:
.asciiz "hola \"esto es\" una prueba"
$str2:
.asciiz "\n"
$str3:
.asciiz "b es true porque su valor es distinto de 0\n"
$str4:
.asciiz "El valor de b va disminuyendo en"
$str5:
.asciiz "cada iteración\n"
$str6:
.asciiz "hola"
$str7:
.asciiz "El valor de b es 0\n"
_x:
.word 0
_y:
.word 0
_b:
.word 0
_z:
```

```
.word 0

#####
# Seccion de codigo
.text
.globl main
main:
    li $t0, 30
    sw $t0, _x
    li $t0, 10
    neg $t0, $t0
    sw $t0, _y
    li $t0, 3
    sw $t0, _b
    lw $t0, _x
    lw $t1, _y
    add $t0, $t0, $t1
    sw $t0, _z
    li $v0, 4
    la $a0, $str1
    syscall
    li $v0, 4
    la $a0, $str2
    syscall
    lw $t0, _b
    beqz $t0, $l1
    li $v0, 4
    la $a0, $str3
    syscall
$l1:
$l2:
    lw $t0, _b
    beqz $t0, $l3
    lw $t1, _b
    li $t2, 1
    sub $t1, $t1, $t2
    sw $t1, _b
    li $v0, 4
    la $a0, $str4
    syscall
    li $v0, 4
    la $a0, $str5
    syscall
    b $l2
$l3:
    lw $t0, _b
    beqz $t0, $l4
    li $v0, 4
    la $a0, $str6
    syscall
    b $l5
$l4:
    li $v0, 4
    la $a0, $str7
    syscall
$l5:

#####
# Fin
```

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

EJECUCIÓN

```
~/Compartida/MiniC2324> ./miniC pruebas/prueba_original.mc out> test.s
~/Compartida/MiniC2324> spim -file test.s
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
hola "esto es" una prueba
b es true porque su valor es distinto de 0
El valor de b va disminuyendo encada iteración
El valor de b va disminuyendo encada iteración
El valor de b va disminuyendo encada iteración
El valor de b es 0
```

4.3.2 PRUEBA JUNIO 2023

CÓDIGO MINIC

```
/******
 * Fichero de prueba junio 2023
 *****/
__junio_2023() {
    const a, b=1;
    var x54=(a+b*20-(b+19/2))/2,y;
    const c=10;
    var z;

    print ("Comienza simulación \\\ junio 2023\n");

    if (a*1000) print ("Esto no va bien con a*1000=",a*1000,"\n");

    if (b-1) print ("Esto no va bien con b-1=\n",b-1);

        else while (x54-2){
            x54 = x54-1;
            print ("Introduce valores de 'y' y 'z' 3 veces\n");
            read (y,z);
            print ("x54=",x54," y=", (y+0)/1, " z=", (z-0)*1, "\n");
        }
    z = 1;
    print ("Termina correctamente con x54=",x54,"\n");
}
```

CÓDIGO GENERADO MIPS

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

$str1:
.asciiz "Comienza simulación \\ junio 2023\n"
$str2:
.asciiz "Esto no va bien con a*1000="
$str3:
.asciiz "\n"
$str4:
.asciiz "Esto no va bien con b-1=\n"
$str5:
.asciiz "Introduce valores de 'y' y 'z' 3 veces\n"
$str6:
.asciiz "x54="
$str7:
.asciiz ", y="
$str8:
.asciiz ", z="
$str9:
.asciiz "\n"
$str10:
.asciiz "Termina correctamente con x54="
$str11:
.asciiz "\n"

_a:
.word 0
_b:
.word 0
_x54:
.word 0
_y:
.word 0
_c:
.word 0
_z:
.word 0

#####
# Seccion de codigo
.text
.globl main
main:
li $t0, 1
sw $t0, _b
lw $t0, _a
lw $t1, _b
li $t2, 20
mul $t1, $t1, $t2
add $t0, $t0, $t1
lw $t1, _b
li $t2, 19
li $t3, 2
div $t2, $t2, $t3
```



```

    add $t1, $t1, $t2
    sub $t0, $t0, $t1
    li $t1, 2
    div $t0, $t0, $t1
    sw $t0, _x54
    li $t0, 10
    sw $t0, _c
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _a
    li $t1, 1000
    mul $t0, $t0, $t1
    beqz $t0, $l1
    li $v0, 4
    la $a0, $str2
    syscall
    lw $t1, _a
    li $t2, 1000
    mul $t1, $t1, $t2
    li $v0, 1
    move $a0, $t1
    syscall
    li $v0, 4
    la $a0, $str3
    syscall
$l1:
    lw $t0, _b
    li $t1, 1
    sub $t0, $t0, $t1
    beqz $t0, $l4
    li $v0, 4
    la $a0, $str4
    syscall
    lw $t1, _b
    li $t2, 1
    sub $t1, $t1, $t2
    li $v0, 1
    move $a0, $t1
    syscall
    b $l5
$l4:
$l2:
    lw $t1, _x54
    li $t2, 2
    sub $t1, $t1, $t2
    beqz $t1, $l3
    lw $t2, _x54
    li $t3, 1
    sub $t2, $t2, $t3
    sw $t2, _x54
    li $v0, 4
    la $a0, $str5
    syscall
    li $v0, 5
    syscall
    sw $v0, _y
    li $v0, 5
    syscall

```

```

    sw $v0, _z
    li $v0, 4
    la $a0, $str6
    syscall
    lw $t2, _x54
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str7
    syscall
    lw $t2, _y
    li $t3, 0
    add $t2, $t2, $t3
    li $t3, 1
    div $t2, $t2, $t3
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str8
    syscall
    lw $t2, _z
    li $t3, 0
    sub $t2, $t2, $t3
    li $t3, 1
    mul $t2, $t2, $t3
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str9
    syscall
    b $l2
$l3:
$l5:
    li $t0, 1
    sw $t0, _z
    li $v0, 4
    la $a0, $str10
    syscall
    lw $t0, _x54
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str11
    syscall

#####
# Fin
    li $v0, 10
    syscall

```

EJECUCIÓN

```

~/Compartida/MiniC2324> ./miniC pruebas/ps.mc out> test.s 05/11/2024 08:34:29 PM
~/Compartida/MiniC2324> spim -file test.s 05/11/2024 08:34:32 PM
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
Comienza simulación \ junio 2023
Introduce valores de 'y' y 'z' 3 veces
1
1
x54=4, y=1, z=1
Introduce valores de 'y' y 'z' 3 veces
1
1
x54=3, y=1, z=1
Introduce valores de 'y' y 'z' 3 veces
1
1
x54=2, y=1, z=1
Termina correctamente con x54=2
~/Compartida/MiniC2324> | 05/11/2024 08:34:42 PM

```

4.3.3 PRUEBA BUCLE DO-WHILE

CÓDIGO MINIC

```

main(){
    var a= 3;

    print ("Primer do while ", a ,"\n");
    do
    {
        print ("El valor de a es ",a,"\n");
        a = a-1;
    } while (a);

    a =5;
    print ("\nSegundo con while ", a,"\n");
    while (a) { print ("El valor de a es ",a, "\n"); a = a-1; }
}

```

CÓDIGO GENERADO MIPS

```

#####
# Seccion de datos
.data

$str1:
.asciiz "Comienza simulación \ junio 2023\n"
$str2:
.asciiz "Esto no va bien con a*1000="
$str3:
.asciiz "\n"
$str4:

```

```

        .ascii "Esto no va bien con b-1=\n"
$str5:
        .ascii "Introduce valores de 'y' y 'z' 3 veces\n"
$str6:
        .ascii "x54="
$str7:
        .ascii ", y="
$str8:
        .ascii ", z="
$str9:
        .ascii "\n"
$str10:
        .ascii "Termina correctamente con x54="
$str11:
        .ascii "\n"
_a:
        .word 0
_b:
        .word 0
_x54:
        .word 0
_y:
        .word 0
_c:
        .word 0
_z:
        .word 0

#####
# Seccion de codigo
        .text
        .globl main
main:
        li $t0, 1
        sw $t0, _b
        lw $t0, _a
        lw $t1, _b
        li $t2, 20
        mul $t1, $t1, $t2
        add $t0, $t0, $t1
        lw $t1, _b
        li $t2, 19
        li $t3, 2
        div $t2, $t2, $t3
        add $t1, $t1, $t2
        sub $t0, $t0, $t1
        li $t1, 2
        div $t0, $t0, $t1
        sw $t0, _x54
        li $t0, 10
        sw $t0, _c
        li $v0, 4
        la $a0, $str1
        syscall
        lw $t0, _a
        li $t1, 1000
        mul $t0, $t0, $t1
        beqz $t0, $l1
        li $v0, 4

```

```

        la $a0, $str2
        syscall
        lw $t1, _a
        li $t2, 1000
        mul $t1, $t1, $t2
        li $v0, 1
        move $a0, $t1
        syscall
        li $v0, 4
        la $a0, $str3
        syscall
$l1:
        lw $t0, _b
        li $t1, 1
        sub $t0, $t0, $t1
        beqz $t0, $l4
        li $v0, 4
        la $a0, $str4
        syscall
        lw $t1, _b
        li $t2, 1
        sub $t1, $t1, $t2
        li $v0, 1
        move $a0, $t1
        syscall
        b $l5
$l4:
$l2:
        lw $t1, _x54
        li $t2, 2
        sub $t1, $t1, $t2
        beqz $t1, $l3
        lw $t2, _x54
        li $t3, 1
        sub $t2, $t2, $t3
        sw $t2, _x54
        li $v0, 4
        la $a0, $str5
        syscall
        li $v0, 5
        syscall
        sw $v0, _y
        li $v0, 5
        syscall
        sw $v0, _z
        li $v0, 4
        la $a0, $str6
        syscall
        lw $t2, _x54
        li $v0, 1
        move $a0, $t2
        syscall
        li $v0, 4
        la $a0, $str7
        syscall
        lw $t2, _y
        li $t3, 0
        add $t2, $t2, $t3
        li $t3, 1

```

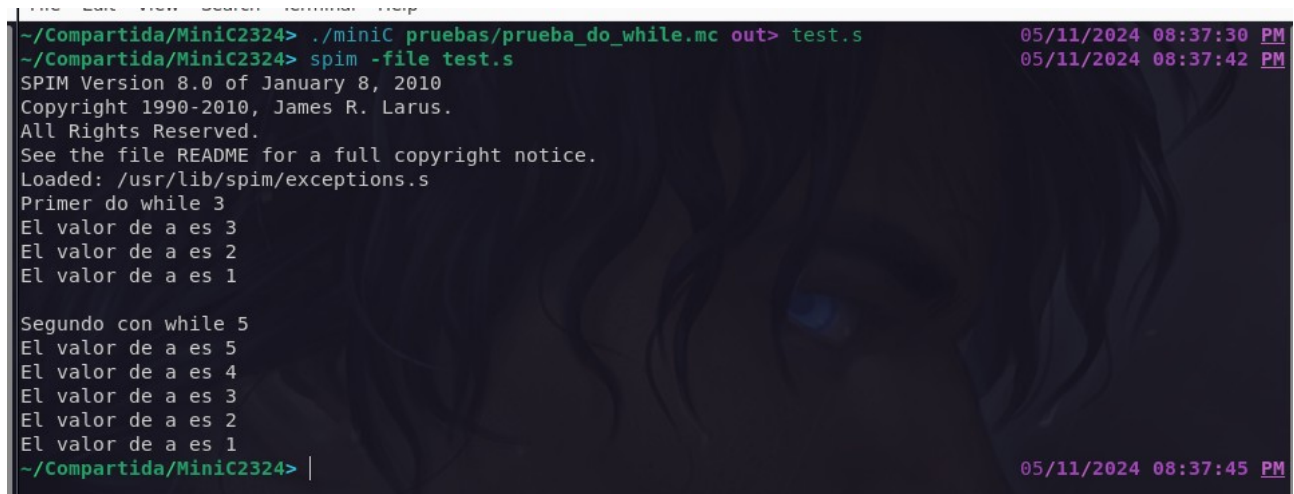
```

        div $t2, $t2, $t3
        li $v0, 1
        move $a0, $t2
        syscall
        li $v0, 4
        la $a0, $str8
        syscall
        lw $t2, _z
        li $t3, 0
        sub $t2, $t2, $t3
        li $t3, 1
        mul $t2, $t2, $t3
        li $v0, 1
        move $a0, $t2
        syscall
        li $v0, 4
        la $a0, $str9
        syscall
        b $l2
$l3:
$l5:
        li $t0, 1
        sw $t0, _z
        li $v0, 4
        la $a0, $str10
        syscall
        lw $t0, _x54
        li $v0, 1
        move $a0, $t0
        syscall
        li $v0, 4
        la $a0, $str11
        syscall

#####
# Fin
        li $v0, 10
        syscall

```

EJECUCIÓN



```

~/Compartida/MiniC2324> ./miniC pruebas/prueba_do_while.mc out> test.s 05/11/2024 08:37:30 PM
~/Compartida/MiniC2324> spim -file test.s 05/11/2024 08:37:42 PM
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
Primer do while 3
El valor de a es 3
El valor de a es 2
El valor de a es 1

Segundo con while 5
El valor de a es 5
El valor de a es 4
El valor de a es 3
El valor de a es 2
El valor de a es 1
~/Compartida/MiniC2324> | 05/11/2024 08:37:45 PM

```

4.3.4 PRUEBA OPERADORES RELACIONALES

CÓDIGO MINIC

```
relacional() {
    const a=0, b=0;
    var c=1, d=-1, e= 48, bool;

    print ("Probando menor\n");

    bool = c < e;
    print("\t",c, " < ",e, " = ",bool, "\n");
    bool = e < c;
    print("\t",e, " < ",c, " = ",bool, "\n");
    bool = d < e;
    print("\t",d, " < ",e, " = ",bool, "\n");
    bool = c < d;
    print("\t",c, " < ",d, " = ",bool, "\n");
    bool = c < c;
    print("\t",c, " < ",c, " = ",bool, "\n\n");

    print ("Probando mayor \n");

    bool = c > e;
    print("\t",c, " > ",e, " = ",bool, "\n");
    bool = e > c;
    print("\t",e, " > ",c, " = ",bool, "\n");
    bool = d > e;
    print("\t",d, " > ",e, " = ",bool, "\n");
    bool = c > d;
    print("\t",c, " > ",d, " = ",bool, "\n");
    bool = c > c;
    print("\t",c, " > ",c, " = ",bool, "\n\n");

    print ("Probando no igual\n");

    bool = c != e;
    print("\t",c, " != ",e, " = ",bool, "\n");
    bool = e != c;
    print("\t",e, " != ",c, " = ",bool, "\n");
    bool = d != e;
    print("\t",d, " != ",e, " = ",bool, "\n");
    bool = c != d;
    print("\t",c, " != ",d, " = ",bool, "\n");
    bool = c != c;
    print("\t",c, " != ",c, " = ",bool, "\n\n");

    print ("Probando mayor o igual\n");

    bool = c >= e;
    print("\t",c, " >= ",e, " = ",bool, "\n");
    bool = e >= c;
    print("\t",e, " >= ",c, " = ",bool, "\n");
    bool = d >= e;
    print("\t",d, " >= ",e, " = ",bool, "\n");
    bool = c >= d;
```

```

print("\t",c," >= ",d," = ",bool,"\n");
bool = c >= c;
print("\t",c," >= ",c," = ",bool,"\n\n");

print ("Probando menor o igual\n");

bool = c <= e;
print("\t",c," <= ",e," = ",bool,"\n");
bool = e <= c;
print("\t",e," <= ",c," = ",bool,"\n");
bool = d <= e;
print("\t",d," <= ",e," = ",bool,"\n");
bool = c <= d;
print("\t",c," <= ",d," = ",bool,"\n");
bool = c <= c;
print("\t",c," <= ",c," = ",bool,"\n\n");

print ("Probando igual\n");

bool = c == e;
print("\t",c," == ",e," = ",bool,"\n");
bool = e == c;
print("\t",e," == ",c," = ",bool,"\n");
bool = d == e;
print("\t",d," == ",e," = ",bool,"\n");
bool = c == d;
print("\t",c," == ",d," = ",bool,"\n");
bool = c == c ;
print("\t",c," == ",c," = ",bool,"\n\n");
}

```

CÓDIGO GENERADO MIPS

```

#####
# Seccion de datos
.data

$str1:
.asciiz "Comienza simulación \ junio 2023\n"
$str2:
.asciiz "Esto no va bien con a*1000="
$str3:
.asciiz "\n"
$str4:
.asciiz "Esto no va bien con b-1=\n"
$str5:
.asciiz "Introduce valores de 'y' y 'z' 3 veces\n"
$str6:
.asciiz "x54="
$str7:
.asciiz ", y="
$str8:
.asciiz ", z="
$str9:
.asciiz "\n"
$str10:
.asciiz "Termina correctamente con x54="

```



```

$str11:
    .asciiz "\n"
_a:
    .word 0
_b:
    .word 0
_x54:
    .word 0
_y:
    .word 0
_c:
    .word 0
_z:
    .word 0

#####
# Seccion de codigo
    .text
    .globl main
main:
    li $t0, 1
    sw $t0, _b
    lw $t0, _a
    lw $t1, _b
    li $t2, 20
    mul $t1, $t1, $t2
    add $t0, $t0, $t1
    lw $t1, _b
    li $t2, 19
    li $t3, 2
    div $t2, $t2, $t3
    add $t1, $t1, $t2
    sub $t0, $t0, $t1
    li $t1, 2
    div $t0, $t0, $t1
    sw $t0, _x54
    li $t0, 10
    sw $t0, _c
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _a
    li $t1, 1000
    mul $t0, $t0, $t1
    beqz $t0, $l1
    li $v0, 4
    la $a0, $str2
    syscall
    lw $t1, _a
    li $t2, 1000
    mul $t1, $t1, $t2
    li $v0, 1
    move $a0, $t1
    syscall
    li $v0, 4
    la $a0, $str3
    syscall
$l1:
    lw $t0, _b

```

```

    li $t1, 1
    sub $t0, $t0, $t1
    beqz $t0, $l4
    li $v0, 4
    la $a0, $str4
    syscall
    lw $t1, _b
    li $t2, 1
    sub $t1, $t1, $t2
    li $v0, 1
    move $a0, $t1
    syscall
    b $l5
$l4:
$l2:
    lw $t1, _x54
    li $t2, 2
    sub $t1, $t1, $t2
    beqz $t1, $l3
    lw $t2, _x54
    li $t3, 1
    sub $t2, $t2, $t3
    sw $t2, _x54
    li $v0, 4
    la $a0, $str5
    syscall
    li $v0, 5
    syscall
    sw $v0, _y
    li $v0, 5
    syscall
    sw $v0, _z
    li $v0, 4
    la $a0, $str6
    syscall
    lw $t2, _x54
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str7
    syscall
    lw $t2, _y
    li $t3, 0
    add $t2, $t2, $t3
    li $t3, 1
    div $t2, $t2, $t3
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str8
    syscall
    lw $t2, _z
    li $t3, 0
    sub $t2, $t2, $t3
    li $t3, 1
    mul $t2, $t2, $t3
    li $v0, 1

```

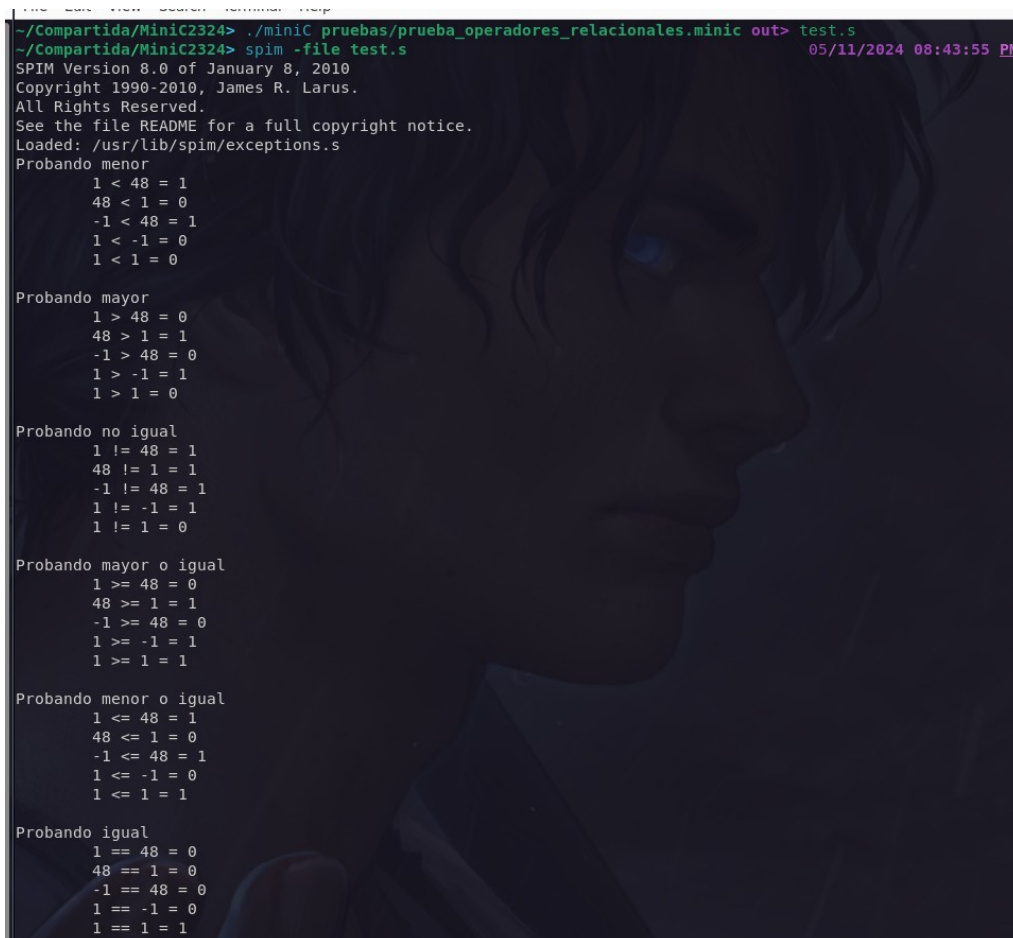
```

        move $a0, $t2
        syscall
        li $v0, 4
        la $a0, $str9
        syscall
        b $l2
$l3:
$l5:
        li $t0, 1
        sw $t0, _z
        li $v0, 4
        la $a0, $str10
        syscall
        lw $t0, _x54
        li $v0, 1
        move $a0, $t0
        syscall
        li $v0, 4
        la $a0, $str11
        syscall

#####
# Fin
        li $v0, 10
        syscall

```

EJECUCIÓN



```

~/Compartida/MiniC2324> ./minic pruebas/prueba_operadores_relacionales.minic out> test.s
~/Compartida/MiniC2324> spim -file test.s
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
Probando menor
    1 < 48 = 1
    48 < 1 = 0
    -1 < 48 = 1
    1 < -1 = 0
    1 < 1 = 0
Probando mayor
    1 > 48 = 0
    48 > 1 = 1
    -1 > 48 = 0
    1 > -1 = 1
    1 > 1 = 0
Probando no igual
    1 != 48 = 1
    48 != 1 = 1
    -1 != 48 = 1
    1 != -1 = 1
    1 != 1 = 0
Probando mayor o igual
    1 >= 48 = 0
    48 >= 1 = 1
    -1 >= 48 = 0
    1 >= -1 = 1
    1 >= 1 = 1
Probando menor o igual
    1 <= 48 = 1
    48 <= 1 = 0
    -1 <= 48 = 1
    1 <= -1 = 0
    1 <= 1 = 1
Probando igual
    1 == 48 = 0
    48 == 1 = 0
    -1 == 48 = 0
    1 == -1 = 0
    1 == 1 = 1

```

4.3.5 PRUEBA BUCLE FOR

CÓDIGO MINIC

```
prueba_for(){
    var a, b;

    print("Probando for simple de 1 a 5\n");

    for (a = 1 to 5){
        print("\tEl valor actual de a es ", a, "\n");
    }

    print("Probando for simple de 5 a 1\n");
    for (a =5 to 1){
        print("\tEl valor actual de a es ", a, "\n");
    }
    if(a==5) print("Fin For funciona correctamente, valor de a ", a,"\n");

    print("Prueba for condicion de 1 a 5\n");
    for (a = 1; a > 5){
        print("\tEl valor actual de a es ", a, "\n");
    }
    print("Prueba for condicion de con b\n");
    b = 0;
    for (a = 1; b){
        print("\tEl valor actual de a es ", a, "\n");
        if (a == 3) b = 1;
    }
    print("El valor de b es ",b, "\n");
}
```

CÓDIGO GENERADO MIPS

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

$str1:
.asciiz "Probando for simple de 1 a 5\n"
$str2:
.asciiz "\tEl valor actual de a es "
$str3:
.asciiz "\n"
$str4:
.asciiz "Probando for simple de 5 a 1\n"
$str5:
.asciiz "\tEl valor actual de a es "
$str6:
.asciiz "\n"
$str7:
.asciiz "Fin For funciona correctamente, valor de a "
$str8:
.asciiz "\n"
$str9:
.asciiz "Prueba for condicion de 1 a 5\n"
$str10:
```

```

        .asciiz "\tEl valor actual de a es "
$str11:
        .asciiz "\n"
$str12:
        .asciiz "Prueba for condicion de con b\n"
$str13:
        .asciiz "\tEl valor actual de a es "
$str14:
        .asciiz "\n"
$str15:
        .asciiz "El valor de b es "
$str16:
        .asciiz "\n"
_a:
        .word 0
_b:
        .word 0

#####
# Seccion de codigo
        .text
        .globl main
main:
        li $v0, 4
        la $a0, $str1
        syscall
        li $t0, 1
        sw $t0, _a
$l1:
        li $t1, 5
        bgt $t0, $t1, $l2
        li $v0, 4
        la $a0, $str2
        syscall
        lw $t2, _a
        li $v0, 1
        move $a0, $t2
        syscall
        li $v0, 4
        la $a0, $str3
        syscall
        addi $t0, $t0, 1
        sw $t0, _a
        b $l1
$l2:
        li $v0, 4
        la $a0, $str4
        syscall
        li $t1, 5
        sw $t1, _a
$l3:
        li $t2, 1
        bgt $t1, $t2, $l4
        li $v0, 4
        la $a0, $str5
        syscall
        lw $t3, _a
        li $v0, 1
        move $a0, $t3

```

```
    syscall
    li $v0, 4
    la $a0, $str6
    syscall
    addi $t1, $t1, 1
    sw $t1, _a
    b $l3
$l14:
    lw $t2, _a
    li $t3, 5
    xor $t2, $t2, $t3
    sltu $t2, $zero, $t2
    xori $t2, $t2, 1
    beqz $t2, $l15
    li $v0, 4
    la $a0, $str7
    syscall
    lw $t2, _a
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 4
    la $a0, $str8
    syscall
$l15:
    li $v0, 4
    la $a0, $str9
    syscall
    li $t2, 1
    sw $t2, _a
$l16:
    lw $t3, _a
    li $t4, 5
    slt $t3, $t4, $t3
    bnez $t3, $l17
    li $v0, 4
    la $a0, $str10
    syscall
    lw $t3, _a
    li $v0, 1
    move $a0, $t3
    syscall
    li $v0, 4
    la $a0, $str11
    syscall
    addi $t2, $t2, 1
    sw $t2, _a
    b $l6
$l17:
    li $v0, 4
    la $a0, $str12
    syscall
    li $t3, 0
    sw $t3, _b
    li $t3, 1
    sw $t3, _a
$l19:
    lw $t4, _b
    bnez $t4, $l10
```

```
    li $v0, 4
    la $a0, $str13
    syscall
    lw $t5, _a
    li $v0, 1
    move $a0, $t5
    syscall
    li $v0, 4
    la $a0, $str14
    syscall
    lw $t5, _a
    li $t6, 3
    xor $t5, $t5, $t6
    sltu $t5, $zero, $t5
    xori $t5, $t5, 1
    beqz $t5, $l8
    li $t5, 1
    sw $t5, _b
$l8:
    addi $t3, $t3, 1
    sw $t3, _a
    b $l9
$l10:
    li $v0, 4
    la $a0, $str15
    syscall
    lw $t4, _b
    li $v0, 1
    move $a0, $t4
    syscall
    li $v0, 4
    la $a0, $str16
    syscall

#####
# Fin
    li $v0, 10
    syscall
```

EJECUCIÓN

```
~/Compartida/MiniC2324> ./miniC pruebas/prueba_for_completa.mc out> test.s 05/11/2024 08:54:00 PM
~/Compartida/MiniC2324> spim -file test.s 05/11/2024 08:54:02 PM
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
Probando for simple de 1 a 5
    El valor actual de a es 1
    El valor actual de a es 2
    El valor actual de a es 3
    El valor actual de a es 4
    El valor actual de a es 5
Probando for simple de 5 a 1
Fin For funciona correctamente, valor de a 5
Prueba for condicion de 1 a 5
    El valor actual de a es 1
    El valor actual de a es 2
    El valor actual de a es 3
    El valor actual de a es 4
    El valor actual de a es 5
Prueba for condicion de con b
    El valor actual de a es 1
    El valor actual de a es 2
    El valor actual de a es 3
El valor de b es 1
~/Compartida/MiniC2324> | 05/11/2024 08:54:05 PM
```

5 EXTENSIONES

5.1 BUCLE DO-WHILE

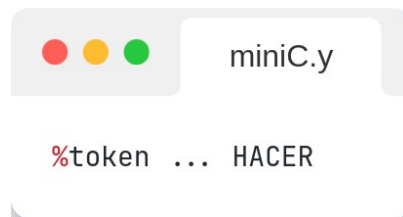
Cuando hablamos de un bucle do-while esencialmente lo que queremos es un bucle while cuyo contenido se ejecute al menos una vez. Si pretendemos implementar uno de estos, no existe mejor “fuente de inspiración” para el lenguaje MiniC que el propio C.

```
do {
    // Código a ejecutar al menos una vez
} while (condición);
```

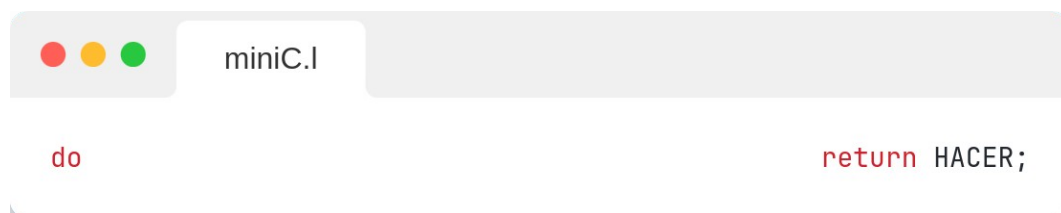
En C, la forma de expresar este bucle es con la palabra reservada `do`, seguido del código a ejecutar, continua la palabra clave `while`, seguida de la condición del bucle y termina con un punto y coma. Esta estructura se puede pasar de manera sencilla a una sentencia sintáctica. El único token que se tendría que añadir al lenguaje para que fuera reconocible por Bison sería el relacionado con `do`, el cual llamaremos HACER.

5.1.1 EDICIÓN LÉXICO

Dentro del fichero miniC.y hace falta declarar el token HACER.



Mientras que el fichero miniC.l se le tiene que añadir la palabra reservada para que sea reconocida.

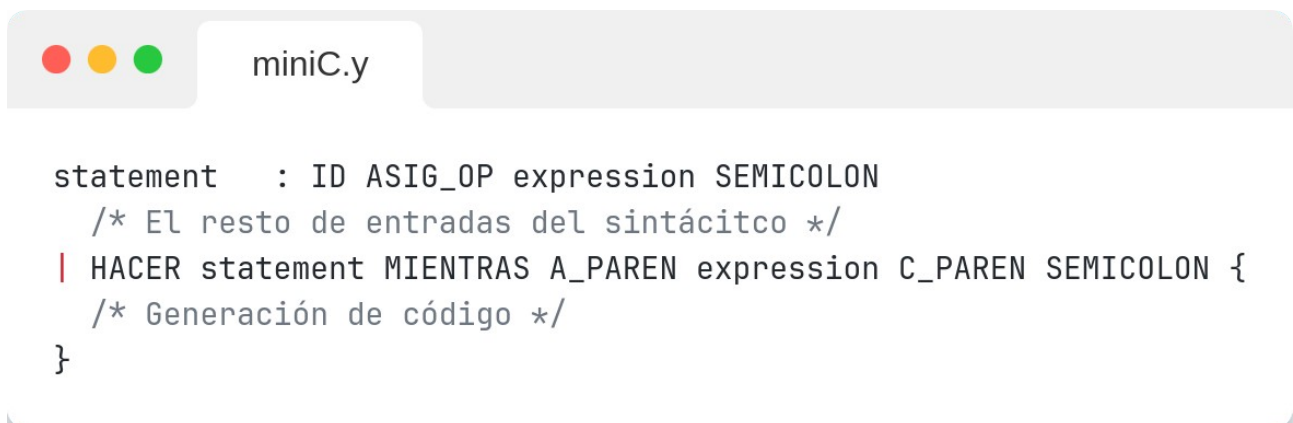


5.1.2 EDICIÓN SINTÁCTICO

En el sintáctico tendremos especificar los token a utilizar y el orden. Serían: HACER para el `do`; el no terminal *statement* para el código a ejecutar; MIENTRAS para el *while*, como hemos hecho en el bucle mientras regular; el no terminal *expression* para la condición, que va a estar entre paréntesis, por lo que tendremos `A_PAREN` y `C_PAREN` rodeándola; y el token SEMICOLON (el `;`) para indicar el final de la sentencia.

HACER statement MIENTRAS A_PAREN expression C_PAREN SEMICOLON

Esta entrada sería añadida como parte derecha de un no terminal *statement*, al mismo nivel que el otro bucle añadido. Como no contiene ningún uso del token ID, no es necesario ninguna implementación en el semántico.



5.1.3 GENERACIÓN DE CÓDIGO

Cuando queremos traducir esta sentencia a ensamblador MIPS, resulta relativamente simple. La idea sería tener una etiqueta antes de la ejecución del código relacionado con statement y una instrucción de salto que vuelve al inicio del bucle si el código relacionado con la expression es “cierto”, es decir, diferente de cero. La estructura sería la siguiente:



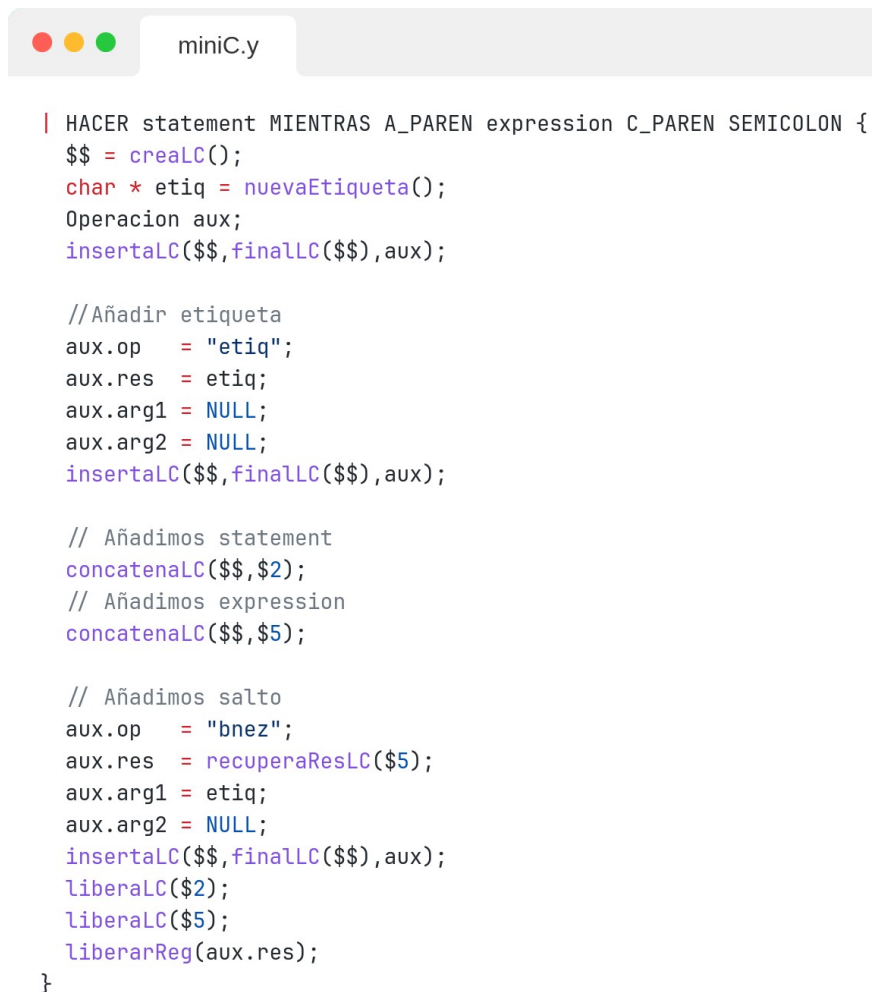
```

etiqueta:
    # Código de statement

    # Código de expression

    bnez resultadoExpression, etiqueta
  
```

Estructura del código en “pseudoMIPS”



```

| HACER statement MIENTRAS A_PAREN expression C_PAREN SEMICOLON {
    $$ = creaLC();
    char * etiq = nuevaEtiqueta();
    Operacion aux;
    insertaLC($$,finalLC($$),aux);

    //Añadir etiqueta
    aux.op = "etiq";
    aux.res = etiq;
    aux.arg1 = NULL;
    aux.arg2 = NULL;
    insertaLC($$,finalLC($$),aux);

    // Añadimos statement
    concatenaLC($$, $2);
    // Añadimos expression
    concatenaLC($$, $5);

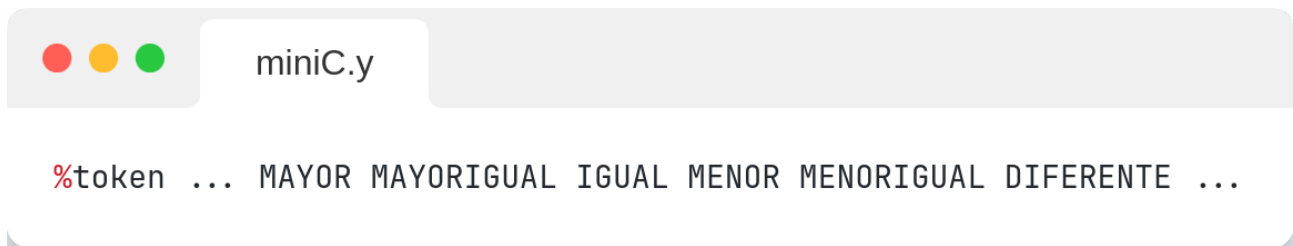
    // Añadimos salto
    aux.op = "bnez";
    aux.res = recuperaResLC($5);
    aux.arg1 = etiq;
    aux.arg2 = NULL;
    insertaLC($$,finalLC($$),aux);
    liberaLC($2);
    liberaLC($5);
    liberarReg(aux.res);
}
  
```

5.2 OPERADORES RELACIONALES

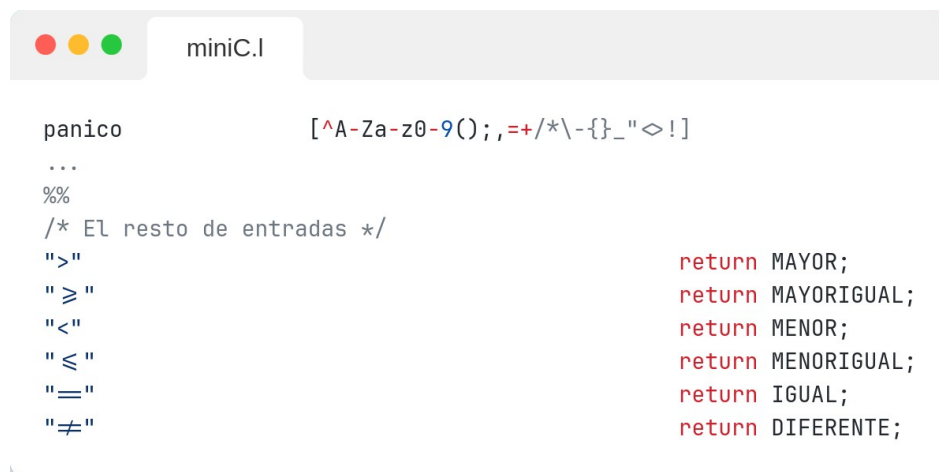
Los operadores relacionales (<, >, ==, !=, <= y >=) en los lenguajes de programación típicamente sirven para comparar entre elementos (a veces solo números otras también objetos, etc) y suelen devolver un booleano o un número cuando esto no existen como el es caso de C. En MiniC, como solo existen los tipos entero y string, solo vamos a poder comparar este primer tipo, y como no tenemos booleano también utilizaremos enteros al igual que C.

5.2.1 EDICIÓN LÉXICO

Cada operador relacional va a necesitar una representación. Para ellos vamos a añadir un token al fichero *miniC.y* por operador.



Una vez declarado el token, tenemos que especificar cual será su representación en un programa, esto se especifica en el fichero *miniC.l*. Como se tratan de símbolos, no tenemos que preocuparnos por patrones solo añadir el su representación. Sin embargo, como este introduce nuevos símbolos a la gramática (<,> y !), se tiene que actualizar la macro pánico.

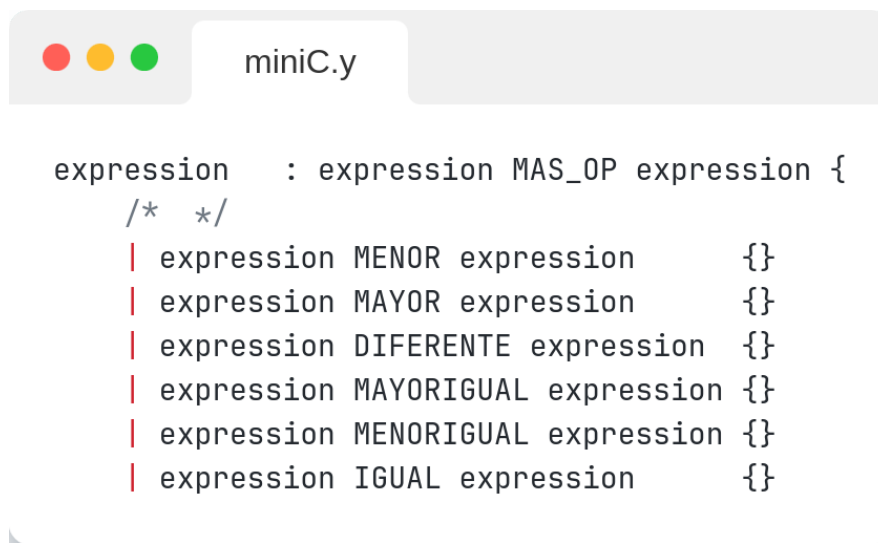


5.2.2 EDICIÓN SINTÁCTICO

Estos operadores son parecidos a los operadores matemáticos (+,-,*,/) en el sentido de que tienen comportamientos similares. Ambos tienen la estructura num-operador-num y devuelven otro número. Por lo tanto cuando queramos describir la estructura de una de estas operaciones, y poniendo al operador menor que como ejemplo, esta sería la siguiente:

$$N < N$$

Por la misma razón por la que los operadores matemáticos se colocan en el no terminal expression, se ponen las reglas de las operaciones de relacionales en el mismo nivel.

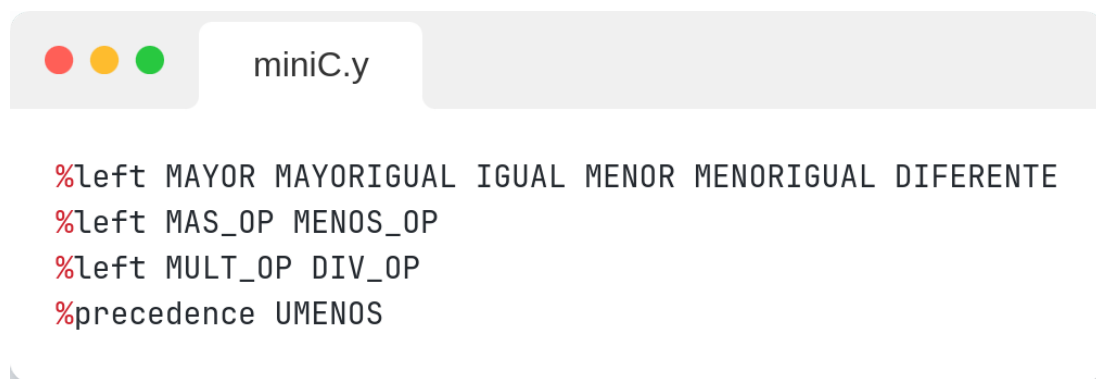


```

expression    : expression MAS_OP expression {
                /* */
                | expression MENOR expression   {}
                | expression MAYOR expression  {}
                | expression DIFERENTE expression {}
                | expression MAYORIGUAL expression {}
                | expression MENORIGUAL expression {}
                | expression IGUAL expression   {}

```

Sin embargo, como todos se encuentran en el mismo nivel y las operaciones matemáticas tienen un orden de precedencia, estas también se tienen que añadir a la jerarquía para asegurarnos de que cumple con el comportamiento adecuado. Vamos a decidir ponerlo como los operadores con la menor precedencia. Si no encontramos en la situación donde a un lado hay un número y al otro una suma de dos números, tiene más sentido que se sume primero y que el resultado se compare con el otro número que se compare primero y el resultado (un cero si es falso o un uno si es verdadero) se sume.



```

%left MAYOR MAYORIGUAL IGUAL MENOR MENORIGUAL DIFERENTE
%left MAS_OP MENOS_OP
%left MULT_OP DIV_OP
%precedence Umenos

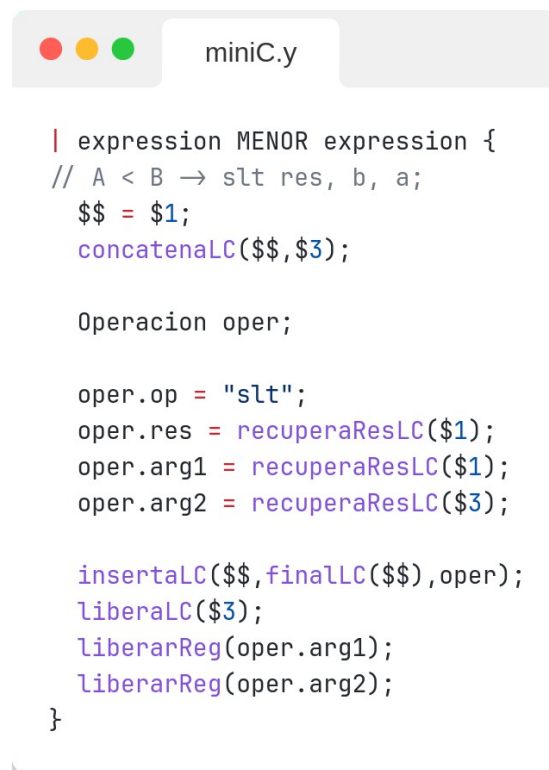
```

5.2.3 GENERACIÓN DE CÓDIGO

A partir de aquí, podemos decir que las operaciones son simplemente obtener el valor de su contraria e invertir el resultado. Para invertir estos valores vamos a realizar una operación XOR con el número 1, invertirla completamente sería con un valor todo a 1s (INT_MAX) pero nosotros solo queremos cambiar la última cifra binaria. La operación de MIPS utilizada es XORI, porque el número 1 no va a ser guardado en ningún registro.

MENOR QUE

Para hacer una operación de menor, simplemente podemos utilizar la que contiene MIPS. Sin embargo, no contiene ninguna de las otras y vamos a tener que utilizar operadores lógico si queremos alcanzar el mismo resultado.



```
| expression MENOR expression {  
// A < B → slt res, b, a;  
$$ = $1;  
concatenaLC($$, $3);  
  
Operacion oper;  
  
oper.op = "slt";  
oper.res = recuperaResLC($1);  
oper.arg1 = recuperaResLC($1);  
oper.arg2 = recuperaResLC($3);  
  
insertaLC($$, finalLC($$), oper);  
liberaLC($3);  
liberarReg(oper.arg1);  
liberarReg(oper.arg2);  
}
```

MAYOR QUE

La operación mayor es exactamente la inversión de menor, por lo que podemos invertir el orden de los valores en la operación, es decir, como $a > b$ es igual que $b < a$, calculamos este último porque es más sencillo.



```
| expression MAYOR expression {  
// A > B → B < A → slt res, b, a;  
    $$ = $1;  
    concatenaLC($$, $3);  
    Operacion oper;  
    oper.op = "slt";  
    oper.res = recuperaResLC($1);  
    oper.arg1 = recuperaResLC($3);  
    oper.arg2 = recuperaResLC($1);  
    insertaLC($$, finalLC($$), oper);  
    liberaLC($3);  
    liberarReg(oper.arg1);  
    liberarReg(oper.arg2);  
}
```

DIFERENTE

Como no existe una operación directa, utilizaremos las operaciones *XOR* y *SLTU* para obtenerlo. Para empezar se hace un *XOR* de los dos números, si estos son iguales es resultado será cero y si no lo son, dará un número diferente. Para convertir esto en un "booleano" (que solo sea 0 o 1), se utiliza *SLTU*. Comparamos si cero (con el registro *\$zero*) es menor que el resultado, y en caso de que lo sea, podemos afirmar que los números son diferentes. Es importante que la instrucción usada sea *SLTU* y no *SLT* porque el resultado de *XOR* podría ser negativo cuyo valor sería menor que cero, sin embargo, su valor absoluto no lo es.



```
| expression DIFERENTE expression {  
  // A ≠ B → xor res1, a, b; sltu res2, $zero, res1;  
  $$ = $1;  
  concatenaLC($$, $3);  
  Operacion oper1;  
  Operacion oper2;  
  
  // XOR  
  oper1.op = "xor";  
  oper1.res = recuperaResLC($1);  
  oper1.arg1 = recuperaResLC($1);  
  oper1.arg2 = recuperaResLC($3);  
  insertaLC($$, finalLC($$), oper1);  
  
  // SLTU  
  oper2.op = "sltu";  
  oper2.res = recuperaResLC($1);  
  oper2.arg1 = "$zero";  
  oper2.arg2 = recuperaResLC($1);  
  insertaLC($$, finalLC($$), oper2);  
  
  liberaLC($3);  
  liberarReg(oper1.arg1);  
  liberarReg(oper1.arg2);  
}
```

MAYOR QUE

En este caso MAYORIGUAL es la contraria de MENOR, por lo que calcularemos el menor e invertiremos el resultado de este.

```
miniC.y

| expression MAYORIGUAL expression {
//  $A \geq B \rightarrow \neg(A < B) \rightarrow \text{slt res, b, a; xori res, res, 1;}$ 
  $$ = $1;
  concatenaLC($$, $3);
  Operacion oper1, oper2;
  // SLT
  oper1.op = "slt";
  oper1.res = recuperaResLC($1);
  oper1.arg1 = recuperaResLC($1);
  oper1.arg2 = recuperaResLC($3);
  insertaLC($$, finalLC($$), oper1);
  // XORI
  oper2.op = "xori";
  oper2.res = recuperaResLC($1);
  oper2.arg1 = recuperaResLC($1);
  oper2.arg2 = "1";
  insertaLC($$, finalLC($$), oper2);

  liberaLC($3);
  liberarReg(oper1.arg1);
  liberarReg(oper1.arg2);
}
```


MENOR QUE

En este caso MENORIGUAL es la contraria de MAYOR, por lo que calcularemos el menor e invertiremos el resultado de este.

```
miniC.y

| expression MENORIGUAL expression {
//  $A \leq B \rightarrow !(A > B) \rightarrow !(B < A) \rightarrow \text{slt res, b, a; xori res, res, 1;}$ 
  $$ = $1;
  concatenaLC($$, $3);
  Operacion oper1;
  Operacion oper2;

  // SLT
  oper1.op = "slt";
  oper1.res = recuperaResLC($1);
  oper1.arg1 = recuperaResLC($3);
  oper1.arg2 = recuperaResLC($1);
  insertaLC($$, finalLC($$), oper1);

  // XORI
  oper2.op = "xori";
  oper2.res = recuperaResLC($1);
  oper2.arg1 = recuperaResLC($1);
  oper2.arg2 = "1";
  insertaLC($$, finalLC($$), oper2);

  liberaLC($3);
  liberarReg(oper1.arg1);
  liberarReg(oper1.arg2);
}
```

IGUAL

En este caso MENORIGUAL es la contraria de MAYOR, por lo que calcularemos el menor e invertiremos el resultado de este.

```

miniC.y

| expression IGUAL expression {
// A = B → !(A ≠ B) → xor res, a, b
//                               sltu res, $zero, res
//                               xori res, res, 1
    $$ = $1;
    concatenaLC($$, $3);
    Operacion oper1;
    Operacion oper2;
    Operacion oper3;

    // XOR
    oper1.op = "xor";
    oper1.res = recuperaResLC($1);
    oper1.arg1 = recuperaResLC($1);
    oper1.arg2 = recuperaResLC($3);
    insertaLC($$, finalLC($$), oper1);

    // SLTU
    oper2.op = "sltu";
    oper2.res = recuperaResLC($1);
    oper2.arg1 = "$zero";
    oper2.arg2 = recuperaResLC($1);
    insertaLC($$, finalLC($$), oper2);

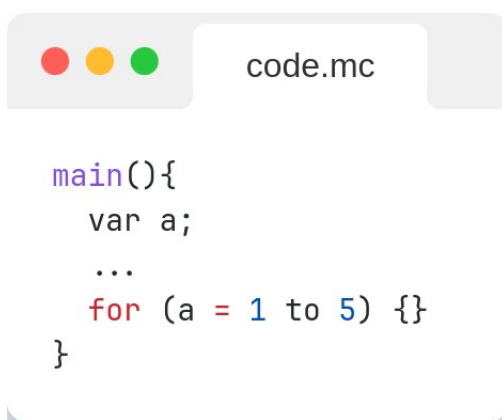
    // XORI
    oper3.op = "xori";
    oper3.res = recuperaResLC($1);
    oper3.arg1 = recuperaResLC($1);
    oper3.arg2 = "1";
    insertaLC($$, finalLC($$), oper3);

    liberaLC($3);
    liberarReg(oper1.arg1);
    liberarReg(oper1.arg2);
}

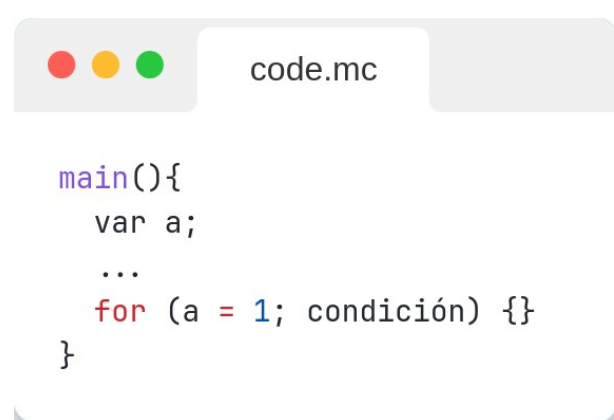
```

5.3 BUCLE FOR

Cuando hablamos de un bucle for, esencialmente queremos un bucle que recorra secuencias de elementos, ya sean los números desde un número hasta otro o los elementos de una lista. En nuestro caso, el bucle tiene que recorrer una secuencia de números desde un mínimo hasta un máximo o hasta que se cumpla una condición especificada por el usuario mientras en cada operación se va incrementando el número en uno en cada iteración (no se ha implementado que el valor a incrementar sea elegido por el programador). Para diferenciar cuando se tiene que llegar hasta un número y cuando la opción la elige el programador, vamos a crear dos maneras diferentes, aunque parecidas, para declarar uno.



```
main(){  
  var a;  
  ...  
  for (a = 1 to 5) {}  
}
```



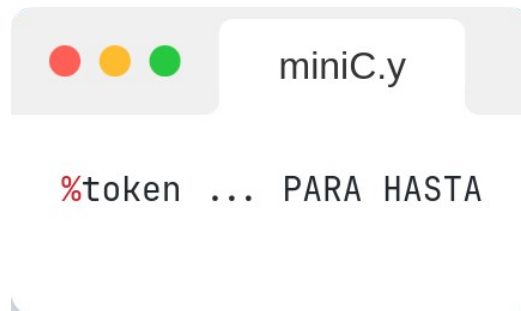
```
main(){  
  var a;  
  ...  
  for (a = 1; condición) {}  
}
```

El código de la imagen superior izquierda se utilizaría para un valor máximo. Con este valor inclusive. Mientras que la otra imagen sería para condición del programador. Hay que tener cuidado cuando se define esta condición porque de no poder cumplirse se provocaría un bucle “infinito” que provocaría un overflow con el valor de la variable de iterador.

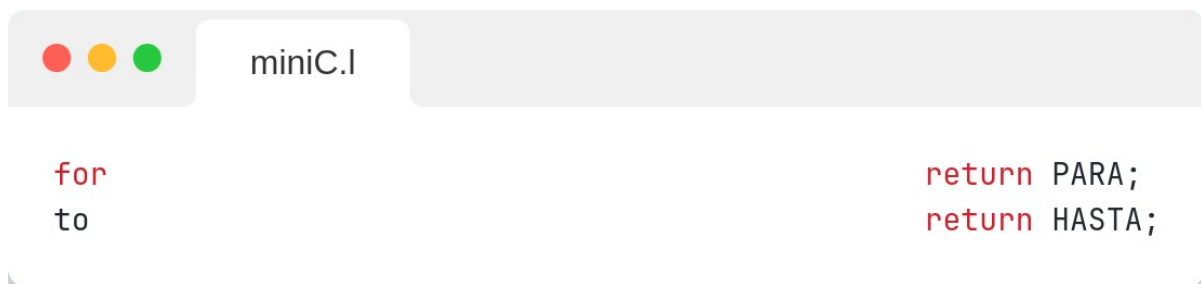
Cabe recordar que siguiendo el enunciado la condición del segundo for no es una condición de continuar como en los otros bucle, sino una condición de parada, es decir, que cuando sea cierta, se termina el bucle.

5.3.1 EDICIÓN LÉXICO

Para empezar tenemos que declarar los tokens que necesitamos. En nuestro caso, como hemos visto en el ejemplo de estructura, vamos a necesitar dos palabras reservadas una para el *for* y otra para el *to*, en nuestro caso hemos elegido PARA para el *for* y HASTA para el *to*.



Añadimos las palabras clave en un a entrada en el fichero miniC.l



5.3.2 EDICIÓN SINTÁCTICO

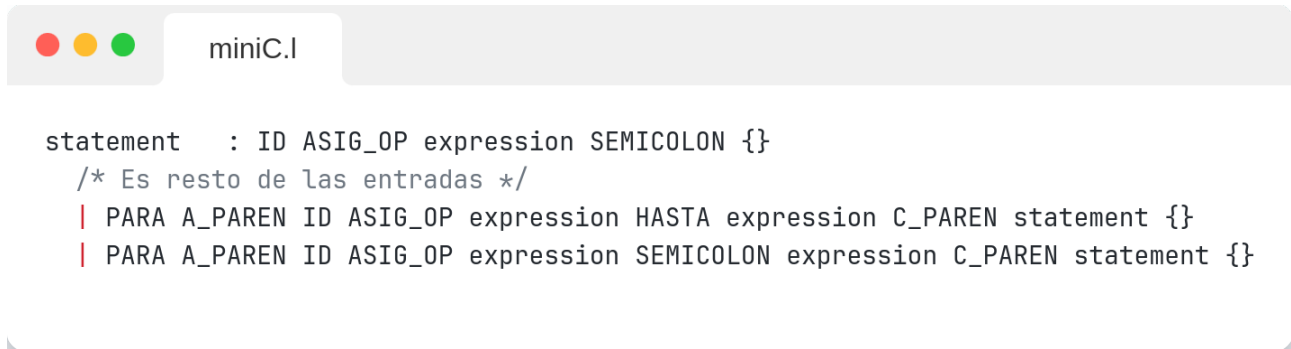
El bucle *for* no difiere tanto de los otros bucles, de hecho es bien conocido que en C, este es esencialmente un bucle *while* pero expresado de manera diferente para que quede más claro al programador que se está recorriendo una secuencia en vez de simplemente iterando. De hecho, se desaconseja utilizar un bucle *for* si durante la ejecución de este existe alguna condición que pueda terminar el bucle antes de terminar de recorrer la secuencia deseada. Así que, si lo pensamos, la estructura de un bucle *for* es la de un *while*, que además contiene una asignación:

Además, si pensamos en la forma de expresar ambos tipos de *for*, podemos observar que son bastante similares y que solo difieren en un token que separa la expresión de asignación con el de la condición o máximo, es decir si el token puesto antes de la segunda expresión es un HASTA sería un máximo pero si es un SEMICOLON es una condición de salida.

La estructura sería la siguiente, separamos con / entre el HASTA y SEMICOLON porque son lo mismo:

PARA A_PAREN ID ASIG_OP expression HASTA/SEMICOLON expresión C_PAREN statement

De esta manera se añadirán dos entradas como reglas de *statement* al igual que los otros bucles.



```

statement  : ID ASIG_OP expression SEMICOLON {}
/* Es resto de las entradas */
| PARA A_PAREN ID ASIG_OP expression HASTA expression C_PAREN statement {}
| PARA A_PAREN ID ASIG_OP expression SEMICOLON expression C_PAREN statement {}

```

5.3.3 EDICIÓN SEMÁNTICO

Como la estructura contiene una asignación, tenemos que comprobar exactamente las mismas cosas en el semántico.



```

| PARA A_PAREN ID ASIG_OP expression HASTA expression C_PAREN statement {
    if (!perteneceTablas(lista, $3)) {
        printf("Variable %s no declarada \n", $3);
        numErroresSem++;
    } else if (esConstante(lista, $3)) {
        printf("Asignación a constante\n");
        numErroresSem++;
    }
    /* Generación de código */
}

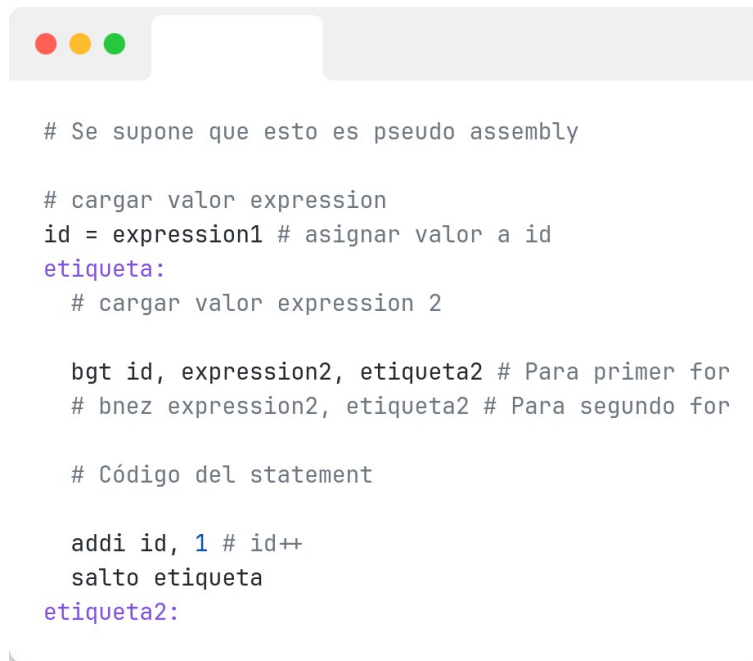
```

5.3.4 GENERACIÓN DE CÓDIGO

Como hemos comentado antes la el bucle for es esencialmente un bucle while, por lo que podemos inspirarnos en la estructura de código de este para la del for. Coincidimos en que el código se debe encontrar entre dos etiquetas y que tiene que tener dos saltos, el primero condicional que cuando la condición sea falsa salta a la segunda etiqueta para terminar la etiqueta y inmediatamente antes de esta segunda etiqueta para volver al inicio al final de una iteración.

Sin embargo, en lo que difieren, es que debe haber una asignación al identificador del valor de la expresión inicial, que se debe aumentar el valor del identificador en cada iteración inmediatamente antes del salto incondicional y que la condición de parada es

diferente. La condición de parada es lo que diferencia los tipos de for entre ellos siendo la primera que el identificador sea mayor que la segunda expresión, mientras que el segundo tipo es que esta segunda sea diferente de cero.



Estructura del código en “pseudo-MIPS”

FOR SIMPLE

```

| PARA A_PAREN ID ASIG_OP expression HASTA expression C_PAREN statement {
    // id = expression inicial
    // etiqueta1:
    // evaluar expression final
    // bgt id, expression final-> salto etiqueta2
    //
    // statement
    //
    // id++
    // salto etiqueta1
    // etiqueta2:

    if (!perteneceTablaS(lista, $3)) {
        printf("Variable %s no declarada \n", $3);
        numErroresSem++;
    } else if (esConstante(lista, $3)) {
        printf("Asignación a constante\n");
        numErroresSem++;
    }

    char * etiq1=nuevaEtiqueta();
    char * etiq2=nuevaEtiqueta();
    char * registroId;

    $$=creaLC();

    Operacion aux;
  
```

```
concatenaLC($$, $5);
// Asignar valor de $5 a ID
registroId=recuperaResLC($5);
aux.op="sw";
aux.res=registroId;
aux.arg1=concatena("_", $3);
aux.arg2=NULL;

insertaLC($$, finalLC($$), aux);

//Añadir etiqueta 1
aux.op="eti";
aux.res=eti1;
aux.arg1=NULL;
aux.arg2=NULL;

insertaLC($$, finalLC($$), aux);

concatenaLC($$, $7);
aux.op="bgt";
aux.res=registroId;
aux.arg1=recuperaResLC($7);
aux.arg2=eti2;

//Liberar lc
liberaLC($7);

//Insertar codigo
insertaLC($$, finalLC($$), aux);

liberarReg(aux.arg1);

//Añadir statement
concatenaLC($$, $9);

//Liberar codigo
liberaLC($9);

//// Sumar id++
// Sumar 1 al registro reservado
aux.op="addi";
aux.res=registroId;
aux.arg1=registroId;
aux.arg2="1";

insertaLC($$, finalLC($$), aux);
// Guardar en variable de ID
aux.op="sw";
aux.res=registroId;
aux.arg1=concatena("_", $3);
aux.arg2=NULL;

insertaLC($$, finalLC($$), aux);
//Añadir salto a etiqueta 1
aux.op="b";
aux.res=eti1;
aux.arg1=NULL;
aux.arg2=NULL;
```

```

//Insertar codigo
insertaLC($$,finalLC($$),aux);

//Añadir etiqueta 2
aux.op = "eti";
aux.res = eti2;
aux.arg1=NULL;
aux.arg2=NULL;
insertaLC($$,finalLC($$),aux);
}

```

FOR CONDICIÓN MANUAL

```

| PARA A_PAREN ID ASIG_OP expression SEMICOLON expression C_PAREN statement {
    // id = expression inicial
    // etiqueta1:
    // bgt id, expression final-> salto etiqueta2
    // beqz , expression final-> salto etiqueta2
    //
    //     statement
    //
    //     id++
    //     salto etiqueta1
    // etiqueta2:

    if (!perteneceTablaS(lista, $3)) {
        printf("Variable %s no declarada \n",$3);
        numErroresSem++;
    } else if (esConstante(lista, $3)) {
        printf("Asignación a constante\n");
        numErroresSem++;
    }

    char * eti1=nuevaEtiqueta();
    char * eti2=nuevaEtiqueta();
    char * registroId;

    $$=creaLC();

    Operacion aux;

    concatenaLC($$, $5);
    // Asignar valor de $5 a ID
    registroId=recuperaResLC($5);
    aux.op="sw";
    aux.res=registroId;
    aux.arg1=concatena("_", $3);
    aux.arg2=NULL;

    insertaLC($$, finalLC($$), aux);

    //Añadir etiqueta 1
    aux.op="eti";
    aux.res=eti1;
    aux.arg1=NULL;
    aux.arg2=NULL;
}

```



```
insertaLC($$,finalLC($$),aux);

// Salto de la condición
// Concatena código de calculo de la condición
concatenaLC($$, $7);

// Salto
aux.op="bnez";
aux.res=recuperaResLC($7);
aux.arg1=etiq2;
aux.arg2=NULL;

insertaLC($$,finalLC($$),aux);
//Liberar lc
liberaLC($7);
liberarReg(aux.res);

//Añadir statement
concatenaLC($$, $9);

//Liberar codigo
liberaLC($9);

///// Sumar id++
// Sumar 1 al registro reservado
aux.op="addi";
aux.res=registroId;
aux.arg1=registroId;
aux.arg2="1";

insertaLC($$,finalLC($$),aux);
// Guardar en variable de ID
aux.op="sw";
aux.res=registroId;
aux.arg1=concatena("_", $3);
aux.arg2=NULL;

insertaLC($$,finalLC($$),aux);
//Añadir salto a etiqueta 1
aux.op="b";
aux.res=etiq1;
aux.arg1=NULL;
aux.arg2=NULL;

//Insertar codigo
insertaLC($$,finalLC($$),aux);

//Añadir etiqueta 2
aux.op = "etiq";
aux.res = etiq2;
aux.arg1=NULL;
aux.arg2=NULL;
insertaLC($$,finalLC($$),aux); }
```