

Memoria Prácticas Compiladores

Convocatoria Enero

Autores:
<i>Marquina Meseguer, Alfredo - alfredo.m.m@um.es</i>
<i>Martínez Prior, Daniel - daniel.m.p@um.es</i>

ÍNDICE

Análisis Léxico	2
Macros	2
Palabras reservadas y Símbolos	3
Comentarios	3
IDs y Tipos	4
Errores	4
Análisis Sintáctico	5
Precedencia de operadores y conflicto	5
Análisis Semántico	6
Errores semánticos	6
Funciones añadidas a Lista Símbolos	6
Generación de Código	9
Implementación Do-While	12
Ejecución de MiniC	14
Manual del Usuario	14
Pruebas	16
Conclusión	25

Análisis Léxico

El análisis léxico lo podemos separar en las diferentes definiciones de lexemas a encontrar y sus tipos: macros, comentarios, palabras reservadas, id, tipos y símbolos.

El análisis léxico se encarga de devolver tokens para el análisis sintáctico, cada token a sido definido en un fichero generado por Bison llamado miniC.tab.h, aquí se encuentran como una enumeración yytokentype, donde cada uno tiene un número asignado a partir de doscientos cincuenta y ocho, con cuatro tokens creados por Bison.

```
enum yytokentype
{
    YYEMPTY = -2,
    YYEOF = 0,
    YYerror = 256,
    YYUNDEF = 257,
    EMPIEZA = 258,
    LEER = 259,
    ESCRIBIR = 260,
    A_PAREN = 261,
    C_PAREN = 262,
    SEMICOLON = 263,
    COMA = 264,
    ...
    /* "end of file" */
    /* error */
    /* "invalid token" */
    /* EMPIEZA */
    /* LEER */
    /* ESCRIBIR */
    /* A_PAREN */
    /* C_PAREN */
    /* SEMICOLON */
    /* COMA */
}
```

Macros

Las macros utilizadas son bastante simples y sus nombres resultan autoexplicativos, seleccionando en la mayor parte de casos simplemente la selección de los valores ASCII que corresponden a qué secciones. Por ejemplo, la macro dígito es el rango de los caracteres entre cero y nueve, y pánico es cualquier carácter que no se encuentra incluido en la gramática.

```
digito      [0-9]
letra       [a-zA-Z]
entero      {digito}+
panico      [^A-Za-z0-9();,=+-{}_~$]
```

Palabras reservadas y Símbolos

Las palabras reservadas también constituyen otra parte del léxico que resulta muy sencillo. Se trata de una serie de términos que deben tener un comportamiento propio, por esta razón las variables creadas más adelante no deberían poder compartir nombre con ninguna de estas palabras.

De la misma manera se obtienen los símbolos (multiplicación, suma, apertura de paréntesis, etc.) que cumplen con las mismas características de las palabras reservadas.

```
// Ejemplo de palabra reservada y de símbolo
void                                     return EMPIEZA;
"+"                                    return MAS_OP;
```

Comentarios

La identificación de líneas de comentario se ha realizado comprobando mediante una expresión regular. Esta comprueba que existen dos caracteres '/' seguidos y de hacerlos considera que el resto de la línea es un comentario.

```
/* Comentario: 1 línea */
"//"(.*)[\n]                ;
```

Para el comentario multilínea ha resultado un poco más difícil ya que requiere el uso de condiciones de contexto para poder detectarlos. Cuando se identifica los caracteres '/' y '*' juntos, se inicia la etiqueta COMENTARIO. A partir de este momento solo se comprobarán reglas que tengan incluida la etiqueta COMENTARIO hasta que se vuelva a iniciar el inicio que es de valor cero, esto llegará al encontrarse los mismo caracteres juntos pero en orden contrario. Con esto conseguimos que cualquier texto que se encuentre entre estos dos marcadores no sea compilado.

```
%x COMENTARIO
...

"/*"                               {yymore(); BEGIN COMENTARIO;}
<COMENTARIO>"*/"                  BEGIN 0;
<COMENTARIO>.                      ;
<COMENTARIO>\n                     ;
```

IDs y Tipos

Los tipos y los ids son elementos que se han conseguido de manera muy similar, ya que además de los tipos se debe identificar el lexema que este contiene, por lo que hay que devolverlo. Hemos pasado el valor del lexema o nombre de la ID a través de la variable `yyval` normalmente utilizada por Bison.

`yytext` es la variable que contiene la cadena de texto reconocida, por lo que al utilizar la función `strdup` con ella como parámetro estamos creando una copia de su contenido. Esta copia es asignada `yyval.lexema` para uso posterior del análisis sintáctico.

Al guardar el valor del id o lexema en la parte de lexema del struct `yyval`, cuando devolvemos uno de los tokens que debería devolver una cadena, el programa comprueba su valor en este sitio en específico.

```
(_|\\$|{letra})({letra}|{digito}|_|\\$){0,15}    {yyval.lexema = strdup(yytext);return ID;}
entero                                           {yyval.lexema = strdup(yytext);return NUM;}
"\"([^\n\\n]|\\.)*\"                               {yyval.lexema = strdup(yytext);return STRING;}
```

Errores

Al igual que los comentarios, se ha necesitado del uso de etiquetas para implementar el modo pánico a este programa.

Primero se ha identificado el inicio de un error aceptando cualquier carácter que no se haya metido en ninguna de las reglas establecidas, es decir, la regla de error se encuentra al final de la lista de reglas. Una vez empezado el error léxico se continúan considerando todos los caracteres que no se entran dentro de los aceptados, como se ha mencionado antes, estos caracteres se encuentran dentro de la macro pánico. Finalmente, cuando se encuentra un carácter que sí está dentro de los aceptados se termina el error y se informa al sintáctico que ha ocurrido para que no compile.

```
<ERROR>{panico}  {yymore();}
<ERROR>.        {yyless(yylen-1);printf("Reconocido error en
                linea %d: %s.\n",yylineno, yytext); BEGIN 0;
                return ERR;}
.               {yymore(); BEGIN ERROR;}
```

snappify.com

Análisis Sintáctico

El propósito de esta parte del compilador es 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.

Esta parte del programa ha sido en su mayor parte proporcionada por el enunciado así que los cambios requeridos para que sea funcional han sido pocos.

Precedencia de operadores y conflicto

Se evita la ambigüedad en el program a partir de expresar la relaciones de precedencia entre los operadores "+", "-", "*" y "/". Se quería especificar que los dos primeros operadores tienen menos preferencia que los dos segundos, con los miembros de los grupos compartiendo precedencia entre ellos. Para ello, se ha otorgado una asociatividad izquierda superior a "+" y "-" sobre "*" y "/". Sin embargo, seguía existiendo cierta ambigüedad pues el símbolo de resta y que indica la negatividad de un número son el mismo, por lo que se ha tenido que especificar precedencia ya no entre símbolos sino entre tokens del UMENOS sobre el resto, incluyendo OP_MENOS, para que se compruebe este sobre el resto.

Finalmente se ha especificado a Bison el número de conflictos desplazamiento-reducción se puede encontrar en el fichero.

```
%left MAS_OP MENOS_OP
```

```
%left POR_OP DIV_OP
```

```
%precedence UMENOS
```

```
OP_MENOS expression %prec UMENOS
```

```
%expect 1 /* conflictos desplazamiento/reduccion*/
```

Análisis Semántico

El análisis semántico tiene como objetivo almacenar los diferentes tipos de datos en una tabla de símbolos y detectar errores semánticos. Para ello, utilizaremos una lista simple como estructura de datos. En las reglas de producción que involucren declaraciones, asignaciones, lecturas o impresiones, se realizará el procesamiento semántico. Se insertará cada variable, constante o cadena al final de la lista de símbolos, almacenando su nombre y tipo. Para las cadenas, se utilizará un campo adicional llamado "valor" para mantener el número de cadena en el código.

Errores semánticos

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

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

Funciones añadidas a Lista Símbolos

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

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;
}
```

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);
}
```

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.

```
listaSimbolos.c
```

```
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);
    }
}
```

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.

```
listaSimbolos.c
```

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

    // Recorremos la lista hasta llegar al final
    while (pos != finalLS(lista)) {
        // Obtenemos el simbolo de la posición actual
        simbolo = recuperaLS(lista, pos);
        /* 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);
        }
        // Avanzamos a la siguiente posición
        pos = siguienteLS(lista, pos);
    }
}
```



```
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")
}
```

Generación de Código

Finalmente terminamos generando el programa sin la sección de datos pues ha sido lo último que se ha realizado durante el análisis semántico. Si este análisis ha transcurrido sin encontrar ningún error, se procederá a generar el código.

Para la representación de este código, se ha utilizado una lista simple. Cada uno de los nodos contendrá una operación a realizar en MIPS, cada nodo cuenta con cuatro campos, uno para la operación y tres más para los registros utilizados en la operación, tres es el número máximo de registros a utilizar y ciertas operaciones pueden simplemente utilizar uno de ellos. A su vez, cada operación realizada en el lenguaje miniC se traducirá a una o más operaciones de MIPS por lo que serán uno o varios nodos de esta lista.

Para empezar tenemos que importar la lista de código en el fichero miniC.y. La generación de código se realiza en este fichero por lo que la ListaC es instanciada aquí.

```
miniC.y

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

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

%union{
char *lexema;
ListaC codigo;
}

%type <codigo> expression statement /*...*/
```

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

```
listaCodigo.c

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.

```
listaCodigo.c
```

```
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 numEtiq nos aseguramos la unicidad de cada etiqueta.

```
miniC.y
```

```
char *nuevaEtiqueta() {
    char aux[16];
    sprintf(aux, "%l%d", numEtiq);
    numEtiq++;
    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 "eti" o no.

```
listaCodigo.c
```

```
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  $\neq$  finalLC(listaCodigo)) {
    oper = recuperaLC(listaCodigo, pos);
    if (strcmp(oper.op,"eti") {
        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.

miniC.y

```

program : {lista=creaLS(); iniReg();} EMPIEZA ID A_PAREN C_PAREN A_LLAVE declarations stati
if (numErroresLex + numErroresSint + numErroresSem == 0) { // Generamos el código
    imprimirTablaS(lista);
    concatenaLC($7, $8);
    liberaLC($8);
    imprimirListaC($7);
}
else {
    printf("-----\n");
    printf("Errores lexicos: %d\n",numErroresLex);
    printf("Errores sintacticos: %d\n",numErroresSint);
    printf("Errores semanticos: %d\n",numErroresSem);
}
/* Liberar memoria de tabla de símbolos y lista de código */
liberaLS(lista);
liberaLC($7); }

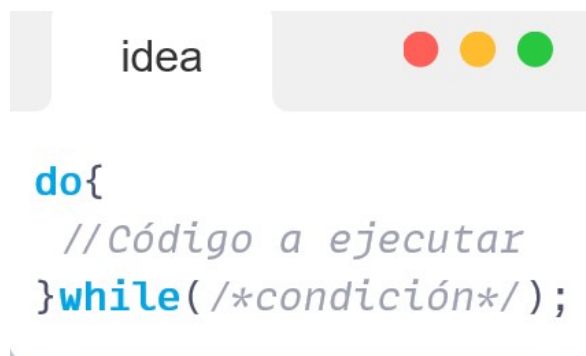
```

Implementación Do-While

La estructura de un código `do while` (imagen inferior izquierda) consta de la palabra clave “do” para señalar el inicio del bucle, entre esta palabra clave y `while` se especifica las instrucciones del bucle y finalmente se pone la expresión lógica. Primero debemos añadir la palabra clave “do” al nuevo vocabulario, para la que se ha utilizado la macro HACER y se ha añadido a Bison, así se reconoce durante el análisis sintáctico. Además, el reconocimiento de la estructura se realiza en el análisis semántico y el bucle se encuentra dentro de la parte del cuerpo del código, por lo que se tiene que añadir como una regla de `statement`. De esta manera la estructura en Flex sería reconocida por la expresión:

HACER statement MIENTRAS A_PAREN expression C_PAREN SEMICOLON

Siguiendo la misma estructura, podemos pensar que, para ensamblador podemos seguir el mismo formato, de hecho, sería hasta más fácil de implementar que el bucle `while`. Simplemente insertamos una etiqueta para marcar el inicio del bucle, ejecutamos el `statement`, evaluamos la expresión lógicas y realizamos el salto. En este bucle, al contrario que el `while`, la operación de salto debe ser *benq*, *branch not equals zero*, ya que el salto implica que el bucle tiene otra iteración y no lo contrario como en este anterior.



```
do{  
    //Código a ejecutar  
}while( /*condición*/ );
```

Estructura do-while en C

etiqueta:

```
# Código statement  
  
# Código expression  
  
bneq ... # Salto
```

estructura do-while en MIPS

A la hora de implementar la generación de código podemos reciclar gran parte del utilizado para el bucle `while`. Mirando el código podemos comprobar la sencillez del código pues se resume en insertar etiqueta, concatenar resultados de *statement* y de *expression* e insertar salto a etiqueta.

```
| HACER statement MIENTRAS A_PAREN expression C_PAREN SEMICOLON {
    $$=creaLC();
    char * etiq = nuevaEtiqueta();

    Operacion aux;

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

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

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

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

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

Ejecución de MiniC

A partir del fichero `makefile` y el código fuente se puede obtener el compilador de miniC. La forma sencilla de ejecutarlo es con el comando *make*, que ejecuta la primera instrucción que te devuelve una ejecutable llamado *miniC* que te permite compilar programas de miniC a MIPS si se pasa el fichero fuente como primer parámetro.

makefile

```
miniC: lex.yy.c miniC.tab.c main.c listaSimbolos.c listaCodigo.c
    gcc -g lex.yy.c miniC.tab.c main.c listaSimbolos.c listaCodigo.c -lfl -o miniC
miniC.tab.h miniC.tab.c: miniC.y
    bison -d -v miniC.y
lex.yy.c: miniC.l miniC.tab.h
    flex miniC.l
limpia:
    rm -f lex.yy.c miniC.tab.* miniC.output miniC
```

Se ha de tener en cuenta que el programa vuelca toda la salida por la salida estándar, por lo que si se desea guardar en un fichero se debería redireccionar la salida. Teóricamente el código generado por miniC, si el programa es correcto, puede ser copiado y ejecutado en un editor como Mars. Sin embargo, no se podría ejecutar directamente por una máquina que utilice este lenguaje ensamblador por los diversos mensajes de aviso lanzados por los compiladores.

Manual del Usuario

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*.

```

root@DESKTOP-04FAV0S:/mnt/UltraMegaMiniC# make
bison -d -v miniC.y
flex miniC.l
gcc -g lex.yy.c miniC.tab.c main.c listaSimbolos.c listaCodigo.c -lfl -o miniC
root@DESKTOP-04FAV0S:/mnt/UltraMegaMiniC# ls -l miniC
-rwxr-xr-x 1 root root 91176 Dec 12 22:58 miniC
root@DESKTOP-04FAV0S:/mnt/UltraMegaMiniC# ./miniC pruebas/ps.mc > pruebas/ps.s
root@DESKTOP-04FAV0S:/mnt/UltraMegaMiniC# spim -file pruebas/ps.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
Comienza simulación \ junio 2023
Introduce valores de 'y' y 'z' 3 veces
1 2 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

```

Aquí tenemos un ejemplo de compilación para la prueba ps.mc

Pruebas

Vamos a ver unas pruebas variadas en las que comprobamos el buen funcionamiento del compilador. **Primero veamos la prueba facilitada por el profesorado:**

prueba_estandar.mc

```
void prueba() {
const a=0, b=0;
var c=5+2-2;
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";
}
```

Para esa entrada de archivo tipo miniC tenemos el siguiente .s:

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

$str1:
.asciiz "Inicio del programa\n"
$str2:
.asciiz "a"
$str3:
.asciiz "\n"
$str4:
.asciiz "No a y b\n"
$str5:
.asciiz "c = "
$str6:
.asciiz "\n"
$str7:
.asciiz "Final"
$str8:
.asciiz "\n"
_a:
.word 0
_b:
.word 0
_c:
.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 \$t0, \$t0, \$t1

li \$t1, 2

sub \$t0, \$t0, \$t1

sw \$t0, _c

li \$v0, 4

la \$a0, \$str1

syscall

lw \$t0, _a

beqz \$t0, \$I5

li \$v0, 4

la \$a0, \$str2

syscall

li \$v0, 4

la \$a0, \$str3

syscall

b \$I6

\$I5:

lw \$t1, _b

beqz \$t1, \$I3

li \$v0, 4

la \$a0, \$str4

syscall

b \$I4

\$I3:

\$I1:

lw \$t2, _c

beqz \$t2, \$I2

li \$v0, 4

la \$a0, \$str5

syscall

lw \$t3, _c

li \$v0, 1

move \$a0, \$t3

syscall

li \$v0, 4

la \$a0, \$str6

```

syscall
lw $t3, _c
li $t4, 2
sub $t3, $t3, $t4
li $t4, 1
add $t3, $t3, $t4
sw $t3, _c
b $l1
$I2:
$I4:
$I6:
li $v0, 4
la $a0, $str7
syscall
li $v0, 4
la $a0, $str8
syscall

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

```

Y la salida con spim -file es la siguiente:

```

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

```

Tenemos ahora otra prueba algo más larga que trata los comentarios de texto, al igual que otros ejemplos similares a la prueba anterior:
prueba_original.mc:

```

void 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";
}
```

El archivo .s generado es el siguiente:

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

$str1:
    .ascii "hola \"esto es\" una prueba"
$str2:
    .ascii "\n"
$str3:
    .ascii "b es true porque su valor es distinto de 0\n"
$str4:
    .ascii "El valor de b va disminuyendo en"
$str5:
    .ascii "cada iteración\n"
$str6:
    .ascii "hola"
$str7:
    .ascii "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 $t1, $t0
sw $t0, _y
li $t0, 3
sw $t0, _b
lw $t0, _x
lw $t2, _y
add $t0, $t0, $t2
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 $t2, _b
    li $t3, 1
    sub $t2, $t2, $t3
    sw $t2, _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
```

Y mediante spim -file vemos que el resultado del programa es:

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

Como última prueba que debe dar satisfactoria tenemos un do-while para comprobar la implementación extra añadida en nuestro compilador:

prueba_do_while.mc:

```
void 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; }
}
```

Que genera el siguiente código .s:

```
#####
```

```
# Seccion de datos
```

```
.data
```

```
$str1:
```

```
.asciiz "Primer do while "
```

```
$str2:
```

```
.asciiz "\n"
```

```
$str3:
```

```
.asciiz "El valor de a es "
```

```
$str4:
```

```
.asciiz "\n"
```

```

$str5:
    .ascii "\nSegundo con while "
$str6:
    .ascii "\n"
$str7:
    .ascii "El valor de a es "
$str8:
    .ascii "\n"
_a:
    .word 0

```

```
#####
```

```
# Seccion de codigo
```

```

    .text
    .globl main
main:
    li $t0, 3
    sw $t0, _a
    li $v0, 4
    la $a0, $str1
    syscall
    lw $t0, _a
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str2
    syscall
    lw $t0, _a

```

```

$I1:
    li $v0, 4
    la $a0, $str3
    syscall
    lw $t0, _a
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 4
    la $a0, $str4
    syscall
    lw $t0, _a
    li $t1, 1
    sub $t0, $t0, $t1
    sw $t0, _a
    lw $t0, _a
    bnez $t0, $I1
    li $t0, 5
    sw $t0, _a

```

```

        li $v0, 4
        la $a0, $str5
        syscall
        lw $t0, _a
        li $v0, 1
        move $a0, $t0
        syscall
        li $v0, 4
        la $a0, $str6
        syscall
$I2:
        lw $t0, _a
        beqz $t0, $I3
        li $v0, 4
        la $a0, $str7
        syscall
        lw $t1, _a
        li $v0, 1
        move $a0, $t1
        syscall
        li $v0, 4
        la $a0, $str8
        syscall
        lw $t1, _a
        li $t2, 1
        sub $t1, $t1, $t2
        sw $t1, _a
        b $I2
$I3:

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

```

Con su correspondiente programa:

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

Por último, tenemos una prueba para comprobar que los errores son analizados correctamente (en este caso, variables no declaradas):

prueba_var_no_declarada.mc:

```
void test3() {  
    var a;  
    const b = 3;  
  
    x = 1;  
    a = 2*y;  
    read z;  
}
```

Que genera lo siguiente:

Variable x no declarada
Variable y no declarada
Variable z no declarada

Errores lexicos: 0
Errores sintacticos: 0
Errores semanticos: 3

Como vemos, las 3 variables no declaradas se analizan correctamente como errores semánticos.

En la carpeta del proyecto hay una carpeta con más pruebas que hemos estado realizando.

Conclusión

En resumen, se ha logrado completar el desarrollo de un compilador capaz de traducir un lenguaje simplificado de C, miniC, al lenguaje ensamblador MIPS. Este proyecto ha supuesto un desafío interesante y nos ha permitido adquirir experiencia en el campo de la compilación y la traducción de lenguajes de programación.

El compilador implementado ha conseguido generar código MIPS válido. Se ha puesto un enfoque especial en el análisis semántico.

Aunque el compilador presenta algunas limitaciones, en su mayoría ha sido capaz de funcionar de manera adecuada en la traducción de programas de miniC a MIPS.

En definitiva, este trabajo ha sido una valiosa oportunidad para sumergirnos en el mundo de la compilación y la traducción de lenguajes de programación, adquiriendo habilidades y experiencia que nos servirán en futuros proyectos.