
MEMORIA DE PRÁCTICAS

COMPILADORES

Curso 2020/21
Convocatoria de Junio

Javier Barceló Pérez
javier.barcelop@um.es

Gema Madrid Sánchez
gema.madrids@um.es

Profesora: María Antonia Cárdenas Viedma

ÍNDICE

Introducción	3
Análisis léxico	3
Tokens	3
Comprobaciones	4
Análisis sintáctico	4
Análisis semántico	5
Generación de código	6
Estructuras de datos:	7
Funciones principales:	7
Manual de usuario:	12
Ejemplo de uso:	12
Conclusiones	17

Introducción

Esta práctica consiste en el desarrollo de un compilador para el lenguaje MiniC, una versión simplificada de C utilizada principalmente para pequeños computadores con recursos limitados y sistemas embebidos. El lenguaje empleado para implementar el compilador será C.

Para ello haremos uso de los conocimientos obtenidos en teoría e implementaremos un analizador léxico, un analizador sintáctico y un analizador semántico. Estos tres elementos irán generando los elementos necesarios, apoyándose unos en otros, para finalmente generar el código ensamblador (MIPS) ejecutable del programa original. También se avisará de los principales errores que pueda haber en el código original. En esta práctica se ha hecho uso de la gramática presentada en el enunciado para desarrollar el archivo .y, y además se ha añadido la sentencia extra do-while a la implementación del compilador.

Análisis léxico

La tarea del analizador léxico es transformar el flujo de caracteres del código de entrada en una serie de tokens que posteriormente utilizarán los demás componentes del compilador. Para la creación del analizador léxico se ha empleado la herramienta *Flex*. Se encuentra implementado en el fichero "minic.l", donde se identifican los tokens correspondientes a los distintos tipos de símbolos terminales de la gramática y se definen las expresiones necesarias para realizar el análisis léxico de un código.

Tokens

- Los números enteros se representan con el token NUM.
- Las cadenas, con el token CADENA.
- Los identificadores, con el token ID.
- Las palabras reservadas se identifican con los siguientes tokens (entre paréntesis): *void* (VOID), *var* (VAR), *const* (CONST), *if* (IF), *else* (ELSE), *do* (DO), *while* (WHILE), *print* (PRINT), *read* (READ).
- Los caracteres especiales (entre comillas) se identifican con los siguientes tokens (entre paréntesis): ";" (SEMICOLON), "," (COMA), "+" (MAS), "-" (MENOS), "*" (POR), "/" (DIV), "=" (IGUAL), "(" (APAR), ")" (CPAR), "{" (ACOR), "}" (CCOR).

Se reconocen además comentarios simples (que comienzan por //) y multilínea (con formato /*comentario*/), pero se omiten, puesto que no forman parte del código y no son necesarios en etapas posteriores.

Comprobaciones

El analizador léxico también realiza comprobaciones para detectar errores léxicos y notificarlos al usuario.

- Detección de comentarios multilínea no cerrados, notificando al usuario de la línea donde comienza el comentario.
- Detección de cadenas no cerradas, notificando la línea de comienzo y la cadena que no está cerrada.
- Comprobar que un identificador no tenga una longitud mayor de 16 caracteres.
- Comprobar que un número tenga un valor entre -2^{31} y 2^{31} .
- Recuperación de errores en modo pánico: si se encuentra un carácter o secuencia de caracteres erróneos, que no pertenecen a la gramática, se notifica al usuario de la línea y la secuencia y se saltan caracteres hasta encontrar uno válido.

Para la detección de errores se han implementado al final del fichero “minic.l” varias funciones en C: *error_lexico* (cuando encontramos una secuencia de caracteres inválidos), *error_comentario*, *error_cadena*, *comprobacion_id* y *comprobacion_numero*. Estas funciones imprimen el error que se ha producido y la línea del fichero de entrada se ha producido. En el caso de las comprobaciones se comprueba si el lexema no cumple las restricciones y en caso afirmativo se imprime el error.

Análisis sintáctico

El analizador sintáctico implementa la gramática del lenguaje, cuya sintaxis está definida con notación BNF en el punto 2.1.2. del enunciado de la práctica, y comprueba si la secuencia de tokens que le proporciona el analizador léxico puede ser generada por la gramática.

Para implementar la gramática se han dejado los mismos nombres para los símbolos no terminales, y los símbolos terminales se han traducido por los tokens que hemos definido en el analizador léxico, utilizando la sintaxis de *Bison*. La implementación se encuentra en el fichero “minic.y”.

En el fichero “minic.l” se han incluido reglas en los tokens CADENA, ID y NUM para devolver atributos al analizador sintáctico. En concreto, se ha añadido la acción

`yyval.cadena=strdup(yytext)` para devolver el valor asociado a los atributos, siendo `cadena` un tipo definido dentro de una estructura “union” y asignado a los tokens CADENA, ID y NUM en “`minic.y`”.

Para las operaciones aritméticas se definen reglas de precedencia para eliminar ambigüedades con la siguiente sentencia:

```
%left MAS MENOS
%left POR DIV
%left UMENOS
```

Las líneas que aparecen antes tienen menos precedencia que las siguientes, siendo la negación la de mayor precedencia, seguida de la multiplicación y división, y finalmente la suma y resta.

Hay un conflicto desplazamiento/reducción que surge de la ambigüedad en las sentencias *if* e *if-else*. Esto ocurre una vez hemos leído *statement* y tenemos un ELSE a continuación, y hay que decidir si reducir o desplazar. En este caso dejamos que *Bison* realice la acción por defecto, que es desplazar, puesto que el ELSE forma parte del IF que estamos procesando actualmente, y por tanto se deberá reducir cuando hayamos procesado la expresión completa.

Análisis semántico

El analizador semántico se encarga de gestionar la declaración y uso de las variables y constantes. Para ello se ha definido en el fichero “`minic.y`” una variable global *tablaSimb* de tipo “Lista”, definido en los ficheros que se han proporcionado “`ListaSimbolos.c`” y “`ListaSimbolos.h`”, que contiene las constantes, variables y cadenas declaradas.

Se han definido varias funciones para añadir elementos a la tabla:

- *perteneceTablaS*: recibe como entrada un lexema y devuelve un valor entero (booleano) para indicar si ya está en la tabla.
- *anadeEntrada*: recibe un lexema y su tipo (VARIABLE, CONSTANTE o STRING) y lo añade a la tabla.
- *esConstante*: comprueba si el lexema que recibe de entrada es de tipo CONSTANTE.
- *imprimirTabla*: imprime los valores almacenados en la tabla, que se corresponden con la cabecera o sección de datos del programa que se generará más tarde en código ensamblador.

La tabla de símbolos se crea inicialmente en la regla *program*, justo antes de comenzar a leer los lexemas del programa.

Las variables y constantes se añaden a la tabla cuando se declaran. Se ha creado una variable global *tipo*, cuyo valor (VARIABLE o CONSTANTE) se establece en la regla *declarations*, antes de comenzar a leer la lista de identificadores. En la regla *asig* se comprueba si un identificador ya está en la tabla, en cuyo caso se notifica al usuario del error (variable ya declarada), y en caso contrario se añade a la tabla.

En la primera regla de *statement* se comprueba que la variable a la que se quiere asignar un valor esté declarada y no sea una constante.

En la segunda regla *print_item* se añade la cadena a la tabla de símbolos si no se encuentra ya en ella.

En *read_list* también se comprueba que la variable en la que se quiere almacenar el valor leído esté declarada en la tabla y no sea una constante.

Generación de código

Finalmente, para la generación de código se ha empleado el tipo *ListaC* proporcionado en los ficheros “listaCodigo.c” y “listaCodigo.h”, donde se irán almacenando la traducción del código de entrada en MiniC a instrucciones en ensamblador.

A la estructura union definida anteriormente se le ha añadido un nuevo tipo *codigo* (tipo *ListaC*), y se ha asignado a todos los símbolos terminales, puesto que iremos creando y asignando listas de código ensamblador a los símbolos no terminales según su función. En la regla *expression* asignaremos listas con instrucciones aritméticas, etc. Conforme se vayan haciendo reducciones, estas listas se irán concatenando hasta obtener dos listas correspondientes a *declarations* (lista con la declaración de las variables) y *statement_list* (sentencias y expresiones aritméticas) en la regla *program*.

Una vez obtenidas las listas finales, las imprimimos a continuación de la tabla de símbolos si no ha ocurrido ningún error léxico, sintáctico o semántico. Si han ocurrido errores se mostrarán los errores y la cantidad de errores léxicos, sintácticos y semánticos.

A continuación se detallan las estructuras de datos y las funciones que se han creado para realizar la traducción del código.

Estructuras de datos:

Se han utilizado diversas estructuras para la traducción del código MiniC a ensamblador MIPS. Primero, se ha usado la ya mencionada tabla de símbolos como una variable global llamada `tablaSimb` donde se han ido registrando todos los símbolos del código haciendo uso de las funciones mencionadas en el apartado del análisis semántico.

Para la gestión de los registros en código ensamblador, el enunciado de la práctica indicaba que los registros a usar eran `$t0-9`, por lo que se ha creado un array de registros llamado `registros[10]`, consistente en una especie de array de booleanos en el que cada posición representa el estado de un registro (si está a 0 está disponible, si está a 1 no se puede utilizar). Con las funciones `buscarReg()` y `liberarReg()` que hemos diseñado, se manejan los registros desde el código de las diferentes traducciones.

Las estructuras que hemos utilizado para la generación del código, como ya se ha mencionado, han sido las `ListasC` que representan una lista enlazada de unas estructuras llamadas Operaciones. Estas operaciones son estructuras consistentes en cuatro cadenas de caracteres, que representan la operación a imprimir en la traducción con sus cuatro campos (nombre de operación, registro resultado, argumento 1 y argumento 2). Además, las `ListaC` tienen un campo `n`, que es un entero que indica la longitud de la lista, y un campo `res` que consiste en una cadena con el registro resultado de todo el bloque de operaciones de la lista. El tratamiento de estas listas se ha llevado a cabo íntegramente con las operaciones que se facilitaban desde la librería `listaCodigo`.

Funciones principales:

En el archivo de Bison `minic.y`, hacemos uso de diversas funciones para el tratamiento de las listas de código y listas de símbolos. Estas funciones son llamadas desde las diferentes reglas de la gramática implementada, y son las siguientes:

- `char* buscarReg()`: Esta función es utilizada para la gestión de los registros por el programa de ensamblador. Hace uso de la estructura `registros` mencionada previamente, que almacena la información de qué registros están en uso y cuáles están libres. Devuelve la cadena con el registro `$tx`, siendo `x` el número del primer registro libre. En caso de que no haya ningún registro libre, la función aborta la

ejecución del programa. Prácticamente todas las funciones hacen uso de esta función para la asignación de registros a los campos de las operaciones.

- *void liberarReg(char* registro)*: La función recibe como parámetro una cadena con el registro a liberar, y obtiene de la cadena el número del registro en el array global de *registros*. Tras esto, pone esa posición del array a 0, indicando así que está libre para las siguientes funciones. Al igual que la función anterior, muchas funciones del programa hacen uso de esta para liberar registros cuando sea necesario.
- *char* obtenerEtiqueta()*: Esta función es utilizada en las funciones que gestionan las instrucciones de salto. La función crea una cadena de caracteres formada por la secuencia "\$1X", siendo X un número diferente cada vez que se llame a la función gracias al uso de una variable global *tag_counter* que vaya contando el número de etiquetas que se generan. Se devuelve dicha cadena.
- *ListaC crearLista(char* arg1, char* op)*: Esta función se utiliza para generar una instrucción *li* o *lw* que será asignada a una expresión en la gramática. Obtiene como entrada la cadena correspondiente, y si es un token NUM se añade una operación *li* a la nueva lista, y si es un token ID se añade una operación *lw*. Devuelve una nueva estructura del tipo *ListaC* que será asignada a la expresión desde la que se llama a la función.
- *ListaC crearLista2(ListaC lista, ListaC arg2, char* op)*: Esta función se utiliza para generar una instrucción de cualquiera de las 4 operaciones aritméticas posibles en MiniC. Toma como parámetros las listas asignadas a las dos expresiones operandos. Primeramente concatena la segunda lista a la primera, y libera acto seguido la segunda lista. Tras eso se añade la operación al final de la lista. Según la operación aritmética, añade una operación *add*, *sub*, *mul* o *div* según corresponda. Los registros resultado de las listas operando se liberan, y se guarda como nuevo registro resultado de la lista el registro resultado de la operación añadida. Devuelve una nueva estructura del tipo *ListaC* que será asignada a la expresión desde la que se llama a la función.
- *ListaC crearLista3(ListaC lista, char* var, char* op)*: La función es utilizada en las declaraciones de constantes y variables con valor inicial, y asignaciones en sentencias. Toma como parámetros la lista de la expresión con el valor a asignar, y la variable en la que se guardará el valor, además de la operación "sw" como

cadena. La función añade a la variable la barra baja “_” previa a la variable para ajustarlo a cómo se guardan las variables en MIPS, y después añade la operación `sw` a la lista, libera el registro resultado de la lista y asigna la lista resultante a la sentencia o la asignación que la haya llamado.

- *ListaC crearListaNeg(ListaC lista, char* op)*: Función similar a la anterior, pero específica para la adición de una instrucción `neg` que invierta el signo del valor del registro resultado de la lista pasada como parámetro. Añade la instrucción al final de la lista, libera el registro resultado de la lista y lo sustituye por el nuevo registro de la negación, y asigna la lista a la expresión que llama a la función.
- *ListaC listalf(ListaC cond, ListaC st)*: En esta función se reciben como parámetros la lista de código correspondiente a la condición del salto, y la correspondiente a las sentencias a ejecutar. Primeramente, se añade una instrucción `beqz` al final de la lista correspondiente a la condición, para que se ejecute o no el salto. Se libera el registro resultado de la lista de la condición, se concatena la segunda lista a la primera, se guarda el valor resultado de la segunda lista en la primera y se libera la segunda lista. Por último, la etiqueta utilizada en la operación `beqz` anteriormente mencionada se le añaden dos puntos “:” al final, y añade como operación al final de la lista para poder saltar a ésta en caso de que el salto se ejecute. Se asigna la lista resultante al statement desde el que se ha llamado a la función.
- *ListaC if else(ListaC exp, ListaC stat1, ListaC stat2)*: Esta función es algo más compleja ya que debe tratar varios saltos y listas simultáneamente. Recibe como parámetros tres listas, la primera con la lista correspondiente a las instrucciones de la expresión condición, la segunda corresponde a las sentencias que se ejecutarán en caso de que la condición sea true, y la tercera lista corresponde a las sentencias a ejecutar cuando no se cumpla la condición (al else). Primeramente, se inserta una operación `beqz` con una etiqueta a la que llamaremos `et1`, y con el valor resultado de la lista `exp` como condición del salto (registro que se liberará al final de la ejecución de la función). A continuación se concatena la lista `stat1` a la lista `exp` (y se libera `stat1`). Tras eso, creamos la operación de salto incondicional `b` a una nueva etiqueta que llamaremos `et2`, y añadimos la operación al final de la lista resultante. Justo después de esto, añadimos la etiqueta `et1` como operación, de manera que cuando se realice el salto condicional se salte directamente a esta parte del código (la parte del else). Después, se concatena la lista `stat2` a `exp` y se libera `stat2`. Por último, se

añade la etiqueta et2 como operación, para que cuando no haya que ejecutar la parte del else, se salte directamente a esta parte del código desde el salto incondicional previamente mencionado. La lista exp resultante se asigna a la sentencia desde la que ha sido llamada.

- *ListaC while (ListaC exp, ListaC stat)*: La función while_ recibe como parámetros dos listas, la lista correspondiente a las instrucciones de la condición del salto y la lista de las instrucciones de las sentencias a ejecutar. Lo primero es generar e insertar una etiqueta et1 como operación al inicio de la lista exp, a la que se saltará al final del while. Después se añade al final de la lista exp el salto condicional beqz a una segunda etiqueta et2 que se añadirá al final de la ejecución de la función, para salir del bucle. El registro resultado de la lista exp se añade a la instrucción como argumento, y se libera. Tras esto, concatenamos la lista stat a la lista exp, y liberamos stat. A continuación se añade al final de la lista un salto incondicional b que salta a la primera etiqueta et1 que hemos añadido. Por último, se añade la etiqueta et2 como operación al final de la lista de código, para salir del bucle cuando se llegue desde el salto condicional. La lista exp resultante se asigna a la sentencia desde la que ha sido llamada.
- *ListaC do_while(ListaC stat, ListaC exp)*: Esta función gestiona las instrucciones, saltos y etiquetas correspondientes a la sentencia do-while, y recibe como parámetros la lista de sentencias a ejecutar y la lista de la expresión de la condición. Primero inserta una etiqueta al inicio de la lista de las sentencias, y después concatena la lista exp a la primera lista, liberando tras esto la lista exp. Al final de esta lista resultante se inserta una instrucción bnez que consulta el registro resultado que tenía la lista exp para ejecutar o no el salto a la etiqueta previamente generada. Inmediatamente después, este registro se libera. La lista se asigna a la sentencia desde la que se ha llamado a la función.
- *ListaC listaPrintItem(char* cadena)*: La función se encarga de crear una lista de código que contenga las instrucciones para imprimir una cadena de caracteres que se pasa como parámetro. Se crea una lista vacía, y se recupera el identificador de la cadena parámetro desde la lista de símbolos. Entonces el valor que se usará para la cadena será \$strX siendo X el identificador de la cadena recuperado. Después se añaden a la lista las instrucciones la \$a0, \$strX -> li \$v0, 4 -> syscall que

se emplean en ensamblador para imprimir por pantalla la cadena. Por último se asigna la lista al `print_item` que llama a la función.

- *ListaC listaPrintExpresion(ListaC arg)*: Funciona de forma similar a la anterior función, pero al no tener que imprimir una cadena sino el valor resultado de una expresión, se empieza recuperando el valor resultado de la lista pasada como parámetro. Dicho registro se añade como argumento a la instrucción `move $a0, $tX`, y junto a las instrucciones `li $v0, 1` y `syscall` se añaden al final de la lista en ese orden. El registro resultado de la lista pasada como parámetro de la expresión, `$tX`, se libera. Por último se asigna la lista al `print_item` que llama a la función.
- *ListaC listaRead(char* cadena)*: La función *listaRead* recibe como parámetro un identificador con el nombre de la variable en la que se guardará el resultado de la lectura. Primero se crea una nueva lista en la que se añaden las instrucciones de ensamblador para la lectura por entrada estándar de datos, que son `li $v0, 5` y `syscall`. Después, el resultado de esa lectura que estará guardado en el registro `$v0` se almacena en la variable pasada como parámetro mediante una operación `sw` que se añade al final de la lista, por supuesto añadiendo la barra baja “_” antes de la variable. La lista resultante se asigna al `read_list` que llama a la función.
- *ListaC concatena(ListaC l1, ListaC l2)*: Esta función es una función que simplemente concatena las dos listas que se pasan como parámetro y libera la memoria de la segunda lista, retornando solo una lista con las dos juntas. Es una función auxiliar para facilitar la legibilidad del código en la parte de la gramática. Se usa en los `statement_list`, `print_list`, `read_list` y las reducciones en las declaraciones y las listas de identificadores al inicio de la ejecución del programa.
- *void debugLista(ListaC lista)*: Es importante aclarar que, aunque esta función no afecta a la ejecución del programa en ningún caso, ha sido de gran ayuda a la hora de corregir errores. Simplemente imprime por pantalla la información y las operaciones de la lista pasada como parámetro.
- *void imprimirTablaS()*: Primero se imprime la directiva `.data` para indicar el comienzo de las declaraciones de cadenas, constantes y variables. La función recorre dos veces la tabla de símbolos, para imprimir por pantalla primero las cadenas con el formato “`$strX:\t.asciiz [cadena]`”, y después los identificadores de variables y constantes con el formato “`_X:\t.word [valor]`”.

- `void imprimirListaC(ListaC declarations, ListaC statements)`: Se imprimen por pantalla las directivas `.text` `.globl main` y tras esto la etiqueta `main:` que indica el comienzo del código. La función recorre primero la lista de declaraciones, para imprimir por pantalla todas las instrucciones de la lista que corresponden a la inicialización de constantes y variables. Tras eso, recorre la segunda lista, las sentencias, y las imprime todas por pantalla. Finalmente, imprime las instrucciones `li $v0, 10 -> syscall` que indican el final de la ejecución en ensamblador.

Manual de usuario:

El uso del compilador es muy sencillo, el ejecutable que realiza la compilación tiene el nombre de `minic`. Supongamos que se realiza un programa en MiniC llamado `program.mc`, para compilarlo y obtener un archivo de ensamblador `out.s` habría que ejecutar el siguiente comando:

```
$ ./minic program.mc > out.s
```

siendo `program.mc` la ruta del archivo a compilar, y `out.s` el archivo ensamblador a generar. Tanto los errores como el código generado en caso de no haber errores aparecerán en el archivo `out.s`. El código de ensamblador después se puede probar ejecutándose en simuladores de MIPS como Mars o Spim.

Ejemplo de uso:

Para probar el código generado, hemos modificado ligeramente el programa de ejemplo que se da en el enunciado para que incluya una sentencia `do-while`. El código de prueba es el siguiente:

```
void prueba() {  
    // Declaraciones  
    const a=0, b=0;  
    var c=5+2-2, d=0;  
    // Sentencias  
    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;
        }
    do {
        print "d =",d,"\n";
        d = d+1;}
    while (d-5)
    print "Final","\n";
}
```

El código que genera el compilador que hay en el archivo out.s es el siguiente:

```
#####
.data

# STRINGS #####
$str0:      .ascii "Inicio del programa\n"
$str1:      .ascii "a"
$str2:      .ascii "\n"
$str3:      .ascii "No a y b\n"
$str4:      .ascii "c ="
$str5:      .ascii "d ="
$str6:      .ascii "Final"

# IDENTIFIERS #####
_a:         .word 0
_b:         .word 0
_c:         .word 0
_d:         .word 0

#####
# Seccion de codigo
.text
.globl main
```

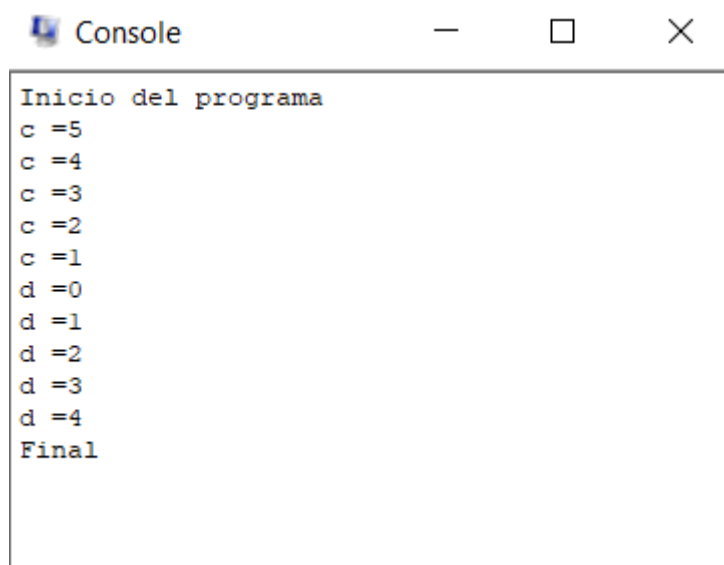
```
main:
##LongDeclarations=12
li    $t0, 0
sw    $t0, _a
li    $t0, 0
sw    $t0, _b
li    $t0, 5
li    $t1, 2
add   $t2, $t0, $t1
li    $t0, 2
sub   $t1, $t2, $t0
sw    $t1, _c
li    $t0, 0
sw    $t0, _d
##LongStatements=68
la    $a0, $str0
li    $v0, 4
syscall
lw    $t0, _a
beqz  $t0, $l4
la    $a0, $str1
li    $v0, 4
syscall
la    $a0, $str2
li    $v0, 4
syscall
b     $l5
$l4:
lw    $t1, _b
beqz  $t1, $l2
la    $a0, $str3
li    $v0, 4
syscall
b     $l3
$l2:

```

```
$l0:
    lw    $t2, _c
    beqz  $t2, $l1
    la    $a0, $str4
    li    $v0, 4
    syscall
    lw    $t3, _c
    move  $a0, $t3
    li    $v0, 1
    syscall
    la    $a0, $str2
    li    $v0, 4
    syscall
    lw    $t3, _c
    li    $t4, 2
    sub   $t5, $t3, $t4
    li    $t3, 1
    add   $t4, $t5, $t3
    sw    $t4, _c
    b     $l0
$l1:
$l3:
$l5:
$l6:
    la    $a0, $str5
    li    $v0, 4
    syscall
    lw    $t0, _d
    move  $a0, $t0
    li    $v0, 1
    syscall
    la    $a0, $str2
    li    $v0, 4
    syscall
    lw    $t0, _d
```

```
li    $t1, 1
add   $t2, $t0, $t1
sw    $t2, _d
lw    $t0, _d
li    $t1, 5
sub   $t2, $t0, $t1
bnez  $t2, $l6
la    $a0, $str6
li    $v0, 4
syscall
la    $a0, $str2
li    $v0, 4
syscall
li    $v0, 10
syscall
#####
# Fin de la ejecución #####
#####
```

Tras esto, haciendo uso del programa Spim, hemos probado el código y comprobado que efectivamente, el resultado es el esperado:



```
Inicio del programa
c =5
c =4
c =3
c =2
c =1
d =0
d =1
d =2
d =3
d =4
Final
```


Conclusiones

A través de la realización de esta práctica hemos ido aprendiendo sobre el proceso de desarrollo de un programa compilador. Utilizar un lenguaje más sencillo como MiniC nos ha parecido adecuado como introducción a los compiladores.

La implementación del analizador léxico y sintáctico fue relativamente sencilla, a excepción de algunas funciones, como la comprobación de comentarios y cadenas sin cerrar, aunque esto tampoco supuso mayor dificultad. El análisis semántico no nos resultó difícil tampoco una vez supimos cómo funcionaba la estructura de datos proporcionada.

Quizás las mayores dificultades que nos han surgido han sido a raíz de la generación de código, ya que había que tener en cuenta varias listas a la vez, operaciones entre ellas y llevar la cuenta de qué registros están libres y cuáles no, así como liberarlos en el momento adecuado. Crear funciones auxiliares para depurar el código e ir imprimiendo el proceso de generación resultó de gran ayuda para detectar los errores que iban surgiendo.