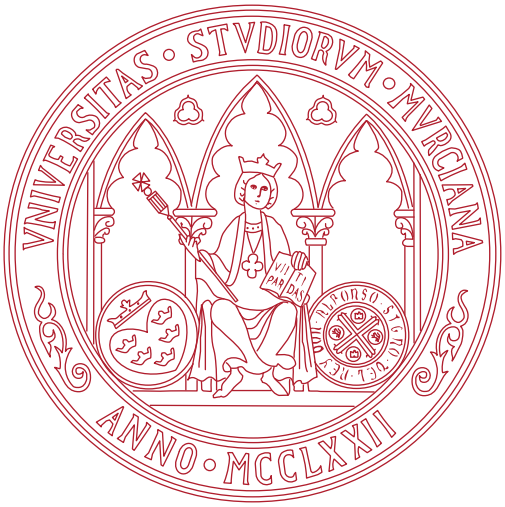

Universidad de Murcia

Grado en Ingeniería Informática

Traducción de miniC a código ensamblador de MIPS

Compiladores 2025/2026
2º Grado en Ingeniería Informática

5 de febrero de 2026

Índice general

1. Definición del problema **2**

2. Definición de los lenguajes fuente y destino **3**
2.1. Definición de *miniC* 3
2.1.1. Símbolos terminales 3
2.1.2. Una sintaxis para *miniC* 4
2.2. Descripción del código ensamblador de MIPS 5
2.3. Ejemplo de entrada/salida 8

3. Práctica 1: Análisis de léxico con Flex **11**

4. Práctica 2: Análisis sintáctico con Bison **12**

5. Práctica 3: Análisis semántico y generación de código MIPS **13**

6. Tarea final: Ejecución y evaluación del código generado **15**

7. Evaluación y normas de presentación **16**

Capítulo 1

Definición del problema

Esta práctica va a consistir en la implementación de un compilador que traduce desde el lenguaje fuente denominado *miniC* al lenguaje ensamblador de MIPS. Este proceso de traducción requiere desarrollar e integrar las siguientes fases:

- Análisis de léxico.
- Análisis sintáctico.
- Análisis semántico.
- Generación de código ensamblador.

A medida que se vayan elaborando estas fases del compilador, será necesario verificar su correcta implementación, tanto con programas de entrada válidos como con programas que contengan errores en los distintos niveles de análisis. Igualmente, se deberá verificar que el código MIPS generado es equivalente semánticamente al programa miniC de entrada.

El lenguaje usado para la implementación del compilador será C en un entorno tipo Linux. Por tanto, se emplearán las herramientas `gcc` y `make`. Además, la implementación de las fases de análisis se apoyará en herramientas que permiten generar código C a partir de expresiones regulares o gramáticas libres de contexto:

→ `flex` para generar automáticamente el analizador léxico.

→ `bison` para generar automáticamente el analizador sintáctico.

Para verificar el programa ensamblador resultante de la traducción, se podrá usar `spim` o **Mars**, a modo de intérprete de MIPS (máquina virtual).

Capítulo 2

Definición de los lenguajes fuente y destino

2.1. Definición de *miniC*

Partiremos de un lenguaje al que hemos llamado *miniC*, una versión de C muy simplificada con las siguientes restricciones:

- Sólo manejaremos constantes y variables enteras.
- El lenguaje permite operadores aritméticos de valores enteros.
- No hay operadores relacionales ni lógicos en la versión básica del lenguaje.
- En consecuencia, las expresiones aritméticas de enteros se podrán utilizar como condiciones lógicas de forma similar a C (0 es *falso* y todos los demás enteros son *verdadero*).
- No hay definición de funciones salvo la del programa principal.

2.1.1. Símbolos terminales

- **Valores literales**
  - Enteros: sólo serán válidos los enteros cuyo valor absoluto esté entre 0 y 2^{31}, para poder representarlos con el tipo `word` (4 bytes) en el ensamblador de MIPS.
  - Cadenas de caracteres: Secuencias de caracteres delimitadas por comillas dobles. Pueden contener dobles comillas siempre que estén precedidas de contrabarra \". También pueden contener caracteres de escape (tabulador \t, salto de línea \n y retorno de carro \r).
- **Identificadores**: Secuencia de letras, dígitos, y símbolos de subrayado, que no comienza por dígito y no excede los 32 caracteres.
- **Palabras reservadas**: `void`, `var`, `const`, `int`, `if`, `else`, `while`, `print` y `read`.
- **Caracteres especiales**: `;`, `,`, `+`, `-`, `*`, `/`, `=`, `(`, `)`,`{`,`}`.

2.1.2. Una sintaxis para *miniC*

program	→	`void` *id* `(` `)` `{` body `}`
body	→	body declaration
	|	body statement
	|	λ
declaration	→	`var` tipo id_list `;`
	|	`const` tipo id_list `;`
tipo	→	`int`
id_list	→	id_decl
	|	id_list `,` id_decl
id_decl	→	*id*
	|	*id* `=` expression
statement	→	*id* `=` expression `;`
	|	`{` statement_list `}`
	|	`if` `(` expression `)` statement `else` statement
	|	`if` `(` expression `)` statement
	|	`while` `(` expression `)` statement
	|	`print` `(` print_list `)` `;`
	|	`read` `(` read_list `)` `;`
statement_list	→	statement_list statement
	|	λ
print_list	→	print_item
	|	print_list `,` print_item
print_item	→	expression
	|	*string*
read_list	→	*id*
	|	read_list `,` *id*
expression	→	expression `+` expression
	|	expression `-` expression
	|	expression `*` expression
	|	expression `/` expression
	|	`-` expression
	|	`(` expression `)`
	|	*id*
	|	*num*

Esta gramática está en notación BNF, y por tanto puede ser usada directamente para construir la especificación del analizador sintáctico con Bison.

2.2. Descripción del código ensamblador de MIPS

En el desarrollo de esta práctica emplearemos un subconjunto del ensamblador de MIPS. Este conjunto es suficiente para la implementación de la práctica aquí descrita, pero puede ser necesario ampliarlo si, opcionalmente, se desea aumentar el lenguaje de entrada con nuevas características.

El fichero ensamblador de salida generado al compilar un fichero de miniC debe comenzar con una representación de la tabla de símbolos construida durante el proceso de análisis. Esta tabla debe tener el siguiente formato:

```
1         .data
2
3  # Cadenas del programa
4  $str1:
5         .asciiz "Cadena1"
6  $str2:
7         .asciiz "Cadena2"
8  $str3:
9         .asciiz "Cadena3"
10
11 # Variables y constantes
12 _x:
13        .word 0
14 _y:
15        .word 0
16 _z:
17        .word 0
```

La tabla de símbolos se encuentra en el segmento de datos del programa, conforme establece la primera línea con `.data`. Los comentarios en ensamblador MIPS quedan indicados con `#`, y son opcionales. Cada cadena de caracteres encontrada en el programa fuente quedará reflejada en la sección de datos con una posición de memoria indicada usando un identificador con el formato `$strX`, donde `X` es un contador que comenzará en 1 y se incrementará en una unidad por cada cadena encontrada. Las cadenas se deben almacenar como texto ascii terminado en un carácter nulo, mediante la directiva `.asciiz`.

Las variables o constantes enteras pueden situarse en la sección de datos como memoria de tipo `.word`, es decir, como datos de 32 bits. La inicialización de su valor será siempre a 0. El nombre de la variable se empleará como identificador de la posición de memoria de la variable, pero deberá ir precedido del carácter `_` con el fin de evitar que los nombres de variables usados por el usuario en el fichero de entrada puedan colisionar con nombres de operaciones del ensamblador de MIPS.

A continuación de la sección de datos, comenzará la sección de texto que contiene las instrucciones del código ensamblador. En esta sección se debe definir forzosamente el punto de entrada al programa, que será una posición etiquetada con `main` y declarada con `.globl`:

```
1         .text
2
3         .globl main
4  main:
5         # Aquí comienzan las instrucciones del programa
```

Para la implementación de la práctica serán suficientes algunas instrucciones del ensamblador MIPS:

Instrucción	Descripción	Ejemplo
`li reg, val`	Carga en el registro `reg` el valor inmediato `val`.	`li $t1, 50`
`lw reg, mem`	Carga en el registro `reg` el valor `word` almacenado en la posición de memoria indicada por `mem`.	`lw $t1, _x`
`la reg, mem`	Carga en el registro `reg` la posición de memoria indicada por `mem`.	`la $a0, $str1`
`sw reg, mem`	Almacena en la memoria indicada por `mem` el valor `word` del registro `reg`.	`sw $t1, _x`
`move reg1, reg2`	Copia al registro `reg1` el valor del registro `reg2`.	`move $a0, $t1`
`add regs, reg1, reg2`	Suma el valor del registro `reg1` con el del registro `reg2` y almacena el resultado en el registro `regs`.	`add $t3, $t1, $t2`
`addi regs, reg1, val`	Suma el valor del registro `reg1` con el valor entero `val` y almacena el resultado en el registro `regs`.	`addi $t3, $t1, 1`
`sub regs, reg1, reg2`	Resta al valor del registro `reg1` el del registro `reg2` y almacena el resultado en el registro `regs`.	`sub $t3, $t1, $t2`
`mul regs, reg1, reg2`	Multiplica el valor del registro `reg1` por el del registro `reg2` y almacena el resultado en el registro `regs`.	`mul $t3, $t1, $t2`
`div regs, reg1, reg2`	Divide el valor del registro `reg1` por el del registro `reg2` y almacena el resultado en el registro `regs`. La división es entera.	`div $t3, $t1, $t2`
`neg regs, reg`	Invierte el signo del valor del registro `reg` y almacena el resultado en el registro `regs`.	`neg $t2, $t1`
`b etiq`	Salto incondicional a la posición de código indicada por `etiq`	`b $l1`
`beqz reg, etiq`	Salto a la posición de código indicada por `etiq` en el caso de que el valor del registro `reg` sea igual a cero.	`beqz $t1, $l1`
`bnez reg, etiq`	Salto a la posición de código indicada por `etiq` en el caso de que el valor del registro `reg` no sea igual a cero.	`bnez $t1, $l1`
`syscall`	Llamada a un servicio del sistema (ver más adelante)	`syscall`
`jal etiq`	Salta a la posición de código indicada por `etiq` y guarda en `$ra` la dirección de retorno	
`jr $ra`	Salta a la posición de código indicada por el registro `$ra`	

Los saltos a posiciones de código tienen siempre un argumento que es una etiqueta de salto. Las etiquetas de salto se establecen en el código ensamblador mediante una cadena terminada con dos puntos:

```
1 $l1:
2         # Aquí comienzan las instrucciones apuntadas por $l1
```

En la arquitectura MIPS, existen 32 registros cuyo uso sigue una convención, tal y como se lista a continuación:

Nombre del registro	Número	Uso
$zero	0	Constante 0.
$at	1	Reservada.
$v0	2	Evaluación de expresiones y resultados de una función.
$v1	3	Evaluación de expresiones y resultados de una función.
$a0	4	Argumento 1 de una función.
$a1	5	Argumento 2 de una función.
$a2	6	Argumento 3 de una función.
$a3	7	Argumento 4 de una función.
$t0	8	Variable temporal volátil (no preservada en llamadas).
$t1	9	Variable temporal volátil (no preservada en llamadas).
$t2	10	Variable temporal volátil (no preservada en llamadas).
$t3	11	Variable temporal volátil (no preservada en llamadas).
$t4	12	Variable temporal volátil (no preservada en llamadas).
$t5	13	Variable temporal volátil (no preservada en llamadas).
$t6	14	Variable temporal volátil (no preservada en llamadas).
$t7	15	Variable temporal volátil (no preservada en llamadas).
$s0	16	Variable temporal no volátil (preservada en llamadas).
$s1	17	Variable temporal no volátil (preservada en llamadas).
$s2	18	Variable temporal no volátil (preservada en llamadas).
$s3	19	Variable temporal no volátil (preservada en llamadas).
$s4	20	Variable temporal no volátil (preservada en llamadas).
$s5	21	Variable temporal no volátil (preservada en llamadas).
$s6	22	Variable temporal no volátil (preservada en llamadas).
$s7	23	Variable temporal no volátil (preservada en llamadas).
$t8	24	Variable temporal volátil (no preservada en llamadas).
$t9	25	Variable temporal volátil (no preservada en llamadas).
$k0	26	Reservada para el kernel del SO.
$k1	27	Reservada para el kernel del SO.
$gp	28	Puntero a área global.
$sp	29	Puntero a la pila.
$fp	30	Puntero al frame de llamada.
$ra	31	Dirección de retorno (usada en llamadas a funciones).

De estos registros, los más prácticos para la implementación del compilador son los registros temporales `$t0-$t9` o `$s0-$s7`.

El lenguaje miniC permite la utilización de la instrucción `print` para la generación de información por la salida estándar. La información que se puede enviar a salida estándar es, o bien una cadena de caracteres, o bien un valor entero. Para generar esta salida es necesario traducir estas operaciones de impresión mediante llamadas a servicios del sistema, usando `syscall`. Dichas llamadas funcionan cargando, en primer lugar, el identificador del servicio que se desea invocar en el registro `$v0`, y el dato que se desea imprimir en el registro `$a0`.

Por ejemplo, el servicio de impresión de una cadena de caracteres tiene el código de llamada 4, y el registro `$a0` debe contener la dirección de comienzo de la cadena:

```
li $v0, 4
la $a0, $str
syscall
```

Para imprimir un entero se debe usar la llamada con código 1, almacenando previamente en `$a0` el valor a imprimir. Si ese valor se encuentra en el registro `$ti`, siendo `$ti` uno de los registros temporales `$t0-$t9`:

```
li $v0, 1
move $a0, $ti
syscall
```

El lenguaje miniC también permite la utilización de la instrucción `read` para la lectura de valores enteros. Para leer un entero con el ensamblador de MIPS, se debe usar la llamada al sistema con código 5. El valor leído se almacena en el registro `$v0`, de modo que una instrucción miniC como `read x` producirá el código ensamblador siguiente:

```
li $v0, 5
syscall
sw $v0, _x
```

Para finalizar el programa producido por el compilador, se realiza una llamada al sistema de tipo *exit*:

```
# Final: exit
li $v0, 10
syscall
```

2.3. Ejemplo de entrada/salida

El siguiente fichero de entrada en *miniC*:

```
void main() {
  const int a = 0, b = 0;
  print ("Inicio del programa\n");
  var int c = 5+2-2;
  if (a) print ("a","\n");
  else if (b) print ("No a y b\n");
        else while (c) {
               print ("c = ",c,"\n");
               c = c-2+1;
             }
  print ("Final","\n");
}
```

puede traducirse a una salida parecida a la siguiente:

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

$str1:
    .asciiz "Inicio del programa\n"
$str2:
    .asciiz "a"
$str3:
```

```
10      .asciiz "\n"
11  $str4:
12      .asciiz "No a y b\n"
13  $str5:
14      .asciiz "c = "
15  $str6:
16      .asciiz "\n"
17  $str7:
18      .asciiz "Final"
19  $str8:
20      .asciiz "\n"
21  _a:
22      .word 0
23  _b:
24      .word 0
25  _c:
26      .word 0
27
28  ####################
29  # Seccion de codigo
30      .text
31      .globl main
32  main:
33      li $t0, 0
34      sw $t0, _a
35      li $t0, 0
36      sw $t0, _b
37      la $a0, $str1
38      li $v0, 4
39      syscall
40      li $t0, 5
41      li $t1, 2
42      add $t2, $t0, $t1
43      li $t0, 2
44      sub $t1, $t2, $t0
45      sw $t1, _c
46      lw $t0, _a
47      beqz $t0, $l5
48      la $a0, $str2
49      li $v0, 4
50      syscall
51      la $a0, $str3
52      li $v0, 4
53      syscall
54      b $l6
55  $l5:
56      lw $t1, _b
57      beqz $t1, $l3
58      la $a0, $str4
59      li $v0, 4
60      syscall
61      b $l4
62  $l3:
63  $l1:
64      lw $t2, _c
65      beqz $t2, $l2
66      la $a0, $str5
67      li $v0, 4
```

```
68      syscall
69      lw $t3, _c
70      move $a0, $t3
71      li $v0, 1
72      syscall
73      la $a0, $str6
74      li $v0, 4
75      syscall
76      lw $t3, _c
77      li $t4, 2
78      sub $t5, $t3, $t4
79      li $t3, 1
80      add $t4, $t5, $t3
81      sw $t4, _c
82      b $l1
83  $l2:
84  $l4:
85  $l6:
86      la $a0, $str7
87      li $v0, 4
88      syscall
89      la $a0, $str8
90      li $v0, 4
91      syscall
92
93  ###############
94  # Fin
95      li $v0, 10
96      syscall
```

Capítulo 3

Práctica 1: Análisis de léxico con Flex

Tareas

- **Identificación de** los **tokens** que aparecen en la definición de la gramática de 2.1. En particular, el analizador léxico deberá reconocer:
  - Las palabras reservadas y los caracteres especiales.
  - Identificadores, números y cadenas.
- **Definición de expresiones regulares** (tipo *Flex*) correspondientes a cada uno de esos tokens y de aquellas que sirvan para reconocer comentarios tipo C.
  - Los *comentarios multilínea* comienzan por `/*` y terminan con `*/`.
  - Los *comentarios de línea* comienzan con `//`.
- **Creación del fichero *Flex*** que genere el fichero C para reconocer la secuencia de tokens correspondiente a un programa de entrada dado, ignorando los comentarios y espacios en blanco que puedan aparecer.
- Implementación de la **recuperación de errores** en modo pánico.
- **Verificar la corrección** del analizador léxico usando ficheros de prueba que deberán incluirse en la entrega de la práctica y comentarse en la memoria.

Capítulo 4

Práctica 2: Análisis sintáctico con Bison

Tareas

- Inclusión en el fichero *Flex* anterior de acciones para devolver **atributos** al analizador sintáctico.
- Crear el fichero *Bison*, que funcione conjuntamente con el analizador léxico, para reconocer sintácticamente ficheros descritos por la gramática de miniC. Para ello, será necesario:
  - Estudiar la necesidad de definir precedencias y asociatividades para los operadores aritméticos, con el fin de que las expresiones no sean ambiguas.
  - Añadir las acciones semánticas necesarias para generar la secuencia de reglas aplicadas para reconocer un programa de entrada.
- Estudiar los posibles conflictos desplazamiento/reducción y reducción/reducción.
- Realizar, opcionalmente, una recuperación de errores.
- **Verificar la corrección** del analizador sintáctico usando ficheros de prueba que deberán incluirse en la entrega de la práctica y comentarse en la memoria.

Capítulo 5

Práctica 3: Análisis semántico y generación de código MIPS

Análisis semántico

Consistirá básicamente en la gestión de declaración y uso de variables y constantes. Para hacerlo:

- Hay que **definir la tabla de símbolos** como una estructura de datos que permita almacenar las variables, constantes y cadenas declaradas en el programa. Se proporcionará una implementación de la tabla[1] para facilitar la tarea, aunque su uso no es obligatorio.
- Será necesario **dar de alta a las variables y constantes** en la *tabla de símbolos* cuando aparezcan en una declaración. Esto podrá llevarse a cabo con acciones en las reglas de producción que van generando las declaraciones de variables. Inicialmente sólo se permiten enteros de tipo word.
- Será necesario **controlar** que una variable o constante no se declare más de una vez, y que no se use sin haber sido declarada. También será necesario controlar que no se reasignen valores a las constantes. En otro caso, se emitirá un mensaje de error.
- Será necesario **controlar** que el identificador del programa tiene el lexema `main`.
- **Verificar la corrección** del analizador semántico usando ficheros de prueba que deberán incluirse en la entrega de la práctica y comentarse en la memoria.

Generación de código

Para realizar esta tarea se proporcionará una implementación de un tipo `lista` en C, con la estructura de datos en la que iremos almacenando el código generado[2]. Cada sentencia de *miniC* corresponderá a un conjunto de operaciones, implementadas cada una de ellas como una estructura con cuatro campos, es decir, en forma de cuádruplas. El código de salida será, por tanto, una **lista enlazada** (dinámica) de cuádruplas. Tenemos fundamentalmente dos opciones para conseguirlo:

- La primera consiste en asociar a cada no terminal principal de la gramática, un atributo del tipo `lista` (de código). Habría que ir uniendo listas enlazadas en una única lista cada vez que necesitemos concatenar código según la construcción ascendente del árbol sintáctico para, finalmente, terminar con la lista enlazada correspondiente al código completo, como contenido de un atributo asociado a los símbolo `body` de la regla de `program`. Debemos construir una

[1] En la carpeta "Lista de símbolos" que encontraréis dentro de "Prácticas" en los Recursos del Aula Virtual.
[2] En la carpeta "Lista para código" que encontraréis dentro de "Prácticas" en los Recursos del Aula Virtual.

función para imprimir las listas de código en el fichero de salida a continuación de la tabla de símbolos.

- También sería posible trabajar con una estructura global que se iría actualizando conforme se va reconociendo el programa fuente. En este caso habría que usar acciones en mitad de las reglas de producción de *Bison* y el acceso directo a valores de la pila de dicha herramienta. También se necesita una función para imprimir el código completo en el fichero de salida, a continuación de la tabla de símbolos.

En cualquier caso debemos añadir al fichero bison los **atributos** y las **acciones semánticas** necesarias para generar código con el formato de la sección 2.2. Para esto:

- Usaremos la *tabla de símbolos* para **generar** la primera parte del programa en ensamblador, es decir, **la sección de datos**. Es importante observar que los registros `$t0-$t9` o `$s0-$s7` se pueden usar como variables temporales que no es necesario declarar en la tabla de símbolos.
- Añadiremos además, a cada regla de producción, las acciones en C necesarias para **traducir**, por un lado las **expresiones aritméticas** y, por otro, las **sentencias de control** del lenguaje fuente a código MIPS. De esta forma, completaremos el programa objeto con la **sección de código**.
- Finalmente, debemos **verificar la corrección** del código generado usando ficheros de prueba que deberán incluirse en la entrega de la práctica y comentarse en la memoria, mostrando la salida de su ejecución con `spim` o Mars.

Capítulo 6

Tarea final: Ejecución y evaluación del código generado

- Ejecutar las prácticas realizadas a lo largo del curso con diferentes programas de entrada:

  `./minic programa.mc > programa.s`

  donde la extensión `.mc` se puede utilizar para representar a ficheros fuente escritos en miniC.
- Usar la salida generada en código ensamblador como entrada del intérprete `spim` o Mars:

  `./spim -file programa.s`
- Comprobar que el código funciona tal y como esperábamos cuando escribimos el programa fuente.

Capítulo 7

Evaluación y normas de presentación

Para la **evaluación** de la parte práctica de la asignatura (ver guía docente) se tendrán en cuenta las siguientes consideraciones:

- Debe realizarse, en la medida de lo posible, por parejas.
- Para aprobar la parte de prácticas de la asignatura es necesaria la correcta implementación de la parte obligatoria de las prácticas, que consiste en la validación y la generación de código de todas las sentencias del lenguaje miniC indicadas en la gramática del enunciado.
- La parte obligatoria de las prácticas puntúa hasta 8.
- Se podrán implementar mejoras para obtener hasta un 10, **siguiendo este orden**:
  1. La implementación de una sentencia `do-while` se valora con 0,5 puntos adicionales.
  2. La implementación de los operadores relacionales (`<`, `>`, `<=`, `>=`, `==`, `!=`) para las condiciones de las sentencias se valora con 0,5 puntos adicionales. Estos operadores se deberán incluir en el no terminal `expression` de la gramática, no deben ser asociativos y deben tener precedencia inferior a los operadores aritméticos.
  3. La implementación de una sentencia `break` se valora con 1 punto adicional. La sentencia sólo se podrá utilizar en el interior de una sentencia `while` o `do-while`, y deberá provocar el salto a la etiqueta de fin de dicha sentencia. Hay que tener en cuenta que estas sentencias pueden estar contenidas dentro de otras similares. La sentencia `break` deberá saltar a la etiqueta de fin del `while` o `do-while` que lo contenga directamente.
- La obtención de la puntuación indicada está condicionada al correcto funcionamiento del compilador, siguiendo las especificaciones de este documento. Se considerará suspensa cualquier práctica que, en alguno de sus módulos (léxico, sintáctico, semántico o generación de código) no supere el proceso de verificación o provoque violaciones de segmento.
- En caso de que se le convoque a una entrevista de prácticas, debe contestar de forma satisfactoria a las cuestiones planteadas.
- Será necesario entregar:
  - Los fuentes, que habrá que subir al Aula Virtual (como respuesta a la correspondiente tarea), junto con un `makefile`. Sólo debe entregarse el código realizado por los alumnos.
  - Ficheros de prueba que verifiquen el correcto funcionamiento del compilador, incluyendo todos los operadores y sentencias posibles implementados en el lenguaje.

    - Una memoria en la que se indique claramente: Nombre y correo electrónico de los componentes del grupo, convocatoria, explicación completa de la práctica (funciones principales, estructuras de datos, manual de usuario para la ejecución, etc.) y ejemplos de funcionamiento explicados (con la entrada y salida correspondientes). En caso de implementar alguna mejora, será obligatoria la inclusión de una sección en la que se describa de forma completa la solución propuesta, teniendo en cuenta las ampliaciones requeridas en todas las fases del compilador.

- Para superar la parte práctica de la asignatura será necesario obtener la calificación de **apto** en el control de prácticas que será convocado junto con el examen de la parte de teoría.

Bibliografía

[AHO] Alfred V. Aho, Monica Lam, Ravi Sethi, Jeffrey D. Ullman, Compiladores. Principios, Técnicas y Herramientas, Addison-Wesley, 2008.

[DON] C. Donnelly y R. Stallman, Bison. The Yacc-compatible parser generator, v2.4.2 (2010).

[PAX] V. Paxson. Lexical Analysis With Flex (2007).

[SPIM] J. Larus, A MIP32 Simulator. http://spimsimulator.sourceforge.net.