# 11

# El proceso de compilación

## Contenido



Compiladores es un capítulo de *Análisis y Diseño de Algoritmos* de Gustavo López, Ismael Jeder y Augusto Vega, quienes amablemente nos autorizaron a incluirlo como lectura complementaria de *Arquitectura de Computadoras* de Patricia Quiroga.

## Objetivos

- Entender como es el proceso de compilación, el proceso de traducción de un código fuente a lenguaje maquina, para que pueda ser ejecutado por la computadora u ordenador.
- Conocer las etapas más importantes de la compilación (preprocesamiento, generación de código y enlace).
- Aprender a automatizar la compilación, técnica que se torna relevante cuando los proyectos son de gran envergadura.

## 11.1 Introducción.

En este punto del libro se espera que el lector haya incorporado los conceptos fundamentales que le permitan desarrollar algoritmos correctos para la resolución de problemas. Por eso es que nos permitiremos ver con mayor detalle temas relacionados con el ambiente de desarrollo para la implementación y la compilación de algoritmos. En particular, será de interés analizar por separado las etapas más importantes que comprenden el proceso de compilación (preprocesamiento, generación de código y enlace), y también se presentará una forma de automatizar ese proceso (mediante la herramienta `make`), en especial cuando los proyectos adquieren tamaños considerables e involucran varios archivos fuente, de cabecera o bibliotecas.

En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

## 11.2 El proceso de compilación.

En el ámbito de las computadoras, los algoritmos se expresan mediante lenguajes de programación, como C, Pascal, Fortran o Java (entre muchos otros). Sin embargo, esta representación no es suficiente, ya que el microprocesador necesita una expresión mucho más detallada del algoritmo, que especifique en forma explícita todas las señales eléctricas que involucra cada operación. La tarea de traducción de un programa desde un lenguaje de programación de alto nivel hasta el lenguaje de máquina se denomina compilación, y la herramienta encargada de ello es el compilador. En la figura siguiente se pueden distinguir las etapas más importantes:

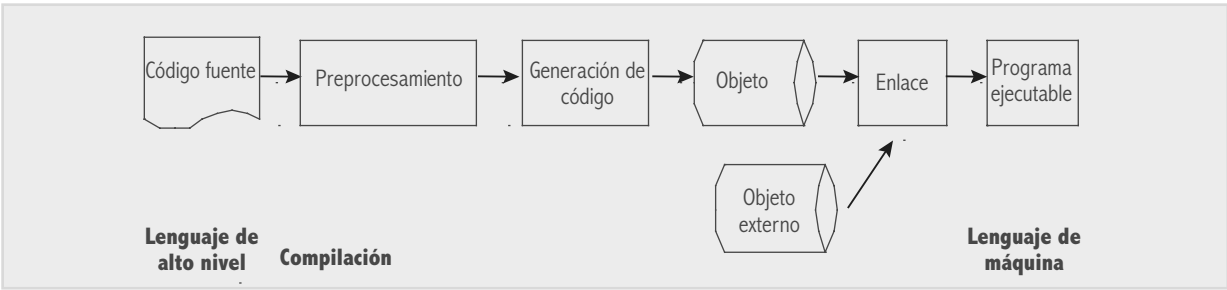



Fig. 11-1. Etapas más importantes de compilación.

A continuación se tratarán estas etapas de manera más extensa, con mayor atención en la primera de ellas: El preprocesamiento.

## 11.3 Preprocesamiento.

La primera etapa del proceso de compilación se conoce como preprocesamiento. En ocasiones a esta etapa ni siquiera se la considera parte de la compilación, ya que es una traducción previa y básica que tiene como finalidad "acomodar" el código fuente antes de que éste sea procesado por el compilador en sí. En este libro no haremos esa distinción.

El preprocesador modifica el código fuente según las directivas que haya escrito el programador. En el caso del lenguaje C, estas directivas se reconocen en el código fuente porque comienzan con el carácter #, y se utilizaron en gran medida a lo largo de este libro (por ejemplo, `#define`... o `#include`...). A continuación, veremos algunas de ellas:

```
#define, #undef, #error, #include, #if, #ifdef, #ifndef, #else,
#endif, #pragma.
```

#### 11.3.1 Directivas #define #undef.

```
#define <nombre macro>   <expansión de la macro>
#undef <nombre identificador>
```

La directiva #define crea un identificador con nombre <nombre macro> y una cadena de sustitución de ese identificador. Cada vez que el preprocesador encuentre el identificador de la macro, realizará la sustitución de éste por la cadena de sustitución. Esta última termina con un salto de línea. Cuando la cadena ocupa más de una línea, se utiliza la barra invertida para indicarle al compilador que omita el salto de línea. Algunos ejemplos:

```
#define MAX_PILA         100
#define VAL_ABS(a) ((a) < 0 ? -(a) : (a))
```

Por su parte, #undef elimina valores definidos por #define.

#### 11.3.2 Directiva #error.

```
#error <mensaje de error>
```

Esta directiva fuerza al compilador a detener la compilación del programa. El mensaje de error no va entre comillas.

#### 11.3.3 Directiva #include.

```
#include "archivo de cabecera"
#include <archivo de cabecera>
```

Esta directiva indica al preprocesador que incluya el archivo de cabecera indicado. Los archivos incluidos, a su vez, pueden tener directivas #include. Si el nombre del archivo está entre llaves < y > significa que se encuentra en alguno de los directorios que almacenan archivos de cabecera (los estándares del compilador y los especificados con la opción -I cuando se ejecuta la compilación). En el otro caso, significa que el archivo que se ha de incluir se encuentra en el directorio actual de trabajo (el directorio a partir del cual se ejecuta el compilador).

#### 11.3.4 Directivas #if #ifdef #ifndef #else #endif.

```
#if <expresión constante>
  <secuencia de sentencias>
#endif

#ifdef <identificador>
  <secuencia de sentencias>
#endif

#ifndef <identificador>
  <secuencia de sentencias>
#endif
```

Estas directivas permiten incluir o excluir condicionalmente partes del código durante la compilación. Si `<expresión constante>` es verdadera, se compila el código encerrado entre `#if` y `#endif`; en caso contrario, se ignora ese código.

En el segundo caso, si `<identificador>` está definido (por ejemplo, usando la directiva `#define`), entonces el código encerrado entre `#ifdef` y `#endif` es compilado.

En el tercer caso, si `<identificador>` **no** está definido, entonces el código encerrado entre `#ifndef` y `#endif` es compilado.

La directiva `#else` establece una alternativa:

```
#if <expresión constante>
  <secuencia de sentencias 1>
#else
  <secuencia de sentencias 2>
#endif
```

#### 11.3.5 Directiva #pragma.

```
#pragma <nombre directiva>
```

Un `pragma` es una directiva que permite proveer información adicional al compilador. Depende del compilador utilizado y de la arquitectura donde se ejecutará la aplicación compilada.

En el caso de Pascal, el preprocesamiento también está gobernado a partir de un conjunto de directivas, de las cuales se detallarán sólo las que se utilizan con mayor frecuencia. A diferencia de C, una directiva en Pascal se define con la sintaxis `{$ … }`. Algunas de ellas:

```
{$define}, {$undef}, {$ifdef }, {$else}, {$endif}, {$I}
```

#### 11.3.6 Directivas {$define} {$undef}.

```
  {$define <nombre identificador>}
{$undef <nombre identificador>}
```

Al igual que en el caso de C, las directivas `{$define}` y `{$undef}` en Pascal permiten definir y eliminar la definición de un identificador, respectivamente. A diferencia de C, no todos los compiladores permiten definir macros (un identificador con una cadena de sustitución asociada, que será reemplazada cada vez que se encuentre una ocurrencia del identificador).

#### 11.3.7 Directivas{$ifdef} {$else} {$endif}.

```
{$ifdef <identificador>}
  <secuencia de sentencias 1>
{$else}
  <secuencia de sentencias 2>
{$endif}
```

La compilación condicional de código en Pascal está soportada mediante las directivas `{$ifdef }`, `{$else}` y `{$endif}`. En el ejemplo siguiente se cargan dos números de punto flotante y, en función de si el identificador `DOBLE_PRECISION` está definido o no, se computa su suma en precisión doble o simple:

```
program ejemplo;

{$define DOBLE_PRECISION}

{$ifdef DOBLE_PRECISION}
var num1, num2, resultado: double;
{$else}
var num1, num2, resultado: real;
{$endif}

begin

{$ifdef DOBLE_PRECISION}
  writeln('Ingrese dos números de precisión doble:');
{$else}
  writeln('Ingrese dos números de precisión simple:');
{$endif}

  readln(num1);
  readln(num2);

  resultado := num1 + num2;

{$ifdef DOBLE_PRECISION}
  writeln(num1:6:4, ' + ', num2:6:4, ' = ', resultado:6:4);
{$else}
  writeln(num1:6:2, ' + ', num2:6:2, ' = ', resultado:6:2);
{$endif}

end.
```

#### 11.3.8 Directiva {$I}.

```
{$I+} / {$I-}
{$I <nombre de archivo>}
```

La directiva `{$I}` tiene dos usos. En el primer caso, cuando se la acompaña de los modificadores + y –, permite habilitar e inhabilitar la comprobación de errores de entrada/salida, respectivamente. Su uso ya se explicó en el capítulo 7, ya que está estrechamente ligado al manejo de archivos. En el ejemplo siguiente se escribe sobre un archivo de texto. Para evitar que el programa se interrumpa con un error en la llamada a `append()` en caso de que el archivo no exista, se utilizan las directivas `{$I-}` y `{$I+}`, y se verifica el contenido de la variable predefinida `ioresult` (si su valor es distinto de 0, significa que se produjo un error en la última operación de entrada/salida).

```
program ejemplo;

var archivo: text;

begin
  assign(archivo, 'mi_archivo.txt');

  {$I-}
  append(archivo);
  {$I+}
  if (ioresult <> 0) then
    rewrite(archivo);

        …

  close(archivo);

end.
```

Con esta introducción se cubren los aspectos relacionados al preprocesamiento. La etapa que sigue, la compilación en sí, se explicará a grandes rasgos a continuación.

## 11. 4 Compilación.

La compilación es el proceso que, en sí, traduce el lenguaje de alto nivel en lenguaje de máquina. Dentro de esta etapa pueden reconocerse, al menos, cuatro fases:

1. Análisis léxico.
2. Análisis sintáctico.
3. Análisis semántico.
4. Generación de código.

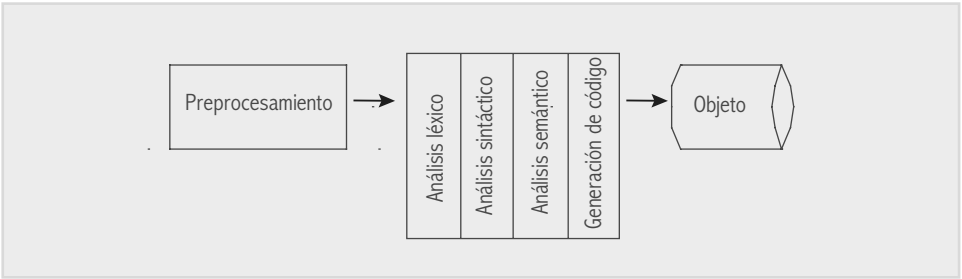



Fig. 11-2. Fases de la compilación.

El **análisis léxico** extrae del archivo fuente todas las cadenas de caracteres que reconoce como parte del vocabulario y genera un conjunto de *tokens* como salida. En caso de que parte del archivo de entrada no pueda reconocerse como lenguaje válido, se generarán los mensajes de error correspondientes.

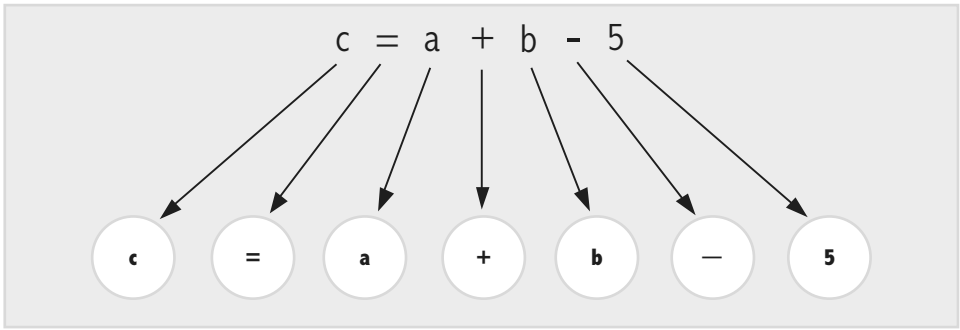



Fig. 11-3. Análisis léxico.

A continuación, durante el **análisis sintáctico**, se procesa la secuencia de *tokens* generada con anterioridad, y se construye una representación intermedia, que aún no es lenguaje de máquina, pero que le permitirá al compilador realizar su labor con más facilidad en las fases sucesivas. Esta representación suele ser un árbol.

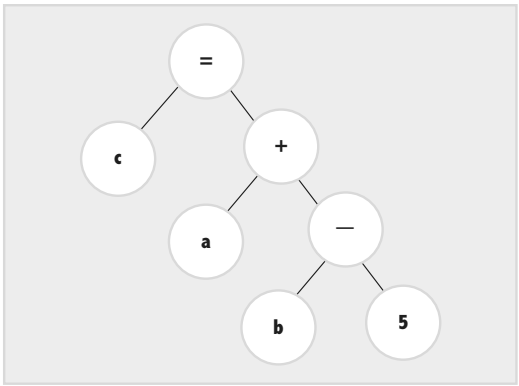



Fig. 11-4. Análisis sintáctico.

Durante el **análisis semántico** se utiliza el árbol generado en la fase previa para detectar posibles violaciones a la semántica del lenguaje de programación, como podría ser la declaración y el uso consistente de identificadores (por ejemplo, que el tipo de dato en el lado derecho de una asignación sea acorde con el tipo de dato de la variable destino, en el lado izquierdo de la asignación).

Por último, en la **generación de código** se transforma la representación intermedia en lenguaje de máquina (código objeto). En los casos típicos esta fase involucra mucho trabajo relacionado con la optimización del código, antes de generarse el lenguaje de máquina.

## 11. 5 Enlace.

No siempre las aplicaciones se construyen de manera monolítica, a partir de un solo archivo fuente. En la práctica sólo se escribe una parte, y lo demás se toma de bibliotecas externas que, en la última etapa de la compilación, se enlazarán unas con otras para generar la aplicación ejecutable final. Ésta es, básicamente, la tarea del enlazador.

Los archivos objeto que se enlazan con nuestro programa se denominan bibliotecas externas que, por su parte, pueden haber sido construidas por nosotros mismos o pueden provenir de terceras partes (por ejemplo, las bibliotecas estándares del compilador). Una biblioteca, en este contexto, es una colección de funciones. Este tipo de archivos almacena el nombre de

En el caso particular del lenguaje C, cada biblioteca externa está acompañada de un archivo de cabecera ("header file") con extensión ".h". Se trata de un archivo de texto que contiene los encabezados (prototipos) de cada función de la biblioteca, además de otras definiciones de tipos o macros. Resumiendo, el archivo de cabecera contiene la parte "pública" de la biblioteca, a lo que el usuario tiene acceso. Es importante aclarar que, si bien es posible hacerlo, en los archivos de cabecera no debe incluirse código (solo definiciones de prototipos, tipos de datos o macros).

cada función, los códigos objeto de las funciones y la información de reubicación necesaria para el proceso de enlace. Entonces, en el proceso de enlace sólo se añade al código objeto el código de la función a la que se hizo referencia.

## 11.6 Automatización del proceso de compilación.

A lo largo de este libro se consideró únicamente la compilación de programas aislados, con un solo archivo fuente. Sin embargo, en la práctica, esta situación no suele ser la más habitual, en cambio, nuestros proyectos por lo general están conformados por varios archivos que, luego del proceso de compilación, darán lugar a una aplicación ejecutable. A medida que estos proyectos crecen en cuanto a cantidad de archivos, la compilación puede tornarse engorrosa. Felizmente, existe la posibilidad de automatizar el proceso de compilación mediante el uso de la herramienta `make`.

A medida que se escriben programas más largos, se observa que la compilación toma cada vez más tiempo. En general, cuando el programador corrige, modifica o mantiene un programa, trabaja en una sección acotada de él, y la mayor parte del código existente no cambia. La herramienta `make` es capaz de distinguir los archivos fuente que fueron alterados, lo que evita la compilación innecesaria de los que no se modificaron. `Make` construye proyectos sobre la base de comandos contenidos en un archivo de descripción o archivo de dependencias, comúnmente denominado `makefile`.

Para ilustrar el uso de la herramienta `make`, vamos a analizar un ejemplo simple. En este caso se cuenta con tres archivos propios y, además, se accede a la biblioteca estándar `stdio.h`. El archivo `mi_aplicacion.c` contiene el programa principal. Su trabajo es solicitar al usuario dos valores numéricos y sumarlos, para lo que invoca la función auxiliar `mi_suma()`, que reside en los archivos `mis_funciones.h` y `mis_funciones.c` (puede afirmarse que `mis_funciones` es una biblioteca externa). Además, el programa principal también hace uso de las funciones `scanf()` y `printf()`, pertenecientes a `stdio`. El código fuente de los archivos involucrados se muestra a continuación:

```
/* mi_aplicacion.c */

#include <stdio.h>
#include "mis_funciones.h"

int main()
{
  int a,b,c;

  scanf("%d", &a);
  scanf("%d", &b);

  c = mi_suma(a,b);

  printf("%d + %d = %d\n", a, b, c);

  return 0;
}
```

```
/* mis_funciones.h */

int mi_suma(int a, int b);
```

```
/* mis_funciones.c */

#include "mis_funciones.h"

int mi_suma(int a, int b)
{
  return a + b;
}
```

El archivo `makefile` define las reglas de dependencia entre estos archivos, además de especificar cómo deben compilarse (compilador que se debe usar, opciones de compilación, etc.):

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
```

```
mi_aplicacion: mi_aplicacion.o mis_funciones.o
  $(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o

mi_aplicacion.o: mi_aplicacion.c mis_funciones.h
  $(CC) $(CFLAGS) -c -o mi_aplicacion.o mi_aplicacion.c

mis_funciones.o: mis_funciones.c mis_funciones.h
  $(CC) $(CFLAGS) -c -o mis_funciones.o mis_funciones.c

clean:
  rm *.o mi_aplicacion
```

La primera línea define el compilador a utilizar (`CC=gcc`) y la segunda, las opciones de compilación (`CFLAGS=-Wall -ansi -pedantic`). Luego siguen tres reglas; la primera de ellas especifica cómo construir la aplicación `mi_aplicacion`, que depende de `mi_aplicacion.o` y `mis_funciones.o`. (Nótese que debajo de cada dependencia se encuentra el comando de compilación para ejecutar: `$(CC) -o mi_aplicacion...`). Las reglas escritas en el `makefile` conforman un grafo de dependencias:

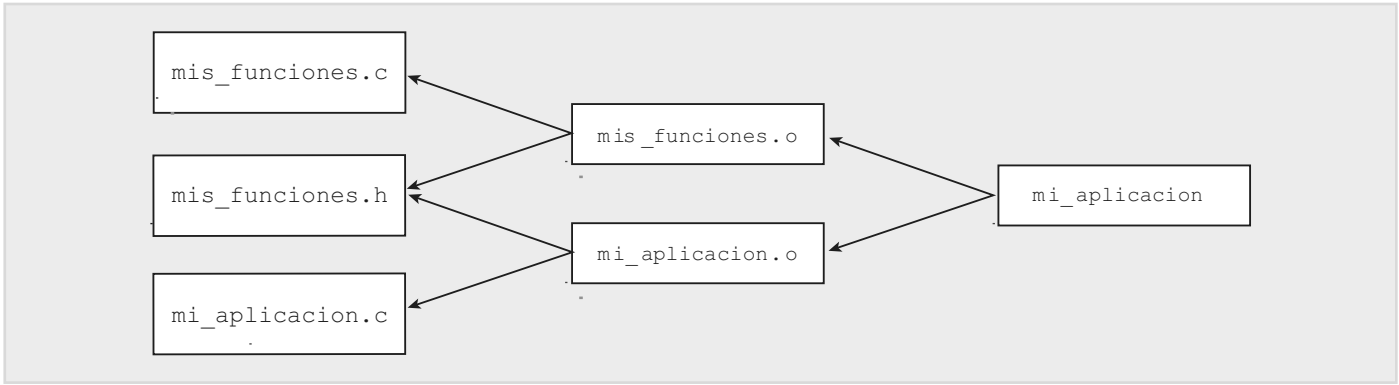



Fig. 11-5. Grafo de dependencias.

Al observar este grafo de dependencias puede deducirse que en caso de modificarse el archivo `mis_funciones.c`, sólo sería necesario regenerar el objeto `mis_funciones.o` y el ejecutable `mi_aplicacion`.

En el `makefile` cada dependencia se expresa utilizando el siguiente formato:

```
objetivo: dependencias (archivos fuentes, objetos)
        comando de compilación (debe estar precedido por un tab)
```

donde objetivo es el archivo que será creado o actualizado cuando alguna dependencia (archivos fuente, objetos, etc.) sea modificada. Para realizar esta tarea se ejecutará el comando de compilación especificado (nótese que el comando de compilación debe estar indentado con un carácter de tabulación).

### 11. 6.1 Herramienta `make`.

Una vez creado el `makefile` es posible ejecutar el comando `make` para llevar a cabo la compilación del proyecto. Esta herramienta buscará en el directorio de trabajo un archivo con nombre `Makefile`, lo procesará para obtener las reglas de dependencias y demás información y, por último, ejecutará la compilación de todos los archivos y el enlace final para generar la aplicación ejecutable.

### 11.6.2 Estructura del archivo `makefile`.

Con un poco más de detalle, se puede ver que un archivo `makefile` contiene:

- Bloques de descripción.
- Comandos.
- Macros.
- Reglas de inferencia.
- Directivas '.' (punto).
- Directivas de preprocesamiento.

Y otros componentes como caracteres comodines, nombres largos de archivos, comentarios y caracteres especiales.

### 11.6.3 Bloques de descripción.

Un bloque de descripción es una línea donde se expresan las dependencias correspondientes a cada objetivo:

objetivo: dependencias (archivos fuentes, objetos)

### 11.6.4 Comandos.

Un bloque de descripción o una regla de inferencia especifica un bloque de comandos para ejecutarse si la dependencia está desactualizada. Un bloque de comandos contiene uno o más comandos, cada uno en su propia línea. Una línea de comandos comienza con un carácter de tabulación. A continuación se muestra un ejemplo donde la primera línea es el bloque de descripción y la segunda, el bloque de comandos:

```
mi_aplicacion: mi_aplicacion.o mis_funciones.o
  $(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o
```

### 11.6.5 Macros.

Las macros son definiciones que tienen un nombre y un valor asociado, y que pueden utilizarse a lo largo del `makefile`. El formato es:

```
<nombre macro>=<valor>
```

Algunos ejemplos:

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
```

La herramienta `make` reemplazará cada ocurrencia de `$(CC)` y `$(CFLAGS)` por el valor asociado. También es posible especificar el valor de la macro cuando se ejecuta `make`:

```
make 'CC=gcc'
```

Existen macros utilizadas internamente por la herramienta `make`, algunas de las cuales se muestran a continuación:

`CC:` Contiene el compilador C utilizado en el desarrollo. El default es `cc`.

`$@:` El nombre completo del archivo destino.

`$<:` El archivo fuente de la dependencia actual.

#### 11. 6. 6 Reglas de inferencia.

Las reglas de inferencia proveen comandos para generar los archivos objetivo infiriendo sus dependencias (evita la necesidad de tener que especificar una regla por cada archivo que se ha de compilar). En el ejemplo siguiente se establece una regla de inferencia que determina que para generar un archivo objeto `.obj` (en ocasiones puede usarse la extensión `.o` en lugar de `.obj`, como se pudo apreciar en ejemplos anteriores) es necesario que exista un archivo fuente `.c` con igual nombre:

```
%.obj : %.c
    $(CC) $(CFLAGS) -c $(.SOURCE)
```

Esta regla de inferencia le indica a `make` cómo construir un archivo `.obj` a partir de un archivo `.c`. La macro predefinida `.SOURCE` es el nombre del archivo de dependencia inferido. A continuación, se presenta el archivo `makefile` expuesto antes, pero ahora utilizando reglas de inferencia:

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic

mi_aplicacion: mi_aplicacion.o mis_funciones.o
  $(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o

%.obj : %.c
  $(CC) $(CFLAGS) -c $(.SOURCE)

clean:
  rm *.o mi_aplicacion
```

En la sección de problemas correspondientes a este capítulo se podrán encontrar otros casos, un poco más elaborados, sobre el uso de `makefiles`. Aquí sólo se pretende dar una introducción básica y que el lector se familiarice con el uso de estas herramientas.

## 11.7 Resumen.

Luego de estudiar los conceptos fundamentales referidos a la construcción de algoritmos, en el último capítulo se trataron las cuestiones relacionadas con el proceso de compilación, que permite generar un programa ejecutable. Las etapas más importantes que pueden distinguirse en este proceso son el preprocesamiento, la compilación y el enlace.

El preprocesamiento, como su nombre lo indica, es una etapa previa que tiene como finalidad "acomodar" el código fuente antes de que éste sea procesado por el compilador. Para ese fin, el preprocesador modifica el código fuente según un conjunto de directivas que el programador puede incluir en distintos puntos del programa. La compilación condicional es un buen ejemplo: Si se cumple cierta condición, el preprocesador podría eliminar porciones del código del programa.

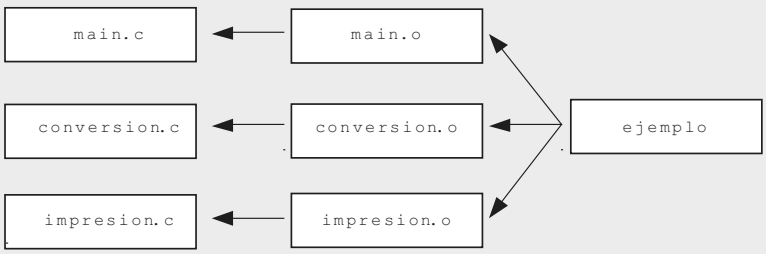
Durante la compilación se traduce el lenguaje de alto nivel en lenguaje de máquina. Dentro de esta etapa pueden distinguirse las siguientes fases: Análisis léxico, análisis sintáctico, análisis semántico y generación de código.

La última de las etapas, el enlace, se encarga de tomar el código de bibliotecas externas a las que nuestro programa podría estar haciendo referencia, y generar (a partir de nuestro código y el de las bibliotecas externas) el programa ejecutable.

En la práctica, nuestros proyectos suelen estar conformados por varios archivos que, luego del proceso de compilación, darán lugar a una aplicación ejecutable. A medida que estos proyectos crecen, en cuanto a cantidad de archivos, la compilación puede tornarse engorrosa. Felizmente, existe la posibilidad de automatizar el proceso de compilación mediante el uso de la herramienta `make`. Ésta es capaz de distinguir dentro de un proyecto cuáles son los archivos fuente que han sido alterados, y evitar la compilación innecesaria de los que no se modificaron. Make construye proyectos sobre la base de comandos contenidos en un archivo de descripción o archivo de dependencias, comúnmente denominado `makefile`.

## 11.8 Problemas resueltos.

1- Esta aplicación permite convertir kilómetros en millas. El programa principal se encuentra en el archivo `main.c`. La conversión se lleva a cabo llamando a la función `convertir()` en `conversion.c` (y cuyo prototipo está definido en el archivo de cabecera `conversion.h`). Por último, para imprimir el resultado se invoca a la función `imprimir_resultado()` en `impresion.c` (y cuyo prototipo está definido en el archivo de cabecera `impresion.h`). El grafo de dependencias es el siguiente:






En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

El `makefile` se construyó usando reglas de inferencia, aprovechando que los archivos de dependencias (`main.c, conversion.c` e `impresion.c`) y los objetivos (`main.o, conversion.o` e `impresion.o`) tienen el mismo nombre.

Obsérvese que en el `makefile` se incluye una regla adicional cuyo objetivo es `clean`. Esta regla permite borrar todos los archivos generados en la compilación del proyecto, mediante el comando: `make clean`. Como puede verse, esta regla no tiene dependencia alguna y su comando asociado es `rm *.o ejemplo` (para eliminar todos los archivos con extensión `.o` y el ejecutable `ejemplo`).

El comando `rm` (`remove`) permite eliminar archivos en ambientes Unix/Linux; en el caso de Windows el comando equivalente es `del/`.

**main.c**

```
#include <stdio.h>
#include "conversion.h"
#include "impresion.h"

int main (void)
{
  int kms;
  float millas;

  printf("Conversión de kilómetros a millas\n");
  printf("---------------------------------\n\n");

  printf("Introduzca la cant. de kms a convertir: ");
  scanf("%i", &kms);

  millas = convertir(kms);
  imprimir_resultado(kms, millas);

  return 0;
}
```

**conversion.h**

```
#define FACTOR 0.6214

/* Prototipos */
float convertir(int);
```

**conversion.c**

```
#include "conversion.h"

float convertir(int kms)
{
  return FACTOR * kms;
}
```

**impresion.h**

```
/* Prototipos */
void imprimir_resultado(int,float);
```

**impresion.c**

```
#include <stdio.h>

void imprimir_resultado(int km, float millas)
{
  printf("%d km equivalen a %.2f millas", km, millas);
}
```

**Makefile**

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic

ejemplo: main.o conversion.o impresion.o
  $(CC) -o ejemplo main.o conversion.o impresion.o

# REGLA DE INFERENCIA
# Esta forma es equivalente a:  %.o : %.c
# La macro '$<' es el archivo fuente del cual el objetivo
# depende (por ejemplo, conversion.c).
.c.o:
  $(CC) $(CFLAGS) -c $<

clean:
  rm *.o ejemplo
```

2- Hasta el momento en todos los casos se asumió que los archivos fuente y de cabecera residen en el mismo directorio y que también los archivos generados (objetos y binario ejecutable) se ubicarán allí. Sin embargo, a medida que un proyecto crece en cuanto a cantidad de archivos, se hace necesario organizarlos en subdirectorios. En un proyecto típico podemos encontrar:

| | |
|---|---|
| `src/` | Contiene los archivos fuente (`.c`). |
| `include/` | Contiene los archivos de cabecera (`.h`). |
| `bin/` | Contiene el binario ejecutable generado. |

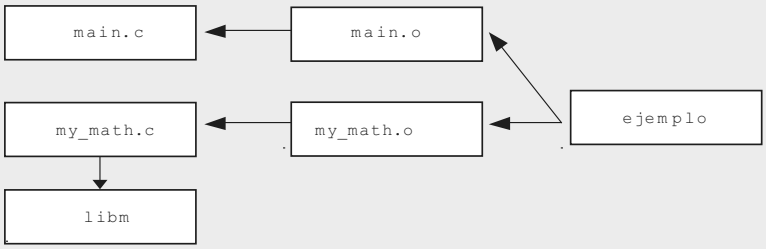
Tomando como caso el mismo proyecto que en el problema anterior, se muestra el archivo `makefile` utilizando la estructura de subdirectorios descripta.

**Makefile**

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic

SRC_DIR=src
INCLUDE_DIR=include
BIN_DIR=bin

VPATH=$(SRC_DIR)

ejemplo: main.o conversion.o impresion.o
  $(CC) -o ejemplo main.o conversion.o impresion.o
  rm *.o
  mv ejemplo $(BIN_DIR)

.c.o:
  $(CC) $(CFLAGS) -I$(INCLUDE_DIR) $< -c

clean:
  rm $(BIN_DIR)/ejemplo
```

La forma que tenemos de indicarle a `make` dónde buscar los archivos de dependencias (`.c`) es utilizar la macro `VPATH`. En nuestro caso, `VPATH` toma el valor `src` (o sea que los archivos fuente `.c` deberán buscarse en el subdirectorio `src/`). Además, también es necesario indicar al compilador dónde buscar los archivos de cabecera (ya que ahora no se encuentran en el mismo directorio que los archivos fuente, sino en el subdirectorio `include/`). Para ese fin se define la macro `INCLUDE_DIR` y se la utiliza con la opción `-I` del compilador. Por último, el binario ejecutable generado (`ejemplo`) se mueve al subdirectorio `bin/` usando el comando `mv` y todos los archivos objeto generados se eliminan (esto suele ser una práctica habitual cuando se hace un lanzamiento del proyecto, ya que el usuario final no necesita la existencia de los archivos objeto).

En este caso, una aplicación solicita al usuario dos valores enteros: Base y exponente, y muestra en pantalla el resultado $base^{exponente}$. La operación se realiza llamando a la función `potencia()` en el archivo `my_math.c`. Por su parte, esta función llama a `pow()`, que se encuentra implementada en la biblioteca matemática estándar de C (`libm`). El grafo de dependencias es el siguiente:




**main.c**

```
#include <stdio.h>
#include "my_math.h"

int main()
{
  int base, exponente;

  printf("Ingrese la base: ");
  scanf("%d", &base);
  printf("Ingrese el exponente: ");
  scanf("%d", &exponente);

  printf("%d elevado a la %d = %d\n", base, exponente, potencia(base,exponente));

  return 0;
}
```

my_math.h

```
/* Prototipo */
int potencia(int base, int exponente);
```

my_math.c

```
#include <math.h>
```

```
int potencia(int base, int exponente)
{
   return (int)pow(base,exponente);
}
```

**Makefile**

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
LIBS=-lm

ejemplo: main.o my_math.o
  $(CC) $(LIBS) -o ejemplo main.o my_math.o

math.o: math.c
  $(CC) $(CFLAGS) -c math.c

clean:
  rm *.o ejemplo
```

En este programa se hace uso de una biblioteca externa (`libm`), en la que se encuentra la implementación de la función `pow()`. Para indicar al compilador que el binario ejecutable debe enlazarse contra una biblioteca externa, se utiliza la opción `-l`. Además, se debe añadir el nombre de la biblioteca en cuestión, para lo cual se toma el nombre de ella y se quita el prefijo `lib`. Para el caso de `libm`, debe usarse: `-lm`. Para que esto quede más claro, en el archivo `makefile` se definió la macro:

```
LIBS=-lm
```

## 11.9 Contenido de la página Web de apoyo.

El material marcado con asterisco (*) sólo está disponible para docentes.

**Mapa conceptual.**

**Simulación:**

- El proceso de compilación.

**Autoevaluación.**

**Video explicativo** (02:21 minutos aprox.)**.**

**Código fuente de los ejercicios resueltos.**

**Evaluaciones propuestas. ***

**Presentaciones. ***

## Bibliografía

Aho, Alfred V., Ullman, Jeffrey D. y Hopcroft, John E. - *Data Structures and Algorithms* - Addison Wesley, EE.UU, 1983.

Aho, Alfred V., Hopcroft, John E. y Ullman, Jeffrey D. - *The Design and Analysis of Computer Algorithms* - Addison-Wesley, EE.UU 1974.

Echevarría, Adriana y López, Carlos Gustavo - *Elementos de Diseño y Programación con Ejemplos en C* - Nueva Librería, Argentina, 2005.

Garey, Michael R. y Johnson, David S. - *Computers and Intractability: a Guide to the Theory of NP-Completeness* - W. H. Freeman, EE.UU, 1979.

Kernighan, Brian W. y Ritchie, Dennis M. - *El Lenguaje de programación C*, 2ª ed. - Prentice Hall, Mexico, 1991.

Knuth, Donald . - *The Art of Computer Programming*, 2ª ed. - Addison-Wesley Professional, EE.UU, 1998.

Lewis, Harry R. y Papadimitriou , Christos.H (1978) - *The Efficiency of Algorithms- Scientific American*, páginas 96 a 108, enero de 1978.

López, Carlos Gustavo - *Algoritmia, Arquitectura de Datos y Programación Estructurada* - Nueva Librería, Argentina, 2003.

Papadimitriou , Christos.H. & Steiglitz, Kenneth. - *Combinatorial Optimization : Algorithms and Complexity* - Prentice-Hall, EE.UU, 1982.

Rheingold, Edward M., Nievergelt, Jurg, Deo, Narsingh - *Combinatorial Algorithms: Theory and Practice* - Prentice-Hall, EE.UU, 1982.

## Índice analítico