Práctica 1: Conceptos de Generación de Código. El lenguaje NASM.

Fecha de entrega:
Antes de las sesiones del 13, 14, 18 y 19 de octubre de cada grupo

Objetivo de la práctica:

El objetivo de la práctica es introducir la etapa de Generación de Código y escribir una librería de funciones C para la generación de código NASM. Esto ayudará al estudiante a familiarizarse con el lenguaje en el que se generará el código objeto en el compilador y agilizará la solución de la generación de código del compilador en la fase final del desarrollo del mismo.

Desarrollo de la práctica:

0. Decisiones de diseño para la generación de código
Se resumen a continuación las decisiones de diseño más importantes que tu profesor te explicará a lo largo del laboratorio respecto a la generación de código. En general son sugerencias que tú puedes seguir; si adoptaras otras decisiones, deberías consensuarlas con el profesor y asegurarte de que se cumplen las condiciones que en cada momento se describan tanto de funcionalidad como de interfaz (nombre de funciones, tipo, número y posición de argumentos, etc.)

Tamaño y tipo de datos en memoria
La aritmética de nuestro compilador se reducirá siempre al tipo de dato entero y a tamaño de 32 bits.
Eso implica que el conjunto de registros sea el extendido, así por ejemplo el acumulador es `eax`, el puntero de pila es `esp`, etc...
Los tipos de datos básicos de nuestro lenguaje de programación son entero y booleano (que será internamente manipulado como entero).

Pila como almacenamiento auxiliar
Cuando se necesite espacio auxiliar (por ejemplo para gestionar las expresiones aritmético-lógicas) se utilizará la pila del sistema.
La manera en la que se gestionarán las **expresiones aritmético-lógicas** será la siguiente:

- Se supondrá que los operandos están siempre en la cima de la pila (dos posiciones contiguas en el caso de operaciones binarias y una en monádicas).
- Por lo tanto, el proceso de cualquier operación implicará:
    - Introducir, cuando se disponga de ellos, los operandos en la pila.
    - Cuando se sepa qué operación realizar:
        - Extraer de la cima de la pila los operandos guardándolos en registros,
        - realizar la operación y

        - copiar el resultado desde el registro que lo tenga a la pila.
    - Cada operación tiene sus peculiaridades y sus códigos de operación NASM que debes consultar en la documentación que se te proporciona al respecto.

Este esquema también determina cómo gestionar las **asignaciones de valor a las variables de alto nivel**.

Por ejemplo `x=4+y;`

- Como se ha indicado previamente, tras gestionar una expresión, su resultado se deja en la cima de la pila.
- Por lo tanto, dado que la asignación considera que el valor que se asigna a la variable está expresado mediante una expresión aritmético-lógica, en las asignaciones se toma el valor que hay que asignar a la variable de la cima de la pila.

Operaciones de entrada salida

Tu profesor te explicará las funciones contenidas en la librería *alfalib.o* que se te proporciona como material del laboratorio.

También te explicará cómo se invocan.

Esta librería contiene, entre otras, todas las operaciones de lectura y escritura que puedes necesitar invocar en NASM cuando generes código.

Tu profesor te explicará con detalle cómo se invocan funciones en ensamblador y el convenio de llamada que usamos y que es el utilizado en la librería.

Nombres internos correspondientes a los símbolos de alto nivel

En general se precederá del carácter "_" los nombres de alto nivel para ser convertidos en nombres correctos NASM.

En algunos casos (como la variable para guardar el puntero de pila) se utilizarán "_" adicionales para asegurar que son nombres correctos (en ese caso se te recomienda que utilices un nombre como `__esp`).

Información semántica que mantiene el compilador.

A lo largo del proceso de compilación, se mantiene cierta información que facilita tanto el análisis semántico como la generación de código.

Una de esas informaciones sirve para distinguir si lo que hay en la cima de la pila (cuando se está procesando una expresión aritmético-lógica) es una variable (se suele llamar referencia) o no (es un valor explícito). No es lo mismo encontrar en el programa fuente un `3` que un `b1`, cuando se introduce este último operando en la pila por primera vez se introduce como "`_b1`" y el compilador debe recordar que es referencia (es decir, que en la pila no está el valor del operando sino una referencia al lugar donde está, en este caso la posición de memoria `_b1`), sin embargo, el operando `3` es ya un valor explícito.

Gestión de las etiquetas

A lo largo de un programa NASM hay múltiples etiquetas necesarias para implementar las estructuras de control de flujo del programa.

Cuando el compilador genere código deberá articular algún mecanismo para poder distinguir unas etiquetas de otras. A lo largo del curso se sugiere el uso de un contador que se incremente cada vez que se utilice una etiqueta y que aparezca explícitamente en ella. Así, por ejemplo, una etiqueta de fin de `if` podría ser `fin_if_99:`, siendo en este caso 99 el correspondiente contador. Para la siguiente etiqueta que se necesite (por ejemplo una etiqueta para una rama `then`) se incrementará el contador de etiquetas y se generará, por ejemplo, la etiqueta `then_100:`.

Las funciones que necesitan utilizar una etiqueta en su generación de código tienen que tener en cuenta estas reflexiones.
Esta librería puede utilizarse tanto desde el compilador (verás en el futuro cómo se pueden incluir en las acciones del fichero de entrada de una de las herramientas que utilizaremos, en concreto en el fichero alfa.y) o bien en un programa principal independiente del compilador.
En el primer caso, el mecanismo de propagación de información (verás que utilizaremos variables globales y síntesis) junto con el propio proceso de análisis realiza la tarea que se va a describir a continuación.
La generación de código de un *ifthenelse* o de un *while* implica el uso de un conjunto de etiquetas que está relacionado (salto al final del bucle cuando se cumple la condición de salida, salto al inicio en otro caso)
Como los programas pueden tener múltiples *whiles* o *ifthenelses*, es necesario llevar un contador de cuántas etiquetas se han usado para poder distinguir unas de otras.
En los casos en los que las estructuras de control de flujo (bloques estructurales) se anidan se da el hecho de que a medida que se profundiza en el anidamiento hay que ir conservando las etiquetas de los bloques de los que aún no se ha salido porque cuando se vaya saliendo de ellos habrá que recuperarlas para realizar esa gestión.
Si se utiliza esta librería de manera independiente al compilador, hay que programar este control de etiquetas con estructuras de datos propias del programa principal. Una aproximación sencilla es una pila, ya que el orden de acceso a sus elementos coincide con el orden de acceso a las sucesivas etiquetas.
En el laboratorio tu profesor te explicará con detenimiento una posible solución.

1. Librería de Funciones de Generación de Código.

Partiendo de la información aportada por el profesor y accesible en el curso de *Moodle* de la asignatura, se van a desarrollar una serie de funciones en lenguaje C, que irán generando código NASM. Se generará una librería "generacion.c" y su correspondiente fichero de encabezados "generacion.h" en los que se van a codificar una serie de funciones que resolverán parcialmente la generación de código de la parte procedural. Todas las funciones recibirán como parámetro, entre otros, el fichero en el que se debe escribir.
Más adelante, en este curso se trabajará en el proceso completo de generación de código. A continuación se enumera la funcionalidad que tu librería debe cubrir.

- **Inicialización y Finalización de Programas NASM**: Todos los programas generados comenzarán y finalizarán de la misma manera y seguirán la estructura descrita en el material de la asignatura. El inicio y fin de programa implica las siguientes funciones:
    - Inicializar (debe subdividirse en varias funciones): Se trata de una serie de rutinas que escribirán la parte inicial de la estructura de un programa NASM. Esto es:
        - Cabecera de compatibilidad (si se usa).
        - Segmento "bss".
        - Segmento "data".
        - Sección de código (Segmento "text").
            - Tras la sección de código se mostrarían las funciones del programa fuente de partida, la generación de su código está fuera del alcance de esta práctica.
        - Etiqueta "main" y acciones iniciales del programa.

    - Finalizar: La finalización de programas debe contemplar la posibilidad de que haya habido algún error controlado de ejecución.

- **Operaciones Aritméticas y Lógicas**: con la recepción correspondiente de parámetros, según lo explicado en clase, se codificarán las funciones necesarias para la realización de:
    - Suma
    - Cambio de signo
    - Resta
    - Multiplicación
    - División
    - Negación (lógica)
    - O (disyunción lógica)
    - Y (conjunción lógica)

- **Comparaciones**: con la recepción correspondiente de parámetros, según lo explicado en clase, se codificarán las funciones necesarias para la realización de las siguientes comparaciones:
    - Igualdad
    - Desigualdad
    - Menor o igual
    - Mayor o igual
    - Menor
    - Mayor

- Funciones **auxiliares para la manipulación de los operandos** para las operaciones aritmético-lógicas.
- Funciones para la **declaración de variables.**
- Funciones de **entrada / salida.**
- Funciones para **asignación de expresiones a variables.**

2 Cabeceras de las funciones pedidas

A continuación se muestran las cabeceras que debes utilizar.

Es imprescindible mantener este convenio de funciones porque es un requisito de diseño que se pueda utilizar la librería con cualquier programa principal que cumpla el convenio.

En cada función tienes un comentario que resume lo que debe hacer. Puedes encontrar con más detalle esta misma información en el material dedicado a la generación de código.

```
#ifndef GENERACION_H
#define GENERACION_H

#include <stdio.h>


/* Declaraciones de tipos de datos del compilador */
#define ENTERO      0
#define BOOLEANO    1


/* OBSERVACIÓN GENERAL A TODAS LAS FUNCIONES:
```

```
   Todas ellas escriben el código NASM a un FILE* proporcionado como primer
argumento.
*/

void escribir_cabecera_bss(FILE* fpasm);
/*
   Código para el principio de la sección .bss.
   Con seguridad sabes que deberás reservar una variable entera para guardar el
puntero de pila extendido (esp). Se te sugiere el nombre __esp para esta variable.
*/

void escribir_subseccion_data(FILE* fpasm);
/*
   Declaración (con directiva db) de las variables que contienen el texto de los
mensajes para la identificación de errores en tiempo de ejecución.
   En este punto, al menos, debes ser capaz de detectar la división por 0.
*/

void declarar_variable(FILE* fpasm, char * nombre,  int tipo,  int tamano);
/*
   Para ser invocada en la sección .bss cada vez que se quiera declarar una
variable:
   - El argumento nombre es el de la variable.
   - tipo puede ser ENTERO o BOOLEANO (observa la declaración de las constantes
     del principio del fichero).
   - Esta misma función se invocará cuando en el compilador se declaren
     vectores, por eso se adjunta un argumento final (tamano) que para esta
     primera práctica siempre recibirá el valor 1.
*/

void escribir_segmento_codigo(FILE* fpasm);
/*
   Para escribir el comienzo del segmento .text, básicamente se indica que se
exporta la etiqueta main y que se usarán las funciones declaradas en la librería
alfalib.o
*/

void escribir_inicio_main(FILE* fpasm);
/*
   En este punto se debe escribir, al menos, la etiqueta main y la sentencia que
guarda el puntero de pila en su variable (se recomienda usar __esp).
*/

void escribir_fin(FILE* fpasm);
/*
   Al final del programa se escribe:
   - El código NASM para salir de manera controlada cuando se detecta un error
     en tiempo de ejecución (cada error saltará a una etiqueta situada en esta
     zona en la que se imprimirá el correspondiente mensaje y se saltará a la
     zona de finalización del programa).
   - En el final del programa se debe:
        ·Restaurar el valor del puntero de pila (a partir de su variable __esp)
        ·Salir del programa (ret).
*/

void escribir_operando(FILE* fpasm, char* nombre, int es_variable);
/*
   Función que debe ser invocada cuando se sabe un operando de una operación
aritmético-lógica y se necesita introducirlo en la pila.
   - nombre es la cadena de caracteres del operando tal y como debería aparecer
     en el fuente NASM
   - es_variable indica si este operando es una variable (como por ejemplo b1)
     con un 1 u otra cosa (como por ejemplo 34) con un 0. Recuerda que en el
```

```
      primer caso internamente se representará como _b1 y, sin embargo, en el
      segundo se representará tal y como esté en el argumento (34).
*/

void asignar(FILE* fpasm, char* nombre, int es_variable);
/*
   - Genera el código para asignar valor a la variable de nombre nombre.
   - Se toma el valor de la cima de la pila.
   - El último argumento es el que indica si lo que hay en la cima de la pila es
     una referencia (1) o ya un valor explícito (0).
*/


/* FUNCIONES ARITMÉTICO-LÓGICAS BINARIAS */
/*
   En todas ellas se realiza la operación como se ha resumido anteriormente:
   - Se extrae de la pila los operandos
   - Se realiza la operación
   - Se guarda el resultado en la pila
   Los dos últimos argumentos indican respectivamente si lo que hay en la pila es
una referencia a un valor o un valor explícito.
   Deben tenerse en cuenta las peculiaridades de cada operación. En este sentido
sí hay que mencionar explícitamente que, en el caso de la división, se debe
controlar si el divisor es "0" y en ese caso se debe saltar a la rutina de error
controlado (restaurando el puntero de pila en ese caso y comprobando en el retorno
que no se produce "Segmentation Fault")
*/
void sumar(FILE* fpasm, int es_variable_1, int es_variable_2);
void restar(FILE* fpasm, int es_variable_1, int es_variable_2);
void multiplicar(FILE* fpasm, int es_variable_1, int es_variable_2);
void dividir(FILE* fpasm, int es_variable_1, int es_variable_2);
void o(FILE* fpasm, int es_variable_1, int es_variable_2);
void y(FILE* fpasm, int es_variable_1, int es_variable_2);

void cambiar_signo(FILE* fpasm, int es_variable);
/*
   Función aritmética de cambio de signo.
   Es análoga a las binarias, excepto que sólo requiere de un acceso a la pila ya
que sólo usa un operando.
*/

void no(FILE* fpasm, int es_variable, int cuantos_no);
/*
   Función monádica lógica de negación. No hay un código de operación de la ALU
que realice esta operación por lo que se debe codificar un algoritmo que, si
encuentra en la cima de la pila un 0 deja en la cima un 1 y al contrario.
   El último argumento es el valor de etiqueta que corresponde (sin lugar a dudas,
la implementación del algoritmo requerirá etiquetas). Véase en los ejemplos de
programa principal como puede gestionarse el número de etiquetas cuantos_no.
*/

/* FUNCIONES COMPARATIVAS */
/*
   Todas estas funciones reciben como argumento si los elementos a comparar son o
no variables. El resultado de las operaciones, que siempre será un booleano ("1"
si se cumple la comparación y "0" si no se cumple), se deja en la pila como en el
resto de operaciones. Se deben usar etiquetas para poder gestionar los saltos
necesarios para implementar las comparaciones.
*/
void igual(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);
void distinto(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);
void menor_igual(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);
void mayor_igual(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);
```

```
void menor(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);
void mayor(FILE* fpasm, int es_variable1, int es_variable2, int etiqueta);

/* FUNCIONES DE ESCRITURA Y LECTURA */
/*
   Se necesita saber el tipo de datos que se va a procesar (ENTERO o BOOLEANO) ya
que hay diferentes funciones de librería para la lectura (idem. escritura) de cada
tipo.
   Se deben insertar en la pila los argumentos necesarios, realizar la llamada
(call) a la función de librería correspondiente y limpiar la pila.
*/
void leer(FILE* fpasm, char* nombre, int tipo);
void escribir(FILE* fpasm, int es_variable, int tipo);

#endif
```

En esta segunda parte de la práctica, también vamos a codificar las siguientes funciones que generan el código para manipular vectores, funciones, bucles (*while*) y estructuras condicionales (*if-else*). El material correspondiente a la teoría sobre cómo implementar estas funciones está colgado en Moodle y será explicado por vuestro profesor. Al final del enunciado de la práctica se os proporcionan 3 ejemplos donde se detalla en qué punto se invoca a cada una de estas funciones con respecto a un código escrito en lenguaje alfa que se está procesando y cuál es la salida de estos programas. Recuerda que mientras no podamos explicar cuál es la gramática de este lenguaje de programación, sólo puedes analizar ejemplos como los que te mostramos en este enunciado.

A continuación tienes la descripción de las nuevas funciones

```
void ifthenelse_inicio(FILE * fpasm, int exp_es_variable, int etiqueta)
```

- Generación de código para el inicio de una estructura if-then-else
- Como es el inicio de uno bloque de control de flujo de programa que requiere de una nueva etiqueta deben ejecutarse antes las tareas correspondientes a esta situación
- exp_es_variable
    - Es 1 si la expresión de la condición es algo asimilable a una variable (identificador, elemento de vector)
    - Es 0 en caso contrario (constante u otro tipo de expresión)

```
void ifthen_inicio(FILE * fpasm, int exp_es_variable, int etiqueta)
```

- Generación de código para el inicio de una estructura if-then
- Como es el inicio de uno bloque de control de flujo de programa que requiere de una nueva etiqueta deben ejecutarse antes las tareas correspondientes a esta situación
- exp_es_variable
    - Es 1 si la expresión de la condición es algo asimilable a una variable (identificador, elemento de vector)
    - Es 0 en caso contrario (constante u otro tipo de expresión)

```
void ifthen_fin(FILE * fpasm, int etiqueta)
```

- Generación de código para el fin de una estructura if-then
- Como es el fin de uno bloque de control de flujo de programa que hace uso de la etiqueta del

mismo se requiere que antes de su invocación tome el valor de la etiqueta que le toca según se ha explicado
- Y tras ser invocada debe realizar el proceso para ajustar la información de las etiquetas puesto que se ha liberado la última de ellas.

```
void ifthenelse_fin_then( FILE * fpasm, int etiqueta)
```

- Generación de código para el fin de la rama then de una estructura if-then-else
- Sólo necesita usar la etiqueta adecuada, aunque es el final de una rama, luego debe venir otra (la rama else) antes de que se termine la estructura y se tenga que ajustar las etiquetas por lo que sólo se necesita que se utilice la etiqueta que corresponde al momento actual.

```
void ifthenelse_fin( FILE * fpasm, int etiqueta)
```

- Generación de código para el fin de una estructura if-then-else
- Como es el fin de uno bloque de control de flujo de programa que hace uso de la etiqueta del mismo se requiere que antes de su invocación tome el valor de la etiqueta que le toca según se ha explicado
- Y tras ser invocada debe realizar el proceso para ajustar la información de las etiquetas puesto que se ha liberado la última de ellas.

```
void while_inicio(FILE * fpasm, int etiqueta)
```

- Generación de código para el inicio de una estructura while
- Como es el inicio de uno bloque de control de flujo de programa que requiere de una nueva etiqueta deben ejecutarse antes las tareas correspondientes a esta situación
- exp_es_variable
    - Es 1 si la expresión de la condición es algo asimilable a una variable (identificador, elemento de vector)
    - Es 0 en caso contrario (constante u otro tipo de expresión)

```
void while_exp_pila (FILE * fpasm, int exp_es_variable, int etiqueta)
```

- Generación de código para el momento en el que se ha generado el código de la expresión de control del bucle
- Sólo necesita usar la etiqueta adecuada, por lo que sólo se necesita que se recupere el valor de la etiqueta que corresponde al momento actual.

- exp_es_variable
    - Es 1 si la expresión de la condición es algo asimilable a una variable (identificador, o elemento de vector)
    - Es 0 en caso contrario (constante u otro tipo de expresión)

```
void while_fin( FILE * fpasm, int etiqueta)
```

- Generación de código para el final de una estructura while
- Como es el fin de uno bloque de control de flujo de programa que hace uso de la etiqueta del mismo se requiere que antes de su invocación tome el valor de la etiqueta que le toca según se ha explicado
- Y tras ser invocada debe realizar el proceso para ajustar la información de las etiquetas puesto que se ha liberado la última de ellas.

```
void escribir_elemento_vector(FILE * fpasm,char * nombre_vector,
                  int tam_max, int exp_es_direccion)
```

- Generación de código para indexar un vector
    - Cuyo nombre es nombre_vector
    - Declarado con un tamaño tam_max
    - La expresión que lo indexa está en la cima de la pila
    - Puede ser una variable (o algo equivalente) en cuyo caso exp_es_direccion vale 1
    - Puede ser un valor concreto (en ese caso exp_es_direccion vale 0)
- Según se especifica en el material, es suficiente con utilizar dos registros para realizar esta tarea.

```
void declararFuncion(FILE * fd_asm, char * nombre_funcion, int num_var_loc)
```

- Generación de código para iniciar la declaración de una función.
- Es necesario proporcionar
    - Su nombre
    - Su número de variables locales

```
void retornarFuncion(FILE * fd_asm, int es_variable)
```

- Generación de código para el retorno de una función.
    - La expresión que se retorna está en la cima de la pila.
    - Puede ser una variable (o algo equivalente) en cuyo caso exp_es_direccion vale 1
    - Puede ser un valor concreto (en ese caso exp_es_direccion vale 0)

```
void escribirParametro(FILE* fpasm, int pos_parametro, int num_total_parametros)
```

- Función para dejar en la cima de la pila la dirección efectiva del parámetro que ocupa la posición pos_parametro (recuerda que los parámetros se ordenan con origen 0) de un total de num_total_parametros

```
void escribirVariableLocal(FILE* fpasm, int posicion_variable_local)
```

- Función para dejar en la cima de la pila la dirección efectiva de la variable local que ocupa la posición posicion_variable_local (recuerda que ordenadas con origen 1)

```
void asignarDestinoEnPila(FILE* fpasm, int es_variable)
```

- Función para poder asignar a un destino que no es una variable "global" (tipo `_x`) por ejemplo parámetros o variables locales (ya que en ese caso su nombre real de alto nivel, no se tiene en cuenta pues es realmente un desplazamiento a partir de `ebp`: `ebp+4` o `ebp-8` por ejemplo).
- Se debe asumir que en la pila estará
    - Primero (en la cima) la dirección donde hay que asignar
    - Debajo (se ha introducido en la pila antes) lo que hay que asignar
- es_variable
    - Es 1 si la expresión que se va a asignar es algo asimilable a una variable (identificador, o elemento de vector)
    - Es 0 en caso contrario (constante u otro tipo de expresión)

```
void operandoEnPilaAArgumento(FILE * fd_asm, int es_variable)
```

- Como habrás visto en el material, nuestro convenio de llamadas a las funciones asume que los argumentos se pasan por valor, esto significa que siempre se dejan en la pila "valores" y no "variables"
- Esta función realiza la tarea de dado un operando escrito en la pila y sabiendo si es variable o no (es_variable) se deja en la pila el valor correspondiente

```
void llamarFuncion(FILE * fd_asm, char * nombre_funcion, int num_argumentos)
```

- Esta función genera código para llamar a la función nombre_funcion asumiendo que los argumentos están en la pila en el orden fijado en el material de la asignatura.
- Debe dejar en la cima de la pila el retorno de la función tras haberla limpiado de sus argumentos
- Para limpiar la pila puede utilizar la función de nombre limpiarPila

```
void limpiarPila(FILE * fd_asm, int num_argumentos)
```

- Genera código para limpiar la pila tras invocar una función
- Esta función es necesaria para completar la llamada a métodos, su gestión dificulta el conocimiento por parte de la función de llamada del número de argumentos que hay en la pila

3. Ejemplos.

A continuación se muestra un ejemplo de un programa principal que realiza lo mismo que el siguiente programa de alto nivel.

Observa que el programa principal toma como argumento al ser invocado el nombre del fichero que contendrá la salida.
El programa también declara variables de tipo entero y realiza operaciones aritméticas con ellas antes de mostrar resultados.

3.1 Ejemplo 1

Código fuente alto nivel para ejemplo 1

Este primer ejemplo declara variables enteras y realiza operaciones con ellas, mostrando por pantalla el resultado.

```
main
{
	int x;
	int y;
	int z;

	x=8;
	scanf y;
	z = x + y;
	printf z;
	 z = 7 + y;
	printf z;
}
```

Posible programa principal para ejemplo 1

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

int main (int argc, char** argv)
{
	FILE* salida;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}

	salida = fopen(argv[1],"w");

	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
	declarar_variable(salida, "x", ENTERO, 1);
	declarar_variable(salida, "y", ENTERO, 1);
	declarar_variable(salida, "z", ENTERO, 1);

	escribir_segmento_codigo(salida);
	escribir_inicio_main(salida);

	/* x=8; */
	escribir_operando(salida,"8",0);
	asignar(salida,"x",0);

	/* scanf(&y); */
	leer(salida,"y",ENTERO);

	/* z = x + y */
	escribir_operando(salida,"x",1);
	escribir_operando(salida,"y",1);
	sumar(salida,1,1);
	asignar(salida,"z",0);

	/* printf(z); */
	escribir_operando(salida,"z",1);
	escribir(salida,1,ENTERO);

	/* z = 7 + y */
	escribir_operando(salida,"7",0);
	escribir_operando(salida,"y",1);
	sumar(salida,0,1);
	asignar(salida,"z",0);

	/* printf(z); */
	escribir_operando(salida,"z",1);
	escribir(salida,1,ENTERO);

	escribir_fin(salida);

	fclose(salida);
	return 0;
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej1`.

```
> ej1   /* La entrada es -9 y la salida -1 -2*/
-9
```

```
-1
-2

> ej1 /* La entrada es 10 y la salida 18 17*/
10
18
17
```

3.2 Ejemplo 2

Código fuente alto nivel para ejemplo 2

El siguiente programa manipula una variable de tipo booleano y realiza alguna operación lógica antes de mostrar resultados.

```
main
{
	boolean b1;

	scanf b1;
	 printf !b1;
	printf !!b1;
}
```

Posible programa principal

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

int main (int argc, char** argv)
{
	FILE * salida;
	int cuantos_no = 0;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}

	salida = fopen(argv[1],"w");

	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
	declarar_variable(salida, "b1", BOOLEANO, 1);

	escribir_segmento_codigo(salida);
	escribir_inicio_main(salida);

	/* scanf b1; */
	leer(salida,"b1",BOOLEANO);

	/* printf !b1; */
	escribir_operando(salida,"b1",1);
	no(salida,1,cuantos_no++);
	escribir(salida,0,BOOLEANO);

	/* printf !!b1; */
	escribir_operando(salida,"b1",1);
	no(salida,1,cuantos_no++);
	no(salida,0,cuantos_no++);
	escribir(salida,0,BOOLEANO);
```

```
	escribir_fin(salida);

	fclose(salida);
	return 0;
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej2`.

```
> ej2   /* La entrada es 0 y la salida true false*/
0
true
false

> ej2 /* La entrada es 1 y la salida false true */
1
false
true
```

3.3 Ejemplo 3

Código fuente alto nivel

Este tercer programa maneja datos enteros y realiza algunas operaciones con ellos a la vez que muestra sus resultados.

```
main
{
	int x;
	int y;
	int z;
	int j;

	scanf x;
	scanf z;
	j = - x;
	printf j;
	printf x-z;
	y=x/2;
	printf y;
	printf x*y;

}
```

Posible programa principal

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

int main (int argc, char** argv)
{
	FILE * salida;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}
```

```
	salida = fopen(argv[1],"w");

	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
	declarar_variable(salida, "x", ENTERO, 1);
	declarar_variable(salida,"y",ENTERO,1);
	declarar_variable(salida,"z",ENTERO,1);
	declarar_variable(salida,"j",ENTERO,1);

	escribir_segmento_codigo(salida);
	escribir_inicio_main(salida);

	/* scanf x; */
	leer(salida,"x",ENTERO);

	/* scanf z; */
	leer(salida,"z",ENTERO);

	/* j = - x; */
	escribir_operando(salida,"x",1);
	cambiar_signo(salida,1);
	asignar(salida,"j",0);

	/* printf j; */
	escribir_operando(salida,"j",1);
	escribir(salida,1,ENTERO);

	/* printf x-z; */
	escribir_operando(salida,"x",1);
	escribir_operando(salida,"z",1);
	restar(salida,1,1);
	escribir(salida,0,ENTERO);

	/* y=x/2;*/
	escribir_operando(salida,"x",1);
	escribir_operando(salida,"2",0);
	dividir(salida,1,0);
	asignar(salida,"y",0);

	/* printf y; */
	escribir_operando(salida,"y",1);
	escribir(salida,1,ENTERO);

	/*printf x*y;*/
	escribir_operando(salida,"x",1);
	escribir_operando(salida,"y",1);
	multiplicar(salida,1,1);
	escribir(salida,0,ENTERO);

	escribir_fin(salida);

	fclose(salida);
	return 0;
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej3`.

```
> ej3   /* La entrada es 10 3  y la salida -10 7 5 50*/
10
3
-10
7
5
50

> ej3 /* La entrada es 3 7 y la salida -3 -4 1 3 */
3
7
-3
-4
1
3
```

3.4 Ejemplo 4

Código fuente alto nivel

Este último programa maneja tanto datos enteros como booleanos, realizando operaciones lógicas y comparaciones.

```
main
{
	boolean b1;
	int x;

	scanf b1;
	scanf x;

	printf (x > 3);
	printf (x >= 3);
	printf (x < 3);
	printf (x <= 3);
	printf (x == 3);
	printf (x != 3);

	printf b1&&false;
	printf b1||true;
}
```

Posible programa principal

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

int main (int argc, char** argv)
{
	FILE * salida;
	int etiqueta = 0;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}

	salida = fopen(argv[1],"w");

	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
```

```
declarar_variable(salida, "b1", BOOLEANO, 1);
declarar_variable(salida, "x", ENTERO, 1);

escribir_segmento_codigo(salida);
escribir_inicio_main(salida);

/* scanf b1; */
leer(salida,"b1",BOOLEANO);
/* scanf x; */
leer(salida,"x",ENTERO);

/* printf (x > 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
mayor(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf (x >= 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
mayor_igual(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf (x < 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
menor(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf (x <= 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
menor_igual(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf (x == 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
igual(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf (x != 3); */
escribir_operando(salida,"x",1);
escribir_operando(salida,"3",0);
distinto(salida,1,0,etiqueta++);
escribir(salida,0,BOOLEANO);

/* printf b1&&false; */
escribir_operando(salida,"b1",1);
escribir_operando(salida,"0",0);
y(salida,1,0);
escribir(salida,0,BOOLEANO);

/* printf b1||true; */
escribir_operando(salida,"b1",1);
escribir_operando(salida,"1",0);
o(salida,1,0);
escribir(salida,0,BOOLEANO);

escribir_fin(salida);

fclose(salida);
return 0;
```

```
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej4`.

```
> ej4 /*La entrada es 1 3  y la salida false true false true true false false
true*/
1
3
false
true
false
true
true
false
false
true

> ej4 /*La entrada es 0 5 y la salida true true false false false true false true
*/
0
5
true
true
false
false
false
true
false
true
```

3.5 Ejemplo 5

Código fuente alto nivel

El siguiente programa hace uso de vectores dentro de un bucle while y nos sirve para comprender todas las funciones expuestas anteriormente que manipulan vectores y el bucle while. Es importante observar que el vector se declara de 4 posiciones y que el bucle va a iterar 5 veces, ya que en su condición se presenta un símbolo mayor o igual. Esto provocará un error en tiempo de ejecución, dado que el índice del vector estaría fuera de rango, que deberemos gestionar con el compilador.

```
main {
	int m;
	int [4] v;
	m = 0;
	while ( m <= 4 ){
		printf m;
		v[m] = m*10;
		m = m + 1;
	}
}
```

Posible programa principal

```
#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include "generacion.h"

int main (int argc, char ** argv)
{

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}

	int etiqueta = 0;
	int getiqueta = 0;
	int etiquetas[MAX_ETIQUETAS];
	int cima_etiquetas=-1;
	FILE * fd_asm;

	fd_asm = fopen(argv[1],"w");
	escribir_subseccion_data(fd_asm);
	escribir_cabecera_bss(fd_asm);

	//int m;
	declarar_variable(fd_asm,"m", 1, 1);

	//int [4] v;
	declarar_variable(fd_asm, "v",  1, 4);
	escribir_segmento_codigo(fd_asm);
	escribir_inicio_main(fd_asm);

	//m=0;
	escribir_operando(fd_asm,"0",0);
	asignar(fd_asm,"m",0);


	 //While. Gestion inicial de las etiquetas, guardado de etiqueta.
	getiqueta++;
	cima_etiquetas++;
	etiquetas[cima_etiquetas]=getiqueta;

	//Inicio del while. Impresion de la etiqueta.
	etiqueta = etiquetas[cima_etiquetas];
	while_inicio(fd_asm, etiqueta);

	//Condicion del bucle while.
	escribir_operando(fd_asm,"m",1);
	escribir_operando(fd_asm,"4",0);
	menor_igual(fd_asm,1,0,etiqueta);

	//Recuperamos la etiqueta para imprimir la comparacion del while.
	etiqueta = etiquetas[cima_etiquetas];
	while_exp_pila(fd_asm, 0, etiqueta);

	//printf m
	escribir_operando(fd_asm,"m",1);
	escribir(fd_asm,1,ENTERO);

	//v[m] = m*10;
	escribir_operando(fd_asm,"m",1);
	escribir_operando(fd_asm,"10",0);
	multiplicar(fd_asm,1,0);
	escribir_operando(fd_asm,"m",1);
	escribir_elemento_vector(fd_asm,"v", 4, 1);
```

```
	asignarDestinoEnPila(fd_asm,0);

	 //m = m + 1
	escribir_operando(fd_asm,"m",1);
	escribir_operando(fd_asm,"1",0);
	sumar(fd_asm,1,0);
	asignar(fd_asm,"m",0);

	//Recuperamos la etiqueta para imprimir el fin de etiqueta del while.
	etiqueta = etiquetas[cima_etiquetas];
	while_fin(fd_asm, etiqueta);

	//Al cerrar el ámbito, decrementamos el contador de cima de etiquetas.
	cima_etiquetas--;

	escribir_fin(fd_asm);

	fclose(fd_asm);
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej5`. Como se ha mencionado anteriormente, la salida del compilador debe controlar el error en tiempo de ejecución en el que incurre el programa mostrado anteriormente.

```
$ ./ej5
0
1
2
3
4
Indice de vector fuera de rango
```

3.5 Ejemplo 6

Código fuente alto nivel

El siguiente programa emplea una estructura condicional if-else. Este programa nos obliga a emplear todas las funciones de generación de código reservadas para las sentencias condicionales. El flujo de ejecución del programa deberá ejecutar el bloque de código de la sentencia else, ya que m no es mayor que 5.

```
main{
	int m;
	m = 0;
	if(m>5)
	{
		printf 2;
	}
	else
```

```
	{
		printf 3;
	}
}
```

Posible programa principal

```
#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include "generacion.h"

int main (int argc, char ** argv)
{

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}

	int etiqueta = 0;
	int getiqueta = 0;
	int etiquetas[MAX_ETIQUETAS];
	int cima_etiquetas=-1;
	FILE * fd_asm;

	fd_asm = fopen(argv[1],"w");
	escribir_subseccion_data(fd_asm);
	escribir_cabecera_bss(fd_asm);

	//int m;
	declarar_variable(fd_asm,"m", 1, 1);

	escribir_segmento_codigo(fd_asm);
	escribir_inicio_main(fd_asm);

	//m=0;
	escribir_operando(fd_asm,"0",0);
	asignar(fd_asm,"m",0);

	//Gestion de etiquetas para abrir el ambito del if. Esta sera la etiqueta
que tenga.
	getiqueta++;
	cima_etiquetas++;
	etiquetas[cima_etiquetas]=getiqueta;
	etiqueta = getiqueta;

//Condición del if. Salto a fin si, si la condicion se da.
	escribir_operando(fd_asm,"m",1);
	escribir_operando(fd_asm,"5",0);
	mayor(fd_asm,1,0,etiqueta);
	ifthenelse_inicio(fd_asm, 0, etiqueta);

	//printf 2
	escribir_operando(fd_asm,"2",0);
	escribir(fd_asm,0,ENTERO);

	//Salto a fin sino al terminar el if, impresion de la etiqueta fin_si.
Recogemos la etiqueta.
	etiqueta = etiquetas[cima_etiquetas];
	ifthenelse_fin_then(fd_asm, etiqueta);
```

```
	//printf 3
	escribir_operando(fd_asm,"3",0);
	escribir(fd_asm,0,ENTERO);

	//Fin del condicional if. Imprimimos la etiqueta de ambito del fin del
condicional y restamos el contador.
	etiqueta = etiquetas[cima_etiquetas];
	ifthenelse_fin(fd_asm, etiqueta);
	cima_etiquetas--;

	escribir_fin(fd_asm);

	fclose(fd_asm);
}
```

Salida esperada

Supondremos que el ejecutable se llama `ej6`.

```
$ ./ej6
3
```

3.7 Ejemplo 7

Código fuente alto nivel

Este último programa de ejemplo invoca a una función, por lo que nos hace emplear todas las funciones de generación de código respectivas a la invocación de una función.

```
main{
	int z;
	function int doble(int arg)
	{
		int auxArg;
		auxArg = arg;
		return 2*arg;
	}
	z=2;
	printf doble(z);
}
```

Posible programa principal

```
#include "generacion.h"

int main (int argc, char ** argv)
{
        int etiqueta = 0;
        int getiqueta = 0;
        int etiquetas[MAX_ETIQUETAS];
        int cima_etiquetas=-1;
      FILE * fd_asm;

        fd_asm = fopen(argv[1],"w");
        escribir_subseccion_data(fd_asm);
        escribir_cabecera_bss(fd_asm);

        //int z;
        declarar_variable(fd_asm,"z", 1, 1);

      escribir_segmento_codigo(fd_asm);
      //Declaramos la funcion. Vamos a imprimir su etiqueta y decir que tiene una
variable local.
      //function int doble(int arg)
        //{
        //        int auxArg;
      declararFuncion(fd_asm,"doble",1);

      //auxArg = arg; Asignacion de parametro a variable local. Solo hay un
parametro.
      escribirParametro(fd_asm,0,1);
      escribirVariableLocal(fd_asm,1);
      asignarDestinoEnPila(fd_asm,1);

      //2*arg.
      escribir_operando(fd_asm,"2",0);
      escribirParametro(fd_asm,0,1);
      multiplicar(fd_asm,0,1);

      //Retornamos de la funcion con lo que esta encima de la pila.
      retornarFuncion(fd_asm,0);

      escribir_inicio_main(fd_asm);
      //z=2
      escribir_operando(fd_asm,"2",0);
       asignar(fd_asm,"z",0);

      escribir_operando(fd_asm,"z",1);

      // printf doble(z)

      //Llamamos a la funcion que tiene 1 argumento. Estamos dando un salto a la
etiqueta. Primero apilamos el parametro.
      operandoEnPilaAArgumento(fd_asm,1);
      llamarFuncion(fd_asm,"doble",1);


      //Imprimimos el resultado de la funcion.
      escribir(fd_asm,0,ENTERO);


      escribir_fin(fd_asm);
      fclose(fd_asm);

}
```

Salida esperada

Supondremos que el ejecutable se llama `ej7`, se imprime el doble del valor que se le proporciona a la función 2.

```
$ ./ej7
4
```

Normas de entrega

Según te indique el profesor de tu grupo.
Realizarás un zip con todos los ficheros necesarios.