Examen de compilación. 2 de junio de 2014

1. En un lenguaje de programación los identificadores son un conjunto de caracteres alfanuméricos. Aceptan también el guión bajo pero con las siguientes restricciones: No admite dos guiones bajo seguidos y después de un guión siempre viene una letra mayúscula salvo si detrás no hay nada. Escribe la expresión regular en notación FLEX correspondiente a este tipo de token.

Ejemplos:

Correctos	**Incorrectos**
_Bien	**__Mal**
5var_Bien	**var_mal**
bien_	**mal_42**

(1 punto)

2. Queremos definir un nuevo tipo de comentario en la práctica de compilación. Estos comentarios comienzan con la secuencia ***[:*** acaban con la secuencia ***:]*** y no acepta en la mitad ni saltos de línea ni la secuencia ***:]***. Ver ejemplos. Escribe un **autómata finito determinista** que reconozca este tipo de cometarios.

Correctos	**Incorrectos**
[:cometario correcto::]	[:]
[::::_&]:$::::]	[: comentario :] incorrecto :]
[: con blanco en medio :] correcto! :]	[: no se admiten saltos de línea! :]

(1 punto)

3. Dada la siguiente gramática, calcula los conjuntos PRIMERO y SIGUIENTE de los no terminales, y basándote en el resultado indica si la gramática cumple o no las condiciones LL(1), razonando la respuesta. Es imprescindible que indiques cuáles son las condiciones LL(1) y verificarlas en todos los casos. En caso de que no sea LL(1), realiza las transformaciones necesarias para que lo sea.

E → E or T	**T → T and F**	**F → id**
| T	**| F**	**| (E)**
		| id[E]

(1 punto)

4. Dada la siguiente gramática, completar su tabla de análisis descendente (que aparece en las hojas de soluciones) y realiza el análisis de la siguiente cadena: **id * id | id**. ¿Es aceptada la entrada por la gramática?

E → T E'	**T → F T'**	**F→ (E) F'**	**F ' → ***
E' → '|' T E'	**T' → F T'**	**| id F'**	**| ξ**
| ξ	**| ξ**		

(1 punto)

5. Dada la siguiente gramática y la tabla de de análisis ascendente descrita en la hoja de soluciones, analiza la secuencia **- - id + id - id**

1) E -> E + E
2) | -E
3) | id

(1 punto)

6. En la hoja de soluciones encontrarás una secuencia de instrucciones en código intermedio. Primeramente señala cuáles son los líderes, cuáles son los bloques básicos y después dibuja el grafo de flujo que corresponde a esa secuencia de código.

(1 punto)

7. Aplica el algoritmo de generación de código a la siguiente secuencia de instrucciones, partiendo de la situación inicial descrita en la *hoja de soluciones*. Todas las variables son de usuario, y deben estar en memoria al término del bloque.

1. c := b - a
2. d := c - b
3. c := d - a
4. a := b

(1 punto)

Algoritmo de generación de código

Para cada instrucción de la forma **x := y op z** (similar para **x := op y**) se realizan las siguientes acciones:

1. Llamar a la función **obtenreg**, para determinar la localización L sobre la que se realizará el cálculo de *y op z*. Habitualmente, L será un registro, aunque puede ser una dirección de memoria.
2. Consultar el Descriptor de direcciones para *y* con el objetivo de determinar *y'*, una de las localizaciones en curso de *y* (*).

Si el valor de *y* no se encuentra en L, generar la instrucción **MOV y',L** para obtener una copia de *y'* en L.

3. Generar la instrucción **op z' , L**, donde *z'* es una de las localizaciones en curso de *z* (*). Modificar el valor del descriptor de direcciones de *x*, indicando que su valor actual se encuentra en L. Si L es un registro, modificar su descriptor para indicar que contiene el valor de *x*.
4. Si el valor de *y* y/o *z* no tiene/n más usos, esto es, no están vivos a la salida del bloque, alterar la descripción de los registros que contengan los valores de *y* y/o *z*, para indicar que sus valores no volverán a ser utilizados.

Para cada instrucción de la forma **x := y** se realizan las siguientes acciones:

1. Si el valor de *y* se encuentra en un registro, cambiar los descriptores para reflejar que *x* e *y* están en el mismo registro. Si *y* no tiene más usos, es decir, no está viva después de la instrucción, borrarla del descriptor de registros.
2. En caso contrario, aplicar el algoritmo de generación para instrucciones de la forma **x := y op z**

(*)Si el valor de *y* se encuentra en un registro y en memoria, se escogerá el registro.

obtenreg

Devuelve la localización L donde quedará el valor de *x* tras la ejecución de la instrucción **x := y op z**. Este algoritmo se basa en la información sobre próximo uso:

1. Si el valor de y se encuentra en un registro que no contiene el valor de otras variables, e y no está viva despues de la ejecución de **x := y op z**, entonces devolver la localización de y. Modificar el descriptor de y para indicar que y no se encuentra a partir de este momento en L.
2. Si falla 1, devolver el primer registro vacío, caso de que exista alguno.
3. Si falla 2, si *x* tiene algún uso más en el bloque, incluida la instrucción actual (o bien **op** exige un registro) buscar un registro ocupado R. Guardar el valor de R en memoria si no se encuentra ya en ella (MOV R, M). Modificar el descriptor de memoria para M, y devolver R. Si R contenía el valor de varias variables, será necesaria una instrucción MOV para cada una de ellas.
4. Si x no se usa en el resto del bloque, devolver la dirección de *x*.

8. Las hojas de soluciones incluyen una especificación parcial de BISON que traduce las expresiones y while-do a código intermedio. La semántica del while-do es la siguiente: el bucle se repite mientras la expresión sea cierta; en caso contrario, sale del bucle. Examina si la especificación es correcta, y si no lo es escribe la especificación BISON correcta.

(1,5 puntos)

9. Queremos añadir dos nuevas instrucciones al ETDS visto en clase, *repetir-until y salto***.** Su forma es la siguiente:

repetir **salto;**
lista_sentencias
until expresion
end repetir ;

La semántica de la instrucción *repetir-until* es un bucle que ejecuta la *lista_sentencias* hasta que la expresion es cierta. Cuando el bucle del *repetir* encuentra un *salto*, interrumpe el ciclo y vuelve al inicio del bucle.

(1,5 puntos)