# Examen de Compilación. 30 de mayo de 2016

**1.** Definimos un token *comentario* en nuestro lenguaje de programación. Comienza con los caracteres **<!** Y finaliza con los caracteres **/>**. Dentro del comentario no puede haber un carácter **/** ni tampoco puede aparecer más de 2 guiones seguidos.

Escribe la **expresión regular en notación FLEX** que describe este token.

| **Ejemplos:** | **comentarios válidos** | **comentarios no válidos** |
|---|---|---|
| | `<! comentario> />` | `<!--- comentario mal` |
| | `<! -- - -- - --/>` | `<! - /> esto va fuera />` |
| | `<!/>` | `<!--/--/>` |
| | `<!-/>` | `<!---/>` |

**(1 punto)**

**2.** En muchos lenguajes de programación, se pueden especificar diferentes formatos para imprimir los números reales. Se suelen dar estas opciones:

```
%[flags][width][.precision]specifier
```

Estos son los modelos:

`flags`: opcional, El caráctert - + o blanco
`width`: opcional, un número entero
`precision`: opcional, (un punto y) un número entero
`specifier`: obligatorio, uno de estos caracteres: e E f F g G

| **Ejemplos:** | **Válidos** | **No válidos** |
|---|---|---|
| | `%f` | `%--f` |
| | `%-f` | `%.f` |
| | `%-4.15F` | `%-4.5` |
| | `% .4g` | `%4-f` |

Escribe el **autómata finito determinista** que acepta este token. **(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 → T ^ E** | **T → T . F** | **F → id** |
| **\| T** | **\| F** | **\| F ( )** |
| | | **\| * F** |
| | | **\| ( E )** |

**(1 punto)**

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

**E → T E'** **T → id**
**E' → + T E'** **| – ( E )**
**| – T E'**
**| ξ**

**(1 punto)**

**5.** Dada la siguiente gramática y la tabla de de análisis ascendente descrita en la hoja de soluciones, analiza la secuencia: **int id , id , id ;** ¿Es generada la entrada por esta gramática?

(1) D → T **id** L
(2) T → int
(3) T → float
(4) L → **, id** L
(5) L → **;**

**(1 punto)**

**6.** El núcleo del método para asociar la información sobre *próximo uso* a una secuencia de instrucciones tiene la forma:

> Si la instrucción i tiene la forma *x := y op z*, entonces:
>
> 1. Asociar a la instrucción i la información en curso en la tabla de símbolos sobre *y, x* y *z*.
> 2. Marcar x en la tabla de símbolos como "muerto".
> 3. Marcar *y* y *z* como "vivos" e introducir *i* como próximo uso de *y* y *z*.

Aplica el método a la siguiente secuencia de instrucciones (*a*, *b*, y *c* son variables no-temporales; *t1*, *t2* y *t3* son variables temporales), reflejando paso a paso la evolución del contenido de la tabla de símbolos. Visto el resultado, ¿se podría decir que hay código muerto? Justifica tu respuesta.

1. a := a * a
2. b := b * b
3. t1 := a + b
4. t2 := t1 - a
5. t3 := t1 + t2
6. t2 := a * t3
7. c := t1

**(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*.

1. t1 := a – b
2. a:= t1 - c
3. t2 := a – c
4. c := t2 **(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 la sentencia ***if-then-else*** a código intermedio. 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, *while-in range y continue***.** Su forma es la siguiente:

**while id in range** E **..** E **begin** lista_sentencias **end;**

a) La semántica de la instrucción *while-in range* es un bucle que ejecuta la *lista_sentencias* mientras **id** esté entre E1 y E2 (incluidas). Las expresiones E1 y E2 se evaluarán solo la primera vez que entra en el bucle. Realiza el ETDS que especifica la traducción de esta sentencia.
b) Añade al ETDS la siguiente comprobación: ***Detectará al inicio si id está fuera del rango de E1 y E2***. ¿Esta comprobación se realizará en tiempo de compilación o de ejecución?

**(1,5 puntos)**