LTP 2B - L2 - Examen primer parcial - 4 de Noviembre de 2020

Alumno:

1 Cuestiones (40%)

1.1. (10%) Describe en qué consiste la *reflexión* en los lenguajes de programación. Enumera sus pros y sus contras. Da al menos dos ejemplos de lenguajes con reflexión.

Solución. En los lenguajes de programación, la reflexión es *la infraestructura* que, durante su ejecución, permite a un programa ver su propia estructura y manipularse a sí mismo. Como aspecto *positivo*, permite, por ejemplo, definir programas capaces de monitorizar su propia ejecución y modificarse, en tiempo de ejecución, para adaptarse dinámicamente a distintas situaciones. Sin embargo, mal usada puede conllevar aspectos *negativos*, afectando al rendimiendo del sistema, ya que suele ser costosa, y a la seguridad, pudiendo exponer información comprometida sobre el código, ya que la reflexión rompe la abstracción, permite acceder a atributos y métodos privados, etc. Ejemplos de lenguajes de programación con capacidades de reflexión son *Java* y *LISP*.

1.2. (10%) Indica qué tipos de áreas de la memoria se distinguen en función de su uso en la ejecución de un programa y cómo se utiliza cada una de ellas.

Solución. Tipos de de almacenamiento según su asignación:

- Estático. Calculado en tiempo de compilación, permanece fijo durante la ejecución del programa. Se suele usar con: *variables globales* (o `static`), *programa compilado* (instrucciones en lenguaje máquina), *variables locales* a un subprograma cuyo valor no cambia en las diferentes llamadas, *constantes* numéricas y cadenas de caracteres, *tablas* producidas por los compiladores y usadas para operaciones de ayuda en tiempo de ejecución (e.g., comprobación dinámica de tipos, depuración, ...). Es eficiente pero incompatible con la recursión o las estructuras de datos dinámicas
- Dinámico. Se calcula y asigna en tiempo de ejecución:
  - en la *pila*, que permite manejar los registros de activación en las llamadas a función/procedimiento durante la ejecución del programa: cuando se requiere espacio de almacenamiento (hay una llamada), éste se toma a partir del *tope* de la pila; una vez terminada la llamada, el espacio se libera *desapilando* lo que fue asignado.
  - en el *heap*, región de almacenamiento en la que los bloques de memoria se asignan y liberan en *momentos arbitrarios*. Es necesario cuando el lenguaje permite estructuras de datos (e.g, conjuntos, listas) cuyo tamaño puede cambiar en tiempo de ejecución o que se crean durante la ejecución (e.g., objetos). Los subbloques asignados pueden ser del *mismo tamaño siempre o de tamaño variable* La desasignación puede ser explícita o implícita (cuando el elemento asignado ya no es alcanzable por ninguna variable del programa) mediante un subsistema denominado *garbage collector*.

1.3. (10%) Describe las etapas del proceso de compilación de un programa indicando brevemente qué problema resuelve cada una.

Solución.

- El *analizador léxico* (*scanner*) divide una secuencia de caracteres (el *programa*) en una secuencia de componentes sintácticos primitivos o palabras (*tokens*) que actúan como identificadores, números, palabras reservadas, etc.

- El *analizador sintáctico* (*parser*) reconoce una secuencia de *tokens* y obtiene una secuencia de *instrucciones* en forma de *árbol sintáctico*.
- El *analizador semántico* parte del árbol sintáctico y comprueba restricciones de la sintaxis que *no* pueden expresarse mediante la notación BNF (variables no declaradas, coherencia de tipos y número de argumentos en llamadas a función, etc.). Después produce código intermedio.
- En la etapa de *optimización* se obtiene una versión mejorada del código intermedio obtenido por el analizador semántico.
- En la etapa de *generación de código*, el código intermedio optimizado se traduce en instrucciones del código máquina del procesador donde vaya a ejecutarse el programa (código objeto).
- En la etapa de *enlazado* se combina el código objeto del programa compilado con librerías auxiliares y otros módulos objeto de programas previamente compilados para obtener el programa *ejecutable*.

1.4. (10%) ¿Cómo se demuestra la corrección de un programa empleando el cálculo de la precondición más débil? Da un ejemplo de terna de Hoare que, independientemente del programa considerado, *siempre* permita demostrar su corrección.

Solución. Dada una terna de Hoare $\{P\}\ S\ \{Q\}$, utilizando el cálculo ideado por Dijkstra se obtiene la *precondición más débil* $P' = pmd(S, Q)$ asociada a la terna. Una vez obtenida ésta, se comprueba si la precondición P de la terna implica la precondición más débil, es decir, si se cumple que $P \Rightarrow P'$. Si es así, la terna es correcta.

Un ejemplo de terna de Hoare que siempre es correcta es $\{P\}\ S\ \{true\}$. Esto es así por la definición de corrección de una terna de Hoare: cualquier estado producido por la ejecución de S (sea cual sea el estado inicial) siempre satisface la postcondición *true*.

2 Problemas (60%)

2.1. (20%) Las siguientes clases de Java describen clases que pueden utilizarse para representar la semántica operacional de paso pequeño del lenguaje SIMP. (...)

1. Señala las líneas donde se utiliza *sobrecarga* y *coerción* en el código. Justifica tus afirmaciones.

   Solución.

   - Respecto al uso de la *sobrecarga*, en las líneas 35, i.e.,

     ```
     35 public int eval(State s) { return s.vvalues[x]; }
     ```

     así como en la 76, i.e.,

     ```
     76 public Boolean eval(State s)
     ```

     se definen dos versiones distintas del método `eval`, con tipos de retorno distintos: `int` y `Boolean`, respectivamente.
   - En cuanto a la *coerción*, se utiliza implícitamente en la línea 50, i.e.,

     ```
     50 for (i=0; i<n; ++i) add = add + aexps[i].eval(s);)
     ```

     para poder sumar la variable `add`(de tipo `long`) y el valor devuelto por `aexps[i].eval(s)` (de tipo `int`). En la línea 51, i.e,

     ```
     51 return (int) (add / n);
     ```

     se realiza una coerción *explícita* para transformar el cociente entre `add` y `n` en un entero.

2. Señala las líneas donde se utiliza *genericidad* en el código. Justifica tus afirmaciones.

   Solución. En la línea 10, i.e.,

   ```
   10 abstract class Expression<V> {
   ```

   se inicia la definición de la clase genérica `Expression<V>` que contiene un método genérico `eval` en la línea 11, i.e.,

   ```
   11 abstract public V eval(State s);
   ```

   La clase `Expression<V>` se particulariza en dos clases, `ArithExpression`, cuando `V` se asocia a la clase `Integer`, línea 21, i.e.,

   ```
   21 extends Expression<Integer> {
   ```

   y `BoolExpression`, cuando `V` se asocia a la clase `Boolean`, línea 61, i.e.,

   ```
   61 extends Expression<Boolean> {
   ```

   que proporcionan implementaciones (sobrecargadas) del método `eval`.

3. ¿En las líneas 2 y 3, podría reemplazarse `public` por `protected`? ¿Y por `private`? Justifica cada respuesta.

   Solución.

   - En la línea 2 de la definición de la clase `State`, i.e.,

     ```
     2 public int vvalues[];
     ```

     puede reemplazarse `public` por `protected` porque este modificador permite que los identificadores sean visibles dentro del mismo *package* de Java. Sin embargo, no puede reemplazarse por `private` porque entonces la variable `vvalues` no sería visible en la línea 35, i.e.,

     ```
     35 public int eval(State s) { return s.vvalues[x]; }
     ```

     que forma parte del código de la clase `VariableExp`.

   - En la línea 3, i.e.,

     ```
     3 public int nvars;
     ```

     puede reemplazarse `public` tanto por `protected` como por `private` porque `nvars` solo es accedida en la línea 6, i.e.,

     ```
     6 nvars = size; vvalues = new int[size];
     ```

     que se encuentra dentro del código de la clase `State` donde se declara el atributo.

4. Las clases `ArithmeticExp` y `BooleanExp` no declaran atributos ni métodos, pero a pesar de todo son abstractas. ¿Podría eliminarse el modificador `abstract`? Justifica tu respuesta.

   Solución. No sería posible, porque tanto `ArithmeticExp` como `BooleanExp` heredan el método abstracto `eval` de `Expression`. Como no proporcionan una implementación para éste, deben seguir siendo abstractas.

2.2. (10%) Considera la siguiente clase `Main`: (...)

- ¿Hay algún tipo de uso de la *sobrecarga*? ¿Y de la *coerción*? ¿Dónde?

  Solución.

  – Con relación a la *sobrecarga*, en las líneas 114 y 115, i.e.,

  ```
  114 System.out.print(str + ".  Hence,  (x <= y) is ");
  115 System.out.print(b.eval(s));
  ```

  se utiliza el método `print` para mostrar cadenas de caracteres (`String`) y booleanos (`Boolean`), respectivamente.

  – Con relación a la *coerción*, en la línea 112, i.e.,

  ```
  112 "x = " + s.vvalues[0] + " and y = " + s.vvalues[1];
  ```

  se utiliza coerción implícita de `s.vvalues[0]`, de tipo `int`, a `string` para poder utilizar el operador de concatenación de cadenas de caracteres.

- En la línea 106, ¿podría utilizarse el constructor de `ArithmeticExp` en lugar del de `VariableExp` para asignar el objeto a la variable `a1`, de tipo `ArithmeticExp`? ¿Podría la variable `leqBexp` de la línea 108 declararse de clase `LeqExp` en vez de `BooleanExp`? ¿Podría la variable `b` de la línea 109 declararse también de clase `LeqExp`?

  Solución.

  – En la línea 106, i.e.,

  ```
  106 ArithmeticExp a1 = new VariableExp(0);
  ```

  no se puede utilizar el constructor de la clase `ArithmeticExp` para obtener un objeto porque es una clase abstracta.

  – En la línea 108, i.e.,

  ```
  108 BooleanExp leqBexp = new LeqExp(a1,a2);
  ```

  la variable `leqBexp` puede declararse de clase `LeqExp` sin problemas, es decir, podemos escribir

  ```
  108' LeqExp leqBexp = new LeqExp(a1,a2);
  ```

  porque lo único que se hace con ella es utilizarla como parámetro en el constructor de `NotExp` en la línea 109, i.e.,

  ```
  109 BooleanExp b = new NotExp(leqBexp);
  ```

  cuyo tipo es `BooleanExp`, superclase de `LeqExp`.

  – En la línea 109 /ver arriba), la variable `b` *no* puede declararse de clase `LeqExp` puesto que el objeto construido es de la clase `NotExp`, que no es subclase de `LeqExp`.

- ¿En qué parte de la memoria se alojan los *objetos* referenciados por las variables `s` y `b` (líneas 102 y 109)? ¿Y las *variables* `s` y `b`?

  Solución. La línea 109 se muestra más arriba y la línea 102 es:

  ```
  102 State s = new State(2);
  ```

  Los objetos se alojan en el *heap*, mientras que las variables, como son locales al método `main`, se alojan en la pila.

2.3. (20%) El lenguaje de programación C permite instrucciones como x `<<=`d (donde x es una variable y d es una expresión aritmética) que asigna a la variable x el resultado de *desplazar* su valor n posiciones donde n es el resultado de evaluar la expresión d. Teniendo en cuenta que cada desplazamiento binario es equivalente a multiplicar por 2, se pide:

1. Define la semántica operacional de paso pequeño para esta nueva instrucción por *traducción* de la misma a las instrucciones ya existentes en SIMP.

   Solución.

   $$\langle x \texttt{ <<= } d, s\rangle \to \langle \texttt{if } d\texttt{<=0 then skip else } x \texttt{ := 2*}d\texttt{*}x, s\rangle$$

2. Realizar los cambios necesarios en la semántica de paso pequeño y paso grande (*small-step*/*big-step*) de SIMP para dar semántica a la nueva instrucción de forma *directa*(no por traducción).

   Solución.

   - En el caso *small-step*:

   $$\frac{\langle d, s\rangle \Rightarrow m}{\langle x \texttt{ <<= } d, s\rangle \to \langle \texttt{skip}, s\rangle} \quad \text{if } m \leq 0$$

   $$\frac{\langle d, s\rangle \Rightarrow m \qquad \langle x, s\rangle \Rightarrow n}{\langle x \texttt{ <<= } d, s\rangle \to \langle \texttt{skip}, s[x \mapsto 2mn]\rangle} \quad \text{if } m > 0$$

   - En el caso *big-step*:

   $$\frac{\langle d, s\rangle \Rightarrow m}{\langle x \texttt{ <<= } d, s\rangle \Downarrow s} \quad \text{if } m \leq 0$$

   $$\frac{\langle d, s\rangle \Rightarrow m \qquad \langle x, s\rangle \Rightarrow n}{\langle x \texttt{ <<= } d, s\rangle \Downarrow s[x \mapsto 2mn]} \quad \text{if } m > 0$$

3. Mostrar la traza de ejecución completa para el siguiente programa:

```
n := 1;
n <<= n;
while (n>1) do n := n-1
```

   Solución.

(1) $\langle$`n := 1; n <<= n; while (n>1) do n := n-1`$, \{\}\rangle \to$
  $\langle$`n := 1`$, \{\}\rangle \to$
    $\langle 1, \{\}\rangle \Rightarrow 1$
  $\langle$`skip`$, \{n \mapsto 1\}\rangle$
(2) $\langle$`skip;n <<= n; while (n>1) do n := n-1`$, \{n \mapsto 1\}\rangle \to$
(3) $\langle$`n <<= n; while (n>1) do n := n-1`$, \{n \mapsto 1\}\rangle \to$
  $\langle$`n <<= n`$, \{n \mapsto 1\}\rangle \to$
    $\langle$`n`$, \{n \mapsto 1\}\rangle \Rightarrow 1$
    $\langle$`n`$, \{n \mapsto 1\}\rangle \Rightarrow 1$
  $\langle$`skip`$, \{n \mapsto 2\}\rangle$
(4) $\langle$`skip; while (n>1) do n := n-1`$, \{n \mapsto 2\}\rangle \to$
(5) $\langle$`while (n>1) do n := n-1`$, \{n \mapsto 2\}\rangle \to$
  $\langle$`n>1`$, \{n \mapsto 2\}\rangle \Rightarrow$
    $\langle$`n`$, \{n \mapsto 2\}\rangle \Rightarrow 2$
    $\langle$`1`$, \{n \mapsto 2\}\rangle \Rightarrow 1$
  true

(6) $\langle$`n := n-1; while (n>1) do n := n-1`$, \{n \mapsto 2\}\rangle \rightarrow$
$\quad \langle$`n := n-1`$, \{n \mapsto 2\}\rangle \rightarrow$
$\quad\quad \langle$`n-1`$, \{n \mapsto 2\}\rangle \Rightarrow 1$
$\quad \langle$`skip`$, \{n \mapsto 1\}\rangle$
(7) $\langle$`skip; while (n>1) do n := n-1`$, \{n \mapsto 1\}\rangle \rightarrow$
(8) $\langle$`while (n>1) do n := n-1`$, \{n \mapsto 1\}\rangle \rightarrow$
$\quad \langle$`n>1`$, \{n \mapsto 1\}\rangle \Rightarrow$
$\quad\quad \langle$`n`$, \{n \mapsto 1\}\rangle \Rightarrow 1$
$\quad\quad \langle$`1`$, \{n \mapsto 1\}\rangle \Rightarrow 1$
$\quad$ *false*
(9) $\langle$`skip`$, \{n \mapsto 1\}\rangle$

2.4. (**10%**) Escoger una de las siguientes:

1. Modificar la definición de *pmd* para poder utilizar también la instrucción x `<<=` d definida en el apartado anterior.

   Solución.
   $$pmd(x \texttt{ <<= } d, Q) = (d > 0 \wedge Q[x \mapsto 2 \cdot d \cdot x]) \vee (d \leq 0 \wedge Q)$$

2. ¿Es correcta la siguiente terna de Hoare? Justifica tu respuesta indicando los cálculos necesarios para obtenerla.

```
{ x > 0 }

  if x < y then z := x else z := y
  y := x+1;

{ 2z = x+y }
```

   Solución.

$$\begin{aligned}
&pmd(\texttt{if x < y then z := x else z := y; y := x+1},\ 2z = x + y)\\
&\quad = pmd(\texttt{if x < y then z := x else z := y}, pmd(\texttt{y := x+1},\ 2z = x + y))\\
&\quad = pmd(\texttt{if x < y then z := x else z := y}, 2z = 2x + 1)\\
&\quad = (x < y \wedge pmd(z := x, 2z = 2x + 1)) \vee (x = y \wedge pmd(z := y, 2z = 2x + 1))\\
&\quad = (x < y \wedge 2x = 2x + 1)) \vee (x \geq y \wedge 2y = 2x + 1))\\
&\quad \Leftrightarrow (x < y \wedge \mathit{false})) \vee (x \geq y \wedge 2y = 2x + 1))\\
&\quad \Leftrightarrow \mathit{false} \vee (x \geq y \wedge 2y = 2x + 1))\\
&\quad \Leftrightarrow \mathit{false}
\end{aligned}$$

   Dado que $P \Rightarrow \mathit{false}$ nunca es cierto, la terna *no* es correcta.