Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 – Lenguajes de Programación I
Abril–Julio 2022

Examen 3

(30 puntos)

A continuación encontrará 6 preguntas (y una sorpresa al final), cada una de las cuales tiene un valor de 5 puntos. Sea lo más detallado y preciso posible en sus razonamientos y procedimientos.

En algunas preguntas, se usarán las constantes X, Y y Z. Estas constantes debe obtenerlas de los últimos tres números de su carné. Por ejemplo, si su carné es 09-40325, entonces $X = 3$, $Y = 2$ y $Z = 5$.

En aquellas preguntas donde se le pida decir qué imprime un programa, incluya los pasos relevantes de la ejecución del mismo con los cuales usted pudo alcanzar su conclusión.

En aquellas preguntas donde se le pida implementar un programa, mantenga su código en un repositorio `git` remoto (preferiblemente `Github`) y coloque un enlace al mismo en lugar de su respuesta. Todo su código debe ser legible y estar debidamente documentado.

La entrega se realizará por correo electrónico a rmonascal@gmail.com hasta las 11:59pm. VET del Miércoles 03 de Agosto de 2022.

1. Escoja algún lenguaje de programación de alto nivel, de propósito general y orientado a objetos que **no** comience con *J*, *C* o *P*.

   (a) De una breve descripción del lenguaje escogido.

      i. Explique la manera de crear y manipular objetos que tiene el lenguaje, incluyendo: constructores, métodos, campos, etc.

      ii. Describa el funcionamiento del manejo de memoria, ya sea explícito (`new/delete`) o implícito (recolector de basura).

      iii. Diga si el lenguaje usa asociación estática o dinámica de métodos y si hay forma de alterar la elección por defecto del lenguaje.

      iv. Describa la jerarquía de tipos, incluyendo mecanismos de herencia múltiple (de haberlos), polimorfismo paramétrico (de tenerlo) y manejo de varianzas.

   (b) Implemente los siguientes programas en el lenguaje escogido:

      i. Defina un interfaz o clase abstracta `Secuencia`, que represente una colección ordenada de elementos. Debe tener los siguientes métodos:

         - **agregar:** Recibe un elemento y lo agrega a la secuencia.
         - **remover:** Devuelve un elemento de la secuencia y lo elimina de la misma. Arroja un error si está vacía.
         - **vacio:** Dice si la secuencia está vacía (no tiene elementos).

         Defina dos clases concretas `Pila` y `Cola` que sean subtipo de `Secuencia`

         - Para `Pila` los elementos se manejan de tal forma que el *último* en ser agregado es el *primero* en ser removido.
         - Para `Cola` los elementos se manejan de tal forma que el *primero* en ser agregado es el *primero* en ser removido.

      ii. Defina un tipo de datos que represente grafos como listas de adyacencias y cada nodo sea representado por un número entero (puede usar todas las librerías a su disposición en el lenguaje).
      Además, defina una clase abstracta `Busqueda` que debe tener un método **buscar**. Este método debe recibir dos enteros: `D` y `H`, y debe devolver la cantidad de nodos explorados, partiendo desde el nodo `D` hasta llegar al nodo `H`. En caso de que `H` no sea alcanzable desde `D`, debe devolver el valor -1 (menos uno).
      Esta clase debe estar parcialmente implementada, dejando solamente abstraído el orden en el que se han de explorar los nodos.
      Defina dos clases concretas `DFS` y `BFS` que sean subtipo de `Busqueda`.

         - Para `DFS` el orden de selección de nodos es a profundidad (usando un pila).
         - Para `BFS` el orden de selección de nodos es a amplitud (usando un cola).

2. Escoja algún lenguaje de programación de alto nivel y de propósito general cuyo nombre tenga la misma cantidad de caracteres que su primer nombre.

   (a) De una breve descripción de los mecanismos de concurrencia disponibles en su lenguaje.

      i. Diga si su lenguaje provee capacidades nativas para concurrencia, usa librerías o depende de herramientas externas.

      ii. Explique la creación/manejo de tareas concurrentes, así como el control de la memoria compartida y/o pasaje de mensajes.

      iii. Describa el mecanismo de sincronización que utiliza el lenguaje.

   (b) Implemente los siguientes programas en el lenguaje escogido (usando la herramienta adecuada):

      i. Dadas dos vectores del mismo tamaño (representados como un arreglo), realizar el producto punto.
      El cálculo debe hacerse de forma concurrente, aprovechando los mecanismos provistos en el lenguaje para ello.

      ii. Dado un *path* que representa un directorio en el sistema operativo, cuenta la cantidad de archivos que están localizados en el subarbol que tiene como raíz el directorio propuesto.
      El proceso debe crear un *thread* por cada subdirectorio encontrado.

3. Tomando como referencia las constantes X, Y y Z planteadas en los párrafos de introducción del examen, considere las siguientes definiciones de clases, escritas en pseudo–código:

```
class Abra {
   int a = X, b = Y

   fun cus(int x): int {
      a = b + x
      return pide(a)
   }

   fun pide(int y): int {
      return a - y * b
   }
}


class Cadabra extends Abra {
   Abra zo = new PataDeCabra()

   fun pide(int y): int {
      return zo.cus(a + b) - y
   }
}


class PataDeCabra extends Cadabra {
   int b = Y + Z, c = Z

   fun cus(int x): int {
      a = x - 3
      c = a + b * c
      return pide(a * b + x)
   }

   fun pide(int y): int {
      return c - y * a
}
```

Considere además el siguiente fragmento de código:

```
Abra ho = new Cadabra()
Abra po = new PataDeCabra()
Cadabra cir = new PataDeCabra()

print(ho.cus(1) + po.cus(1) + cir.cus(1))
```

Diga qué imprime el programa en cuestión si el lenguaje tiene:

(a) Asociación estática de métodos

(b) Asociación dinámica de métodos

Recuerde mostrar paso a paso el estado del programa (la pila de ejecución).

4. Se desea que modele e implemente, en el lenguaje de su elección, un manejador de tablas de métodos virtuales para un sistema orientado a objetos con herencia simple y despacho dinámico de métodos:

   (a) Debe saber tratar con definiciones de clases, potencialmente con herencia simple. Estas definiciones tendrán únicamente los nombres de los métodos que poseerá.

   (b) Una vez iniciado el programa, pedirá repetidamente al usuario una acción para proceder. Tal acción puede ser:

      i. `CLASS <tipo> [<nombre>]`
      Define un nuevo tipo que poseerá métodos con nombres establecidos en la lista proporcionada. El `<tipo>` puede ser:
      - Un nombre, que establece un tipo que no hereda de ningún otro.
      - Una expresión de la forma `<nombre> : <super>`, que establece el nombre del tipo y el hecho de que este tipo hereda del tipo con nombre `<super>`.

      Por ejemplo: `CLASS A f g` y `CLASS B : A f h`

      Notemos que es posible reemplazar definiciones de una super clase en clases que la heredan.
      El programa debe reportar un error e ignorar la acción si el nombre de la nueva clase ya existe, si la clase `super` no existe, si hay definiciones repetidas en la lista de nombres de métodos o si se genera un ciclo en la jerarquía de herencia.

      ii. `DESCRIBIR <nombre>`
      Debe mostrar la tabla de métodos virtuales para el tipo con el nombre propuesto.

      Por ejemplo: `DESCRIBIR B`

      Esto debe mostrar:

      ```
      f -> B :: f
      g -> A :: g
      h -> B :: h
      ```

      El programa debe reportar un error e ignorar la acción si el nombre del tipo no existe.

      iii. `SALIR`
      Debe salir del simulador.

   Al finalizar la ejecución de cada acción, el programa deberá pedir la siguiente acción al usuario.

   Investigue herramientas para pruebas unitarias y cobertura en su lenguaje escogido y agregue pruebas a su programa que permitan corroborar su correcto funcionamiento. Como regla general, su programa debería tener una cobertura (de líneas de código y de bifuración) mayor al 80%.

5. Considere las funciones `foldr` y `const`, escritas en un lenguaje muy similar a Haskell:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ e []     = e
foldr f e (x:xs) = f x $ foldr f e xs

const :: a -> b -> a
const x _ = x
```

Considere también la siguiente función que aplica una función solamente sobre la cola de una lista y agrupa la cabeza con otro valor dado:

```
what :: (a -> b -> c) -> a -> ([b] -> [c]) -> [b] -> [c]
what _ _ _ []     = []
what h x f (y:ys) = h x y : f ys
```

(a) Considere la siguiente implementación de una función `misteriosa`, usando `foldr`:

```
misteriosa :: ???
misteriosa f = foldr (what f) (const [])
```

Considere también la siguiente función, que genera una lista de números enteros a partir de un cierto valor inicial:

```
gen :: Int -> [Int]
gen n = n : gen (n * 2)
```

Muestre la evaluación, paso a paso, de la expresión `misteriosa (+) [1..3] (gen 1)`, considerando que:

i. El lenguaje tiene orden de evaluación *normal*.

ii. El lenguaje tiene orden de evaluación *aplicativo*.

(b) Considere el siguiente tipo de datos que representa árboles binarios con información en las ramas:

```
data Arbol a = Hoja | Rama a (Arbol a) (Arbol a)
```

Construya una función `foldA` (junto con su firma) que permita reducir un valor de tipo `(Arbol a)` a algún tipo `b` (de forma análoga a `foldr`). Su implementación debe poder tratar con estructuras potencialmente infinitas.

Su función debe cumplir con la siguiente firma:

```
foldA :: (a -> b -> b -> b) -> b -> Arbol a -> b
```

(c) Considere una versión de la función `what` que funciona sobre árboles (aplica la función proporcionada a ambos sub–árboles) y llamésmola *what tree function*:

```
whatTF :: (a -> b -> c)
       -> a
       -> (Arbol b -> Arbol c)
       -> (Arbol b -> Arbol c)
       -> Arbol b
       -> Arbol c
whatTF _ _ _ _ Hoja         = Hoja
whatTF h x f g (Rama y i d) = Rama (h x y) (f i) (g d)
```

Usando su función `foldA` definimos la función `sospechosa`:

```
sospechosa :: ???
sospechosa f = foldA (whatTF f) (const Hoja)
```

Definimos también la siguiente función, que genera un árbol de números enteros a partir de un cierto valor inicial:

```
genA :: Int -> Arbol Int
genA n = Rama n (genA (n * 2)) (genA (n * 3))
```

Finalmente, definimos el valor `arbolito` como una instancia de `Arbol Char`:

```
arbolito :: Arbol Char
arbolito = Rama '1' (Rama '2' Hoja (Rama '3' Hoja Hoja)) Hoja
```

Muestre la evaluación, paso a paso, de la expresión `sospechosa (+) arbolito (genA 1)`, considerando que:

i. El lenguaje tiene orden de evaluación *normal*.

ii. El lenguaje tiene orden de evaluación *aplicativo*.

Si sospecha que en algún momento uno de estos programas puede caer en una evaluación recursiva infinita, realice las primeras expansiones, detenga la evaluación y argumente las razones por las que cree que dicha evaluación no terminaría.

6. Se desea que modele e implemente, en el lenguaje de su elección, un intérprete para un subconjunto del lenguaje `Prolog`:

   (a) Debe saber manejar hechos, reglas y consultas. Todas estas estarán formadas por expresiones, que pueden adoptar una de las siguientes formas:

      i. **Átomo:** Cualquier cadena alfanumérica que empiece con un caracter en minúscula. Por ejemplo:
      `hola`
      `ci3641`
      `quickSort`

      ii. **Variable:** Cualquier cadena alfanumérica que empiece con un caracter en mayúscula. Por ejemplo:
      `Hola`
      `CI3641`
      `QuickSort`

      iii. **Estructura:** Un átomo, seguido de una secuencia, parentizada y separada por comas, de otras expresiones. Por ejemplo:
      `f(x, y)`
      `quickSort(entrada, Salida)`
      `in(c(e,P), t(i, o(N)))`

   (b) Una vez iniciado el programa, pedirá repetidamente al usuario una acción para proceder. Tal acción puede ser:

      i. `DEF <expresion> [<expresion>]`
      Define un nuevo hecho o regla, representado por la primera `<expresion>` (en el formato previamente establecido).
      Si la lista de expresiones (después de la primera) es vacía, la definición corresponde a un **hecho**.

      Por ejemplo: `DEF padre(juan, jose)` y `DEF true`

      Si la lista de expresiones (después de la primera) no es vacía, la definición corresponde a una **regla**, donde la primera expresión es el consecuente y el resto son los antecedentes.

      Por ejemplo: `DEF abuelo(X, Y) padre(X, Z) padre(Z, Y)`

      El programa debe reportar un error e ignorar la acción si el functor asociado a alguna de las expresiones involucradas (el nombre del predicado) no tiene forma de átomo, si las listas de argumentos no están bien formadas o si alguna de las expresiones es una variable fuera de alguna estructura.

      ii. `ASK <expresion>`
      Realiza una consulta por el predicado representado en `<expresion>`.

      Por ejemplo: `ASK abuelo(juan, X)`

      De ser satisfacible, debe reportar el triunfo y el conjunto de unificaciones para las variables involucradas. Luego, debe ofrecer dos opciones al usuario:

      - **aceptar:** La búsqueda termina, ya que el usuario se da por satisfecho con el resultado obtenido.
      - **rechazar:** La búsqueda continúa, ya que el usuario no está satisfecho aún con el resultado obtenido.

De no ser satisfacible, debe reportar que dicha consulta ha fallado.
El programa debe reportar un error e ignorar la acción si el functor asociado a la expresiones (el nombre del predicado) no tiene forma de átomo, si la lista de argumentos no está bien formada o si la expresión es una variable fuera de alguna estructura.

iii. `SALIR`
Debe salir del simulador.

Al finalizar la ejecución de cada acción, el programa deberá pedir la siguiente acción al usuario.

Consideremos un ejemplo un poco más elaborado para comprender el funcionamiento del programa:

```
$> DEF padre(juan, jose)
 Se definió el hecho 'padre(juan, jose)'
$> DEF padre(jose, pablo)
 Se definió el hecho 'padre(jose, pablo)'
$> DEF padre(pablo, gaby)
 Se definió el hecho 'padre(pablo, gaby)'
$> DEF ancestro(X, Y) padre(X, Y)
 Se definió la regla 'ancestro(X, Y) :- padre(X, Y)'
$> DEF ancestro(X, Y) padre(X, Z) ancestro(Z, Y)
 Se definió la regla 'ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y)'
$> ASK ancestro(X, gaby)
 Satisfacible, cuando 'X = pablo'. ¿Qué desea hacer?
$[consultando ancestro(X, gaby)]> RECHAZAR
 Satisfacible, cuando 'X = jose'. ¿Qué desea hacer?
$[consultando ancestro(X, gaby)]> RECHAZAR
 Satisfacible, cuando 'X = juan'. ¿Qué desea hacer?
$[consultando ancestro(X, gaby)]> RECHAZAR
 No es satisfacible.
$> ASK ancestro(gaby, X)
 No es satisfacible
$> ASK ancestro(X, Y)
 Satisfacible, cuando 'X = juan, Y = jose'. ¿Qué desea hacer?
$[consultando ancestro(X, Y)]> RECHAZAR
 Satisfacible, cuando 'X = jose, Y = pablo'. ¿Qué desea hacer?
$[consultando ancestro(X, Y)]> RECHAZAR
 Satisfacible, cuando 'X = juan, Y = pablo'. ¿Qué desea hacer?
$[consultando ancestro(X, Y)]> ACEPTAR
 Consulta aceptada
```

Investigue herramientas para pruebas unitarias y cobertura en su lenguaje escogido y agregue pruebas a su programa que permitan corroborar su correcto funcionamiento. Como regla general, su programa debería tener una cobertura (de líneas de código y de bifuración) mayor al 80%.

7. **RETO EXTRA: *¡VELOCISTA!***

Considere la misma función *evil*, definida en el parcial anterior:

$$evil(n) = fib\left(\lfloor \log_2(B_{n+1}) \rfloor + 1\right)$$

Queremos un programa que permita calcular valores para $evil(n)$ de la forma más eficiente posible, para valores lo más grandes posibles de n.

Desarrolle un programa, en el lenguaje de su elección, que:

- Reciba por *la entrada estándar* un valor para n, tal que $n \geq 0$ (esto puede suponerlo, no tiene que comprobarlo).
- Imprima el valor de $evil(n)$.

Su programa debe imprimir el valor correcto y será probado con valores más y más grandes de n (no necesariamente acotado por la precisión disponible para un numero entero de 32 o 64 bits).

Reglas del reto: Intente desarrollar su programa de tal forma que pueda manejar valores grandes de n y responder eficientemente. Se impondrá un *timeout* de 1 segundo para cada invocación. Ganará quien pueda imprimir la respuesta correcta para el n más grande posible en ese tiempo.

- El ganador del reto tendrá 5 puntos extras.
- El segundo lugar tendrá 3 puntos extras.
- El tercer lugar tendrá 1 punto extra.

Para evitar que se listen los valores de evil(n), su programa debe pesar a lo sumo 32KB (en otras palabras, 2^{15} bytes).