Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 – Lenguajes de Programación I
Enero–Marzo 2021

Examen 2

(25 puntos)

A continuación encontrará 5 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 17 de Marzo de 2021.

1. Escoja algún lenguaje de programación de alto nivel y de propósito general cuyo nombre empiece con la misma letra que su apellido (por ejemplo, si su apellido es "Rodríguez", podría escoger "Ruby", "Rust", "R", etc.).

   (a) De una breve descripción del lenguaje escogido.

      i. Diga qué tipos de datos posee y qué mecanismos ofrece para la creación de nuevos tipos (incluyendo tipos polimórficos de haberlos).

      ii. Describa el funcionamiento del sistema de tipos del lenguaje, incluyendo el tipo de equivalencia para sus tipos, reglas de compatibilidad y capacidades de inferencia de tipos.

      iii. Explique los diferentes tipos de rutinas que el lenguaje ofrezca, así como los diferentes tipos de pasaje de parámetros.

      iv. Detalle el mecanismo de manejo de excepciones del lenguaje. Si no tiene uno, explique cómo un programador usual del mismo haría para tener una alternativa a esto (por ejemplo, la pareja `setjmp`/`longjmp` de C).

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

i. Defina un árbol binario con información en ramas y hojas. A continuación un ejemplo en Haskell:

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

Sobre este árbol, defina una función `esDeBusqueda` que diga si el árbol en cuestión es un árbol de búsqueda o no.
Recuerde que un árbol binario de búsqueda es un árbol binario tal que, para cada rama:

- El valor almacenado en la misma es *mayor o igual* que todos los valores que se encuentran en el sub-árbol *izquierdo*.
- El valor almacenado en la misma es *menor o igual* que todos los valores que se encuentran en el sub-árbol *derecho*.

ii. Defina un tipo de datos recursivo que represente numerales de Church. A continuación un ejemplo en Haskell:

```
data Church = Cero | Suc Church
```

Recuerde que un numeral de Church se construye a partir de:

- Un valor constante que representa al cero.
- Una función *sucesor* que, para cualquier número n, devuelve $n + 1$.

Sobre este tipo se desea que implemente las funciones `suma` y `multiplicación`.

2. Considere el siguiente programa escrito en pseudo-código:

```
int tu = X + Y, que = Y + Z;

proc lagarto(int x, int y, int z) {
   x = x + y - z
   y = z * 2 - x
   z = x + y - 1
}

lagarto(tu, tu, que)
lagarto(tu, que, que)
lagarto(que, tu, tu)

print(tu, que)
```

Note que deberá reemplazar los valores para X, Y y Z como fue explicado en los párrafos de introducción del examen.

Diga qué imprime el programa en cuestión, con cada combinación de *referencia* o *valor/resultado* para los parámetros `x`, `y` y `z`. Esto es:

(a) `x` es pasado por referencia, `y` es pasado por referencia, `z` es pasado por referencia

(b) `x` es pasado por referencia, `y` es pasado por referencia, `z` es pasado por valor/resultado

(c) `x` es pasado por referencia, `y` es pasado por valor/resultado, `z` es pasado por referencia

(d) `x` es pasado por referencia, `y` es pasado por valor/resultado, `z` es pasado por valor/resultado

(e) `x` es pasado por valor/resultado, `y` es pasado por referencia, `z` es pasado por referencia

(f) `x` es pasado por valor/resultado, `y` es pasado por referencia, `z` es pasado por valor/resultado

(g) `x` es pasado por valor/resultado, `y` es pasado por valor/resultado, `z` es pasado por referencia

(h) `x` es pasado por valor/resultado, `y` es pasado por valor/resultado, `z` es pasado por valor/resultado

Recuerde mostrar los pasos de su ejecución (por lo menos al nivel de cada llamada a `lagarto`).

Puede suponer que este lenguaje empila sus parámetros de *izquierda* a *derecha*.

3. Se desea que modele e implemente, en el lenguaje de su elección, un programa que simule un manejador de tipos de datos. Este programa debe cumplir con las siguientes características:

   (a) Debe saber manejar tipos atómicos, registros (`struct`) y registros variantes (`union`).

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

      i. `ATOMICO <nombre> <representación> <alineación>`
      Define un nuevo tipo atómico de nombre `<nombre>`, cuya representación ocupa `<representación>` bytes y debe estar alineado a `<alineación>` bytes.

      Por ejemplo: `ATOMICO char 1 2` y `ATOMICO int 4 4`

      El programa debe reportar un error e ignorar la acción si `<nombre>` ya corresponde a algún tipo creado en el programa.

      ii. `STRUCT <nombre> [<tipo>]`
      Define un nuevo registro de nombre `<nombre>`. La definición de los campos del registro viene dada por la lista en `[<tipo>]`. Nótese que los campos no tendrán nombres, sino que serán representados únicamente por el tipo que tienen.

      Por ejemplo: `STRUCT foo char int`

      El programa debe reportar un error e ignorar la acción si `<nombre>` ya corresponde a algún tipo creado en el programa o si alguno de los tipos en `[<tipo>]` no han sido definidos.

      iii. `UNION <nombre> [<tipo>]`
      Define un nuevo registro variante de nombre `<nombre>`. La definición de los campos del registro variante viene dada por la lista en `[<tipo>]`. Nótese que los campos no tendrán nombres, sino que serán representados únicamente por el tipo que tienen.

      Por ejemplo: `UNION bar int foo int`

      El programa debe reportar un error e ignorar la acción si `<nombre>` ya corresponde a algún tipo creado en el programa o si alguno de los tipos en `[<tipo>]` no han sido definidos.

      iv. `DESCRIBIR <nombre>`
      Debe dar la información correspondiente al tipo con nombre `<nombre>`. Esta información debe incluir, tamaño, alineación y cantidad de bytes desperdiciados para el tipo, si:
      - El lenguaje guarda registros y registros viariantes *sin empaquetar*.
      - El lenguaje guarda registros y registros viariantes *empaquetados*.
      - El lenguaje guarda registros y registros viariantes *reordenando los campos de manera óptima* (minimizando la memoria, sin violar reglas de alineación).

      El programa debe reportar un error e ignorar la acción si `<nombre>` no corresponde a algún tipo creado en el programa.

      v. `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%.

4. Tomando como referencia las constantes X, Y y Z planteadas en los párrafos de introducción del examen, definamos:

   - $A = 2 \times ((X + Y) \bmod 5) + 3$
   - $B = 2 \times ((Y + Z) \bmod 5) + 3$

   Tomando en cuentas las constantes A y B, considere las siguientes definiciones de co–rutinas escritas en pseudo-código:

```
coroutine w():
   int a = 0
   loop:
      a = a + A
      print(a)
      if a mod 2 == 0:
         transfer t()
      else:
         transfer f()
```

```
coroutine t():
   int b = 1
   loop:
      b = (b + 1) * B
      print(b)
      if b mod 2 == 0:
         transfer t()
      else:
         transfer f()
```

```
coroutine f():
   int c = 1
   loop:
      c = c + 1
      print(c)
      if c mod 2 == 0:
         transfer w()
      else:
         transfer t()
```

   Suponga que el programa inicial con una llamada `transfer w()`.

   Se desea que ejecute el programa anterior

   (a) Muestre paso a paso el estado del programa (puede utilizar el mismo tipo de corridas en frío que usamos para *iteradores*, donde el *program counter* o `pc` es almacenado como una variable local más de la rutina).

   (b) Diga cuáles son los primeros 10 valores que imprime este programa. Una vez haya impreso el décimo valor, puede detener la ejecución del programa (aún si la ejecución pudiera continuar).

5. Se desea que modele e implemente, en el lenguaje de su elección, un programa que maneje tipos de datos polimórficos. Este programa debe cumplir con las siguientes características:

   (a) Debe saber tratar expresiones, con la siguiente estructura:

   - **Átomo**: Representada por una cadena de caracteres alfa–numérica.
     Por ejemplo: `f`
   - **Aplicación funcional**: Representada por la yuxtaposición de dos expresiones.
     Por ejemplo: `f x`
   - **Expresión parentizada**: Representada por una expresión encerrada entre paréntesis (útil para organizar aplicaciones funcionales).
     Por ejemplo: `f (f x)`

   (b) Debe saber tratar tipos, con la siguiente estructura:

   - **Constantes de tipo**: Representado por una cadena de caracteres alfa–numérica que empieza con una letra mayúscula.
     Por ejemplo: `T`
   - **Variables de tipo**: Representado por una cadena de caracteres alfa–numérica que empieza con una letra minúscula.
     Por ejemplo: `t`
   - **Funciones**: Representado por dos tipos unidos con el operador (`->`). El primero de estos tipos representa el *dominio* y el segundo representa la *imagen*.
     Por ejemplo: `t -> T`
   - **Tipos parentizados**: Representado por un tipo encerrado entre paréntesis (útil para organizar funciones).
     Por ejemplo: `(a -> a) -> a`

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

   i. `DEF <nombre> <tipo>`
   Representa una definición del nombre en `<nombre>`, como un átomo que tiene asociado el tipo en `<tipo>`.
   El programa puede redefinir nombres, por lo que no se debe verificar si ya existe una asociación para `<nombre>`.
   Por ejemplo:
   - `DEF x T` define `x` teniendo tipo `T`.
   - `DEF f t -> T` define `f` teniendo tipo `t -> T`.
   - `DEF g (a -> a) -> a` define `g` teniendo tipo `(a -> a) -> a`.

   ii. `TIPO <expr>`
   Consulta el tipo de la expresión en `<expr>`, realizando la unificación necesaria y construyendo el tipo más general posible.
   Por ejemplo, considerando las definiciones en la sección anterior:
   - `TIPO f` deberá imprimir `t -> T` (o usando cualquier otra variable de tipo).
   - `TIPO f x` deberá imprimir `T`.
   - `TIPO g f` deberá imprimir `T`.

   El programa debe indicar que ha ocurrido un error si alguno de los nombres presentes en `<expr>` no está definido o si a la expresión no se le puede calcular un tipo consistente.

   iii. `SALIR`
   Debe salir del programa.

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 0 Int
 Se definió '0' con tipo Int
$> DEF 1 Int
 Se definió '1' con tipo Int
$> DEF n Int
 Se definió 'n' con tipo Int
$> DEF eq a -> a -> Bool
 Se definió 'eq' con tipo a -> a -> Bool
$> TIPO eq 0
 Int -> Bool
$> TIPO eq 2
 ERROR, el nombre '2' no ha sido definido
$> DEF prod Int -> Int -> Int
 Se definió 'prod' con tipo Int -> Int -> Int
$> DEF dif Int -> Int -> Int
 Se definió 'dif' con tipo Int -> Int -> Int
$> DEF if Bool -> a -> a -> a
 Se definió 'if' con tipo Bool -> a -> a -> a
$> TIPO if (eq 0 n) 1 n
 Int
$> TIPO if (eq 0 n) 1 eq
 ERROR, no se pudo unificar Int con (a -> a -> Bool)
$> TIPO if (eq 0 n) if
 (Bool -> a -> a -> a) -> Bool -> a -> a -> a
$> DEF fact t
 Se definió 'fact' con tipo t
$> TIPO eq (fact n) (if (eq n 0) 1 (prod n (fact (dif n 1))))
 Bool
$> TIPO eq fact (if (eq n 0) 1 (prod n (fact (dif n 1))))
 ERROR, no se pudo unificar Int con (Int -> Int)
```

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%.

6. **RETO EXTRA: *¡POLIGLOTA!***

Considere la misma función $jaweno$, definida en el parcial anterior:

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

Decimos que un programa es *políglota* si el mismo código fuente puede ser compilado/interpretado por al menos dos diferentes lenguajes de programación.

Desarrolle un programa políglota que:

- Reciba por argumentos del sistema un valor para n, tal que $n \geq 0$ (esto puede suponerlo, no tiene que comprobarlo).
- Imprima el valor de $jaweno(n)$.

Su programa debe imprimir el valor correcto y tomando menos de 1 segundo de ejecución, por lo menos hasta $n = 20$ en todos los lenguajes de programación considerados.

Reglas del reto: Intente desarrollar su programa de tal forma que la mayor cantidad de lenguajes de programación puedan compilarlo/interpretarlo. Debe indicar todos los lenguajes para los cuales su código fuente funciona y proporcionar instrucciones para ejecutarlo en cada uno de estos (que puede ser sencillamente un enlace a alguna herramienta online para interpretar el lenguaje, como tio.run o ideone.com)

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