

## Examen 2

(35 puntos)

A continuación encontrará 7 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 6 de Abril de 2022.

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. Enumere y explique las estructuras de control de flujo que ofrece.
- ii. Diga en qué orden evalúan expresiones y funciones
- iii. Diga qué tipos de datos posee y qué mecanismos ofrece para la creación de nuevos tipos (incluyendo tipos polimórficos de haberlos).
- iv. 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.

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

- i. 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`.

ii. 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 `esMinHeapBalanceado` que diga si el árbol en cuestión es un *heap* balanceado o no.

Recuerde que un *heap* es un árbol binario tal que, para cada rama, el valor almacenado en la misma es *menor o igual* que todos los valores que se encuentran en ambos sub-árbol *hijos*.

Consideraremos que un *heap* está *balanceado* sí y sólo si la diferencia entre las profundidades de todas las hojas (distancia desde la raíz hasta la hoja en cuestión) es de a lo sumo uno.

2. Se desea que modele e implemente, en el lenguaje de su elección, un programa que maneje expresiones aritméticas sobre enteros. Este programa debe cumplir con las siguientes características:

(a) Debe saber tratar expresiones escritas en orden *pre-fijo* y *post-fijo*, con los siguientes operadores:

- **suma:** Representada por el símbolo +.
- **resta:** Representada por el símbolo -.
- **multiplicación:** Representada por el símbolo \*.
- **división entera:** Representada por el símbolo /.

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

i. EVAL <orden> <expr>

Representa una *evaluación* de la expresión en <expr>, que está escrita de acuerdo a <orden>.

El <orden> solamente puede ser:

- PRE: Que representa expresiones escritas en orden *pre-fijo*.
- POST: Que representa expresiones escritas en orden *post-fijo*.

Por ejemplo:

- EVAL PRE + \* + 3 4 5 7 deberá imprimir 42.
- EVAL POST 8 3 - 8 4 4 + \* + deberá imprimir 69.

ii. MOSTRAR <orden> <expr>

Representa una *impresión en orden in-fijo* de la expresión en <expr>, que está escrita de acuerdo a <orden>.

El <orden> sigue el mismo patrón que en el punto anterior.

Su programa debe tomar la precedencia y asociatividad estándar, donde:

- La suma y la resta tienen la misma precedencia.
- La multiplicación y la división entera tienen la misma precedencia.
- La multiplicación y la división entera tienen mayor precedencia que la suma y la resta.
- Todos los operadores asocian a izquierda.

La expresión resultante debe tener la menor cantidad posible de paréntesis, de tal forma que la expresión mostrada como resultado tenga la misma semántica que la expresión que fue pasada como argumento a la acción.

Por ejemplo:

- MOSTRAR PRE + \* + 3 4 5 7 deberá imprimir (3 + 4) \* 5 + 7.
- MOSTRAR POST 8 3 - 8 4 4 + \* + deberá imprimir 8 - 3 + 8 \* (4 + 4).

iii. SALIR

Debe salir del programa.

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 bifurcación) mayor al 80%.

3. Considere los siguientes iteradores, escritos en Python:

(a) El iterador `ins`:

```
def ins(e, ls):
    yield [e, *ls]
    if ls:
        for i in ins(e, ls[1:]):
            yield [ls[0], *i]
```

Considere también el siguiente fragmento de código que hace uso del iterador `ins`:

```
for i in ins(0, [1, 2, 3]):
    print i
```

Ejecute, paso a paso, el fragmento de código mostrado (por lo menos al nivel de cada nuevo marco de pila creado) y muestre lo que imprime.

*Pista: Los elementos generados por este iterador serán listas.*

(b) El iterador `misterio`, que hace uso de `ins`:

```
def misterio(ls):
    if ls:
        for m in misterio(ls[1:]):
            for i in ins(ls[0], m):
                yield i
    else:
        yield []
```

Considere también el siguiente fragmento de código que hace uso del iterador `misterio`:

```
for m in misterio([1,2,3]):
    print m
```

i. Ejecute, paso a paso, el fragmento de código mostrado (por lo menos al nivel de cada nuevo marco de pila creado) y muestre lo que imprime.

*Pista: Los elementos generados por este iterador serán listas.*

ii. Explique, a grandes rasgos, cómo funciona el iterador `misterio` y qué colección de elementos conocida está generando. Diga cómo aprovecha el iterador `ins` para generar la colección deseada.

iii. Modifique el iterador `misterio` de tal forma que obtenga un nuevo iterador `suspense`, que devolverá cada elemento de las listas generadas por `misterio` de forma *aplanada*.

Por ejemplo, si al ejecutar `misterio` resultan generadas las listas:

- [a, b, c]
- [d]
- [e, f]

El nuevo iterador `suspense` deberá generar:

- a
- b
- c
- d
- e
- f

4. Considere la siguiente definición para una familia de funciones:

$$F_{\alpha,\beta}(n) = \begin{cases} n & \text{si } 0 \leq n < \alpha \times \beta \\ \sum_{i=1}^{\alpha} F_{\alpha,\beta}(n - \beta \times i) & \text{si } n \geq \alpha \times \beta \end{cases}$$

Notemos que  $F_{2,1}$  corresponde a la definición para los números de Fibonacci:

$$F_{2,1}(n) = \begin{cases} n & \text{si } 0 \leq n < 2 \\ F_{2,1}(n-1) + F_{2,1}(n-2) & \text{si } n \geq 2 \end{cases}$$

Como un segundo ejemplo,  $F_{3,4}$  corresponde a:

$$F_{3,4}(n) = \begin{cases} n & \text{si } 0 \leq n < 12 \\ F_{3,4}(n-4) + F_{3,4}(n-8) + F_{3,4}(n-12) & \text{si } n \geq 12 \end{cases}$$

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

- $\alpha = ((X + Y) \bmod 5) + 3$
- $\beta = ((Y + Z) \bmod 5) + 3$

Se desea que realice implementaciones, en el lenguaje *imperativo* de su elección:

- (a) Una subrutina recursiva que calcule  $F_{\alpha,\beta}$  para los valores de  $\alpha$  y  $\beta$  obtenidos con las fórmulas mencionadas anteriormente. Esta implementación debe ser una traducción directa de la fórmula resultante a código.
- (b) Una subrutina recursiva **de cola** que calcule  $F_{\alpha,\beta}$ .
- (c) La conversión de la subrutina anterior a una versión iterativa, mostrando claramente cuáles componentes de la implementación recursiva corresponden a cuáles otras de la implementación iterativa.

Debe usar el mismo el lenguaje para estos tres ejercicios y asegurarse que su lenguaje tenga las estructuras de control de flujo necesarias para realizarlos (su lenguaje escogido debe, por tanto, ser imperativo).

Realice también un análisis comparativo entre las tres implementaciones realizadas, mostrando tiempos de ejecución para diversos valores de entrada y ofreciendo conclusiones sobre la eficiencia. Es recomendable que se apoye en herramientas de visualización de datos (como los *plots* de Matlab, R, Octave, Excel, etc.)

5. 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 arreglos de tamaño estático.
- (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. **ARREGLO** <nombre> <tipo> <tamaño>

Define un nuevo alias para arreglos de <tipo>, con <tamaño> elementos, de nombre <nombre>.

Por ejemplo: **ARREGLO** farray foo 10

El programa debe reportar un error e ignorar la acción si <nombre> ya corresponde a algún tipo creado en el programa, si <tipo> no ha sido definido o <tamaño> no corresponde con un número entero no-negativo.

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 arreglos *sin empaquetar*.
- El lenguaje guarda registros *empaquetados* y arreglos *sin empaquetar*.
- El lenguaje guarda registros y arreglos *empaquetados*.
- El lenguaje guarda registros *reordenando los campos de manera óptima* (minimizando la memoria, sin violar reglas de alineación) y arreglos *sin empaquetar*..

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%.

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

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

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

<pre>coroutine w():   int a = 0   loop:     a = a + A     print(a)     if a mod 3 == 0:       transfer t()     else:       transfer f()</pre>	<pre>coroutine t():   int b = 1   loop:     b = (b + 1) * B     print(b)     if b mod 3 == 1:       transfer w()     else:       transfer f()</pre>	<pre>coroutine f():   int c = 1   loop:     c = 2 * (c + C)     print(c)     if c mod 3 == 2:       transfer w()     else:       transfer t()</pre>
---	---	---

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

Se desea que ejecute el programa anterior

- 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).
- 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).

7. Se desea que modele e implemente, en el lenguaje de su elección, un programa que simule un manejador de memoria dinámica con liberación explícita. Este programa debe cumplir con las siguientes características:
- (a) Debe saber manejar apuntadores generales (cuyo tipo apuntado no es relevante).
  - (b) Puede implementar el manejador ya sea con *lápidas* o con *llaves y cerraduras*. En cualquier caso, debe reportar, al iniciar la ejecución del programa, la alternativa con la cual realizó su implementación.
  - (c) Una vez iniciado el programa, pedirá repetidamente al usuario una acción para proceder. Tal acción puede ser:
    - i. **RESERVAR** <nombre> <valor>  
Define un nuevo apuntador <nombre> que apunta a una dirección de memoria recién reservada. Esta, a su vez, contiene un **valor** (que puede ser cualquier cadena de caracteres alfanumérica)  
  
Por ejemplo: `RESERVAR hotel trivago`  
  
Esta acción es análogo a ejecutar, en lenguaje C, instrucciones de la forma:  

```
<nombre> = malloc(...);  
*<nombre> = <valor>;
```

  
Nótese que los argumentos de malloc no son relevantes en este contexto.
    - ii. **ASIGNAR** <nombre1> <nombre2>  
Asigna al apuntador <nombre1> el apuntador <nombre2>. Esto crea un alias entre ambos apuntadores.  
  
Por ejemplo: `ASIGNAR lva rval` y `ASIGNAR whynot whynot`  
  
Esta acción es análogo a ejecutar, en lenguaje C, una instrucción de la forma:  

```
<nombre1> = <nombre2>;
```
    - iii. **LIBERAR** <nombre>  
Libera el espacio ocupado por el apuntador <nombre>.  
  
Por ejemplo: `LIBERAR awilly`  
  
Esta acción es análogo a ejecutar, en lenguaje C, una instrucción de la forma:  

```
free(<nombre>;
```

  
El programa debe reportar un error e ignorar la acción si <nombre> no tiene asociado valor alguno.
    - iv. **IMPRIMIR** <nombre>  
Si el espacio apuntado por <nombre> tiene un valor asignado, debe imprimir el mismo.  
  
Por ejemplo: `IMPRIMIR fotocopias`  
  
Esta acción es análogo a ejecutar, en lenguaje C (suponiendo que "%?" corresponde al patrón de impresión para los valores manejados), una instrucción de la forma:  

```
printf("%?\n", *<nombre>;
```

  
El programa debe reportar un error e ignorar la acción si <nombre> no tiene asociado valor alguno.
    - v. **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:

```
$> RESERVAR a aval
  Se reservó 'a' con valor 'aval'
$> RESERVAR b bval
  Se reservó 'b' con valor 'bval'
$> IMPRIMIR a
  aval
$> ASIGNAR c b
  Se asignó 'b' a 'c'
$> IMPRIMIR c
  bval
$> IMPRIMIR d
  ERROR, el nombre 'd' no apunta a un valor válido
$> LIBERAR c
  Se liberó 'c'
$> IMPRIMIR b
  ERROR, el nombre 'b' no apunta a un valor válido
$> ASIGNAR b a
  Se asignó 'a' a 'b'
$> IMPRIMIR b
  aval
$> IMPRIMIR c
  ERROR, el nombre 'c' no apunta a un valor válido
```

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%.

## 8. RETO EXTRA: ¡POLÍGLOTA!

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

$$ohno(n) = \lfloor \log_2(C_{trib}(C_{n+1})) \rfloor$$

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  $ohno(n)$ .

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](http://tio.run) o [ideone.com](http://ideone.com))

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