Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Enero-Marzo 2013

Carnet:_____

Nombre:_____

Examen III

(40 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	**Total**
27 puntos	5 puntos	8 puntos	**40 puntos**

Pregunta 1 - 27 puntos

Esta pregunta consta de nueve (9) subpreguntas de selección, numeradas de 1.1 a 1.9. Cada una de las subpreguntas viene acompañada de cuatro posibles respuestas (a, b, c, d), entre las cuales sólo **una** es correcta. Ud. deberá marcar completamente y sin posibilidad de confusión la opción que considere correcta o, si desea no contestar la subpregunta, marcar completamente y sin posibilidad de confusión la última opción (e) que indica que Ud. prefiere omitir la respuesta.
Cada una de las subpreguntas tiene un valor de tres (3) puntos. Tres subpreguntas incorrectas eliminan una correcta. Las subpreguntas omitidas (marcadas en la opción e) no suman ni restan puntos.

1.1. Considere un lenguaje con alcance estático en el que se permite anidar subrutinas. El compilador está generando código para la rutina `foo` justo para una línea en la cual se invoca a la rutina `bar`. Suponiendo que `bar` está anidada en `foo`, entonces el compilador

a) Genera código para recorrer la cadena estática de `foo` y usar la última dirección encontrada como cadena estática de `bar`.

b) Genera código para que la cadena estática de `bar` sea igual a la cadena dinámica de `foo`.

c) Genera código para que la cadena estática de `bar` apunte directamente al llamador.

d) No tiene nada que hacer, porque la cadena estática se maneja en el prólogo del llamado.

e) No sabe / No contesta.

1.2. Considere alguna operación particular relacionada con la secuencia de llamadas que puede realizarse indistintamente tanto en el llamador como en el llamado. Entonces,

a) Es preferible hacerla en el llamador, porque el código resultante será más rápido.

b) Es preferible hacerla en el llamador, porque el código resultante será más corto.

c) Es preferible hacerla en el llamado, porque el código resultante será más rápido.

d) Es preferible hacerla en el llamado, porque el código resultante será más corto.

e) No sabe / No contesta.

1.3 ¿Cuál de las siguientes afirmaciones es cierta con respecto al manejo de excepciones en un lenguaje que no tiene manejo automático de memoria?

a) Es irrelevante mantener la pila en un estado consistente, pues la excepción siempre provocará que se aborte la ejecución.

b) Es relevante mantener la pila en un estado consistente, y para eso es obligatorio que el llamador desmantele el registro de activación del llamado antes de propagar la excepción.

c) Es relevante mantener la pila en un estado consistente, y para eso es obligatorio que el llamado desmantele el registro de activación antes de propagar la excepción.

d) Es irrelevante preocuparse por los valores de la pila, pues lo único importante es mantener la cadena dinámica y estática, cosa que puede hacer el llamador al recibir la excepción propagada.

e) No sabe / No contesta.

1.4. Considere el siguiente programa escrito en pseudocódigo

```
proc grok( foo, bar, baz : int )
  foo := foo + bar + baz;
  bar := foo + bar + baz;
  baz := foo + bar + baz;
end

begin
  qux : int;
  qux := 7;

  grok(qux,qux,qux)
  print(qux)
end
```

Suponga que `foo` y `baz` son pasados por valor, pero `bar` es pasado por copia ("valor/resultado"). ¿Cuál sería la salida del programa en este caso?

a) 7
b) 35
c) 42
d) 63
e) No sabe / No contesta.

1.5. Continúe considerando el programa de la pregunta 1.4, pero ahora suponga que `foo` y `bar` son pasados por referencia, mientras que `baz` es pasada por valor.
¿Cuál sería la salida del programa en este caso?

a) 21
b) 42
c) 49
d) 105
e) No sabe / No contesta.

1.6. Tanto en Java como en C++, es posible declarar

```
Foo x;
```

donde `Foo` es una clase *concreta*.
En C++ el constructor `Foo()` será invocado automáticamente tan pronto se ingrese al alcance de dicha declaración, sin embargo en Java el constructor sólo es invocado si se ejecuta explícitamente `new Foo()`. Esta diferencia se debe a que:

a) La inicialización de objetos en Java es opcional, y si el programador no lo hace, el compilador genera la llamada automáticamente.
b) Esa declaración corresponde a un valor concreto que debe ser inicializado inmediatamente.
c) Eventualmente se va a ejecutar el destructor, cuya precondición es ejecutar el constructor.
d) Ninguna de las anteriores.
e) No sabe / No contesta.

1.7. Considere un lenguaje orientado a objetos en el que la herencia corresponde a la noción de subtipo y las variables son manejadas con modelo de valor. Suponga que en ese lenguaje se tiene una jerarquía de tres clases `Baz` heredando de `Bar` a su vez heredando de `Foo`.
En un programa del lenguaje se declara

```
proc grok( b0 : Bar, var b1 : Bar )
```

de manera que `b0` está siendo pasada por valor, mientras que `b1` está siendo pasada por referencia. El compilador encuentra la llamada

```
grok( q0, q1 )
```

¿en cuál de los siguientes casos el compilador permitirá la llamada?

a) Si `q0` es un `Bar` y `q1` es un `Foo`.
b) Si `q0` es un `Foo` y `q1` es un `Baz`.
c) Si `q0` es un `Baz` y `q1` es un `Foo`.
d) Ninguna de las anteriores.
e) No sabe / No contesta.

1.8 Considere las siguientes declaraciones en un lenguaje orientado a objetos que maneja las variables con modelo de valor y en el que la herencia corresponde a la noción de subtipo. Suponga que en ese lenguaje se tiene una jerarquía de dos clases `Baz` subclase de `Bar`, en la que `Bar` es abstracta y `Baz` es concreta. Considere las siguientes declaraciones

I. `Bar m1()`
II. `Baz Bar::m2(Bar *p)`
III. `Bar *Baz::m3(Baz *q, Baz *(r)())`
IV. `Bar *baz`
V. `Bar::Bar(Baz b)`

¿Cuales de las declaraciones son **inválidas**?

a) Sólo I y V.
b) Sólo I y IV.
c) Sólo II, IV y V.
d) Sólo III y V.
e) No sabe / No contesta.

1.9 En relación a las corutinas y los iteradores, se puede decir que

a) Las corutinas son un caso particular de los iteradores.
b) Los iteradores son un caso particular de las corutinas.
c) Siempre se pueden implantar usando la pila de ejecución del sistema.
d) Las corutinas son necesariamente concurrentes pero los iteradores nunca.
e) No sabe / No contesta.

Pregunta 2 - 5 puntos

Utilizaremos listas Haskell para representar polinomios en x. De este modo, un polinomio cualquiera como

$$a_n \times x^n + a_{n-1} \times x^{n-1} + \cdots + a_1 \times x + a_0$$

será representado por su lista de coeficientes, desde el más significativo hasta el menos significativo,

$$[a_n, a_{n-1}, \ldots, a_1, a_0]$$

Escriba la función

```
poly :: Num a => a -> [a] -> a
```

que *evalúe* el polinomio representado por el segundo argumento, usando al primer argumento como valor de la variable x. Esto es,

```
> poly 2 [4, -1, 0, 14]
42
> poly 2.0 [4, -1, 0, 14]
42.0
```

pues siendo $p(x) = 4x^3 - x^2 + 14$ entonces $p(2) = 42$.

Su solución **debe** utilizar funciones de orden superior y **no puede** utilizar `length`. Soluciones recursivas directas o que usen `length`, no obtendrán puntaje alguno.

Una solución basada en `foldr` – sumamente ineficiente en espacio

```
poly x cs = fst $ foldr addp (0,0) cs
  where
    addp n (q,l)  =  (n*x^l + q, l+1)
```

Una solución basada en `foldl'` – eficiente en espacio y tiempo

```
poly x cs = foldl' (\a c -> x * a + c) 0 cs
```

conocido como la Regla de Horner y que generalmente se aprende en bachillerato.

Pregunta 3 - 8 puntos

Prolog ofrece una variedad de predicados útiles para metaprogramación, que permiten la descomposición o construcción de functores a tiempo de ejecución, con la intención de agregarlos como predicados. En implantaciones clásicas de Prolog, se cuenta con los predicados

- `arg(Pos,Functor,Arg)` – triunfa si `Arg` es el `Pos`-ésimo argumento del functor `Functor`, comenzando a contar desde uno, i.e.

  ```
  ?- arg(2,foo(bar,42,[baz,69]),X).
  X = 42
  yes
  ?- arg(2,foo(bar,X,[baz,69]),42).
  X = 42
  yes
  ```

  El argumento que indica la posición *siempre* debe estar instanciado a un número, pero el predicado puede usarse tanto para determinar como para establecer el argumento posicional aprovechando la unificación. Si se suministra una posición menor a uno o mayor a la cantidad de argumentos en el functor, el predicado falla con un error.

- `functor(Functor,Name,NumArgs)` – triunfa si `Functor` es un functor cuyo átomo es `Name` y tiene `NumArgs` argumentos, comenzando a contar desde uno, i.e.

  ```
  ?- functor(foo(bar,42),Name,NumArgs).
  Name = foo
  NumArgs = 2
  yes
  ?- functor(F,foo,2).
  F = foo(_,_)
  yes
  ```

  Si el primer argumento está instanciado, los otros dos estarán unificados con la descomposición (nombre y número de argumentos), pero si el primer argumento no está instanciado los otros dos **deben** estar instanciados y sirven para construir un nuevo functor.

Las implantaciones más modernas de Prolog comenzaron a proveer el predicado `univ/2`, disponible a través del operador `=..` que permite convertir entre functores y listas.

```
?- foo(bar,42,[meh,69]) =.. L.
L = [foo,bar,42,[meh,69]]
yes
?- F =.. [foo,bar,42,[meh,69]].
F = foo(bar,42,[meh,69])
yes
?- foo =.. L.
L = [foo]
yes
?- F =.. [foo].
F = foo
yes
```

Cuando el primer argumento está instanciado con un functor, el segundo unificará a una lista cuyo primer elemento es el átomo correspondiente al functor, y los argumentos en el resto de la lista conservando el orden posicional. Simétricamente, cuando el primer argumento no está instanciado, el segundo *debe* estarlo con una lista de al menos un elemento.
Se desea que Ud. implante el predicado `uni/2`

```
uni(Functor,List)
```

que se comporte como el predicado `univ/2`, para lo cual puede usar los predicados `functor/3`, `arg/3` y cualquier otro predicado estándar de Prolog (`length/2`, `reverse/2`, `is/2`, etc.).

```
?- uni(foo(bar,42,[meh,69]),L).
L = [foo,bar,42,[meh,69]]
yes
?- uni(F,[foo,bar,42,[meh,69]]).
F = foo(bar,42,[meh,69])
yes
?- uni(foo,L).
L = [foo]
yes
?- uni(F,[foo]).
F = foo
yes
```

Note que el predicado `univ/2` **nunca** tiene más de una solución, de manera que su implantación de `uni/2` debe controlar el *backtracking* de manera apropiada para reflejar ese comportamiento.

```
% Si el predicado triunfa, debe impedir el backtracking, por eso el cut.
uni(Functor,[Name|Args]) :-
        length(Args,NumArgs),
        functor(Functor,Name,NumArgs),
        argList(1,Args,Functor), !.

% Comprueba que cada uno de los argumentos del functor coincida con las
% posiciones de la lista. Si la lista estuviera vacía, se llenará como
% consecuencia de la unificación. Notar que se usa recursión de cola para
% eficiencia, además de un cut en el caso base porque sólo hay una forma
% de construir la lista.
argList(_,[],_) :- !.
argList(Pos,[Arg|Args],Functor) :-
        arg(Pos,Functor,Arg),
        Next is Pos+1,
        argList(Next,Args,Functor).
```