Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Enero-Marzo 2015

Carnet:_____

Nombre:_____

Examen III

(40 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	Pregunta 4	**Total**
18 puntos	6 puntos	8 puntos	8 puntos	**40 puntos**

Pregunta 1 - 18 puntos

Esta pregunta consta de seis (6) subpreguntas de selección, numeradas de 1.1 a 1.6. 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. Cada una de las subpreguntas tiene un valor de tres (3) puntos. Tres subpreguntas incorrectas eliminan una correcta.

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

b) Genera código para regresar por la cadena estática de `foo` y usar la primera dirección encontrada como cadena estática de `bar`.

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

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

1.2. Considere las características mínimas de los lenguajes funcionales Scheme y Haskell comparados en clase, ¿cuál de las siguientes afirmaciones es **falsa**?

a) Ambos ofrecen mecanismos para meta-programación.

b) Solamente uno dispone de evaluación perezosa.

c) Solamente uno dispone de evaluación ambiciosa.

d) Tienen diferentes mecanismos de recolección de basura.

1.3. El libro de texto discute un mecanismo para manejo de excepciones que consiste en manter una pila de manejadores de excepciones a tiempos de ejecución. ¿Cuáles de las siguientes características de la implantación son postuladas por el autor?

I. Todo `throw` debe empilar un manejador.

II. Todo `throw` debe desempilar un manejador.

III. Sólo los `throw` que ocurren en bloques protegidos por `try` deben desempilar un manejador.

IV. Se debe empilar un manejador cada vez que se entra a un bloque protegido `try`.

V. Se debe empilar un manejador cada vez que se entra a una subrutina no protegida.

a) Sólo I, IV y V.

b) Sólo II, IV y V.

c) Sólo II, III y IV.

d) Sólo III, IV y V.

1.4. En relación a las co-rutinas y los iteradores, se puede decir de manera general que

a) Si un lenguaje los provee, entonces dispone de continuaciones.

b) Si un lenguaje compilado los provee, almacena los registros de activación en el *heap*.

c) Una co-rutina es un iterador que sólo puede ceder control a su llamador.

d) Ninguna de las anteriores.

1.5. 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( var b0 : Bar, b1 : Bar ) : returns Bar
```

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

```
r = grok( q0, q1 )
```

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

a) Si `q0` es un `Baz`, `q1` es un `Bar` y `r` es un `Foo`.

b) Si `q0` es un `Bar`, `q1` es un `Baz` y `r` es un `Foo`.

c) Si `q0` es un `Baz`, `q1` es un `Baz` y `r` es un `Baz`.

d) No se prohibirá ninguno, pues todos son válidos.

1.6. 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(Baz p)`

II. `Baz Bar::m2(Bar p)`

III. `Bar *Baz::m3(Bar *q,Baz r)`

IV. `Bar Bar::m4(Bar *b)`

V. `Baz::Baz(Bar *b)`

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

a) Sólo I y III.

b) Sólo II y IV.

c) Sólo III y V.

d) Sólo III, IV y V.

Pregunta 2 - 6 puntos

Considere el siguiente programa escrito en pseudocódigo

```
qux : int;

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

begin
  qux := 6;
  grok(qux,qux,qux)
  print(qux)
end
```

Ejecute el programa aplicando las reglas de pasaje de parámetros para cada uno de los casos especificados en la siguiente tabla. Suponga que el lenguaje pasa los parámetros en orden de derecha a izquierda.

Escriba en la columna **Salida del Programa** los resultados emitidos durante cada corrida.

Parámetro	**Pasaje por**	**Salida del Programa**
`foo`	Copia	`6`
`bar`	Referencia	`42`
`baz`	Valor	`24`
`foo`	Copia	`6`
`bar`	Referencia	`42`
`baz`	Referencia	`24`

Pregunta 3 - 8 puntos

Los algoritmos de búsqueda en árboles DFS y BFS pueden expresarse de manera muy sucinta como expresiones funcionales de orden superior. Si pensamos que para un nodo n particular, su lista de hijos corresponde al resultado de aplicar una función de cálculo de sucesores posibles, concluimos que un árbol se puede representar *implícitamente* como una función `children :: a -> [a]`

Entonces, escriba las funciones

```
bfs :: (a -> [a]) -> a -> [a]
dfs :: (a -> [a]) -> a -> [a]
```

que reciben como primer argumento la función de cálculo de sucesores que define el árbol, el nodo de inicio de búsqueda, produce como resultado la lista con los nodos del grafo de acuerdo con el recorrido BFS o DFS. Por ejemplo, si se cuenta con el grafo

```
children 1 = [2,3,4]
children 2 = [5,6,7]
children 3 = [8,9,10]
children 4 = [11,12,13]
children 8 = [14,15]
children _ = []
```

entonces esperaríamos

```
> dfs children 1
[1,2,5,6,7,3,8,14,15,9,10,4,11,12,13]
> bfs children 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Su solución **debe** utilizar funciones de orden superior. Soluciones recursivas no obtendrán puntaje alguno.

```
dfs :: Eq a => (a -> [a]) -> a -> [a]
dfs succ start = start : concatMap (dfs succ) (succ start)

bfs :: Eq a => (a -> [a]) -> a -> [a]
bfs succ start = concat $ takeWhile (not.null) $ iterate (concatMap succ) [start]
```

Pregunta 4 - 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. Las implantaciones clásicas proveen los predicados

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

  ```
  ?- 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 entero, pero el predicado puede usarse tanto para determinar como para establecer el argumento posicional aprovechando la unificación.

- `functor(Functor,Name,NumArgs)` que 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.

Se desea que Ud. utilize ambos predicados clásicos, para implantar el predicado moderno `universal/2` que permite convertir entre functores y listas:

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

Cuando el primer argumento está instanciado con un functor, el segundo debe unificar 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.

En su implantación de `universal/2` puede usar los predicados `var/1` y `nonvar/1` (si sabe para qué sirven, porque yo no). Además, el predicado debe producir *exactamente* una solución para cada caso de invocación, por lo que debe asegurarse de controlar el *backtracking* en los lugares precisos.

```
universal(F,[Name|Args]) :-
        var(F),
        length(Args,NumArgs),
        functor(F,Name,NumArgs),
        setargs(F,Args,1), !.
universal(F,[Name|Rest]) :-
        functor(F,Name,NumArgs),
        length(Rest,NumArgs),
        setargs(F,Rest,1), !.

setargs(_,[],_) :- !.
setargs(F,[A|As],Pos) :- arg(Pos,F,A),
                         Pos1 is Pos + 1,
                         setargs(F,As,Pos1).
```