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

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 `foo` está anidada en `bar`, entonces el compilador

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

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

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

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

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

a) Solamente uno hace verificación fuerte de tipos.

b) Solamente uno trata las funciones como objetos de primera clase.

c) Solamente uno asume funciones currificadas.

d) Solamente uno provee evaluación perezosa.

1.3. El libro de texto discute dos mecanismos para implantar el despacho de excepciones hacia el bloque manejador dinámicamente más cercano. El mecanismo más eficiente en el caso general, tiene el inconveniente de que no puede ser aplicado de manera directa en lenguajes con compilación separada. ¿Por qué?

a) Porque no se puede garantizar el invariante de búsqueda sobre todo el cuerpo de código.

b) Porque se implantan con `setjmp/longjmp`, y cada pareja requiere *buffers* de salto que no son alcanzables entre módulos diferentes.

c) Porque el mecanismo está diseñado para lenguajes interpretados que provean continuaciones.

d) Porque no se puede determinar estáticamente si cada `throw A` es atrapado por un `catch A`, y tampoco si hay un `catch A` por cada `throw A`.

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

a) Si un lenguaje los provee, su implantación debe almacenar los registros de activación en el *heap*.

b) Un iterador es una co-rutina que sólo puede ceder control a su llamador.

c) No tienen sentido en lenguajes funcionales, porque no corresponden a aplicaciones funcionales.

d) Si un lenguaje los provee, entonces automáticamente provee hilos.

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 `Baz` y `r` es un `Baz`.

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

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

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(Bar *p)`

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

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

IV. `Bar baz`

V. `Bar::Bar(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 I, III 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`	Valor	`6 42 114`
`bar`	Referencia	
`baz`	Copia	
`foo`	Referencia	`24 24 78`
`bar`	Copia	
`baz`	Referencia	

Pregunta 3 - 8 puntos

Sea el tipo de datos Haskell

```
data Inst = Add | Sub | Mul | Div | Num Int
  deriving (Eq,Show)
```

que representa las "instrucciones" de una calculadora. Utilizaremos listas Haskell no vacías para representar una pila de instrucciones de la calculadora usando notación *postfija*. Así se podrían tener expresiones como

```
[Num 42]
[Num 100, Num 16, Sub, Num 2, Div]
[Num 3, Num 7, Mul, Num 2, Mul]
```

Escriba la función

```
rpn :: [Inst] -> [Inst]
```

que recibe una lista de instrucciones, y produce como resultado una lista de *exactamente* un elemento, resultado de completar las operaciones siguiendo el orden de evaluación *postfijo*. Por ejemplo,

```
> rpn [Num 42]
[Num 42]
> rpn [Num 3, Num 7, Mul, Num 2, Mul]
[Num 42]
> rpn [Num 100, Num 16, Sub, Num 2, Div]
[Num 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. Puede suponer que la lista está construida correctamente; esto es, tiene al menos un elemento, y cuando tiene varios, corresponden a una expresión válida en postfijo. No necesita detectar la divisón entre cero.

```
rpn = foldl step []
  where step (Num x : Num y : r) Add     = Num (y+x) : r
        step (Num x : Num y : r) Mul     = Num (y*x) : r
        step (Num x : Num y : r) Sub     = Num (y-x) : r
        step (Num x : Num y : r) Div     = Num (y `div` x) : r
        step a                   (Num n) = Num n : a
```

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 modernas proveen 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. utilize el predicado `univ` para implantar los predicados clásicos de metaprogramación:

- **(3 puntos)** El predicado `argumento(Pos,Functor,Arg)` que triunfa si `Arg` es el `Pos`-ésimo argumento del functor `Functor`, comenzando a contar desde uno. Por ejemplo,

  ```
  ?- argumento(2,foo(bar,42,[baz,69]),X).
  X = 42
  yes
  ?- argumento(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. Si se suministra una posición menor a uno o mayor a la cantidad de argumentos en el functor, el predicado debe fallar.

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

  ```
  ?- estructura(foo(bar,42),Name,NumArgs).
  Name = foo
  NumArgs = 2
  yes
  ?- estructura(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.

En su implantación de `argumento/3` y `estructura/3` **no** puede usar el predicado `length/2`. Deberá usar los predicados `integer/1`, `var/1` y `nonvar/1` (si sabe para qué sirven, porque yo no). Además, los predicados deben producir *exactamente* una solución para cada caso de invocación, por lo que debe asegurarse de controlar el *backtracking* en los lugares precisos.

```
argumento(Pos,F,Val) :-
        integer(Pos),
        F =.. [N|L],
        get(Pos,L,Val), !.

get(1,[Val|_],Val).
get(N,[_|R],Val) :- N > 1, N1 is N-1, get(N1,R,Val).

estructura(F,Name,NumArgs) :-
        var(F), nonvar(Name), nonvar(NumArgs), !,
        build(0,NumArgs,L),
        F =.. [Name|L], !.

estructura(F,Name,NumArgs) :-
        nonvar(F), !,
        F =.. [Name|L],
        count(0,L,NumArgs).

build(N,N,[]).
build(P,N,[_|R]) :- P =< N, P1 is P+1, build(P1,N,R).

count(N,[],N) :- !.
count(N,[_|R],N2) :- N1 is N+1, count(N1,R,N2).
```