Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2010

Examen III

(40 puntos)

Nombre:

Carnet:

1. **(24 puntos)** Considere cuidadosamente las siguientes cuestiones y seleccione exactamente una respuesta de las alternativas que se presentan. Cada cuestión contestada correctamente **suma tres (3) puntos**, pero una cuestión contestada incorrectamente **resta un (1) punto**. Si Ud. selecciona más de una opción, la pregunta se considera contestada **incorrectamente.**

   (a) En la secuencia de llamada hay tareas que son responsabilidad exclusiva del llamador, tareas que son responsabilidad exclusiva del llamado y tareas que podrían ser realizadas por cualquiera de los dos. ¿Cuál de las siguientes afirmaciones es **cierta**?

   i. Si la tarea a realizar es responsabilidad del llamado, el código que la implanta se ejecuta justo antes de saltar a la dirección de inicio de la rutina llamada.
   ii. Si la tarea a realizar es responsabilidad del llamador, el código que la implanta se ejecuta en el prólogo o en el epílogo.
   iii. Si cualquiera de los dos puede realizar la tarea, entonces es preferible que el código que la implanta esté en el llamador porque así se ahorra tiempo.
   iv. ***Si cualquiera de los dos puede realizar la tarea, entonces es preferible que el código que la implanta esté en el llamado porque así se ahorra espacio.***

   (b) Se tiene el siguiente programa escrito en pseudocódigo:

```
procedure foo (bar, baz : int)
begin
  bar := bar + 1
  baz := baz + 2
  meh[bar] = meh[baz] + 10
end
begin
  var qux : int;
      meh : array [0..3] of int
  meh := { 3, 1, 0, 2 }
  qux := 1
  foo(qux, meh[qux])
  print meh
end
```

   Asuma que toda variable no inicializada explícitamente es elaborada con el valor cero y que los parámetros son pasados de izquierda a derecha.

   ¿Cuál sería la salida del programa si el lenguaje utiliza pasaje de parámetros por valor-resultado?

   i. `{ 3, 1, 0, 10 }`
   ii. `{ 3, 12, 0, 2 }`
   iii. `{ 3, 1, 12, 2 }`
   iv. ***Ninguna de las anteriores (`{3,3,12,2}`).***

(c) Continuando con la pregunta anterior, ¿cuál sería la salida del programa si el lenguaje utiliza pasaje de parámeros por referencia?

i. `{ 5, 3, 12, 2 }`
ii. *`{ 3, 3, 12, 2 }`*
iii. `{ 5, 11, 0, 2 }`
iv. Ninguna de las anteriores.

(d) Considere el método descrito en el libro de texto para implantar manejo de excepciones usando búsqueda binaria sobre una tabla generada a tiempo de compilación. ¿Cuál de las siguientes afirmaciones es **falsa**?

i. Si un manejador de excepciones provisto por el usuario es incapaz de procesar una excepción particular y vuelve a lanzarla, se busca en la tabla la dirección correspondiente al valor actual del contador de programa para encontrar su manejador.
ii. Siempre debe haber un manejador de excepciones implícito para desmantelar el registro de activación en caso de que sea necesario propagar las excepciones fuera de la subrutina en la que ocurren.
iii. ***El manejador de excepciones implícito propaga las excepciones de la misma manera que un manejador de excepciones provisto por el usuario.***
iv. Si el lenguaje ofrece compilación separada, es necesario que el enlazador provea un mecanismo para combinar todas las tablas en una sola o recurrir a un mecanismo mixto de tablas por módulo referenciadas en el registro de activación.

(e) 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 `Bar` subclase de `Foo`, en la que `Foo` es abstracta y `Bar` es concreta. ¿Cuáles de las siguientes declaraciones son **válidas**?

i. `Foo f1()`
ii. `Bar Foo::f2(Foo *p)`
iii. `Foo *Bar::f3(Bar q, Foo (*r)())`
iv. `Foo *bar`
v. `Foo qux = new Bar()`
vi. `Foo::Foo(Bar *b)`

A. Sólo ii y iv.
B. ***Sólo ii, iv y vi.***
C. Sólo i, iv y vi.
D. Sólo iii y v.

(f) Considere las siguientes definiciones en un lenguaje orientado a objetos con modelo de **valor** con resolución dinámica de métodos que dispone de operadores para tomar la dirección de un objeto (&) para almacenarla en un apuntador (*).

```
Foo f;
Bar b;
Foo *q;
Bar *s;
```

¿Cuál de las siguientes afirmaciones es **falsa**?

i. La asignación `q = &b` es válida
ii. La asignación `s = &f` es inválida, y genera un error semántico estático.
iii. ***La asignación `s = q` siempre es válida.***
iv. La asignación `q = s` siempre es válida.

(g) ¿Cuál de las siguientes características es **esencial** en un lenguaje de programación funcional?

i. ***Que las funciones sean objetos de primera clase.***
ii. Que existan funciones de orden superior.
iii. Que las funciones estén curryficadas.
iv. Que sea homoicónico.

(h) Considere el siguiente predicado en Prolog que tiene el propósito de invertir el orden de los elementos de una lista

```
reverse([],A,A).
reverse([X|R],S,A) :- reverse(R,[X|S],A).
```

¿cuál de las siguientes preguntas a Prolog consigue, en efecto, invertir el orden de los elementos de la lista `[foo,bar,baz]` para instanciar `A = [baz,bar,foo]`

i. `?- reverse([foo,bar,baz],A,[]).`
ii. `?- reverse([],[foo,bar,baz],A).`
iii. `?- reverse([foo,bar,baz],A,A).`
iv. *`?- reverse([foo,bar,baz],[],A).`*

2. **Programación Funcional.** Considere el tipo de datos

```
data Term = Num Int | Var String | Fun String [Term]
```

que podemos utilizar para expresar en Haskell cualquiera de los tipos de datos en Prolog, e.g.

Prolog	Term
42	`Num 42`
X	`Var "X"`
foo	`Fun "foo" []`
bar(Baz,quux,42)	`Fun "bar" [ Var "Baz", Fun "quux" [], Num 42 ]`

Así mismo, considere el tipo de datos

```
type Substitution = [ (Term,Term) ]
```

que podemos utilizar para expresar en Haskell las sustituciones de variables para el proceso de unificación en Prolog, de manera tal que cada tupla de la lista corresponde a una asociación entre una variable y un término. De este modo, la sustitución en Prolog

```
X = foo
Y = bar(42)
Z = Y
```

puede expresarse como

```
s = [ (Var "X", Fun "foo" []), (Var "Y", Fun "bar" [ Num 42 ]), (Var "Z", Var "Y") ]
```

Puede suponer que en el primer elemento de cada tupla *siempre* estará la descripción de una variable.

- **(3 puntos)** Escriba la función Haskell

  ```
  apply :: Substitution -> [Term] -> [Term]
  ```

  que toma *cada* término de una lista de términos, y sustituye *todas* las ocurrencias de las variables presentes en la sustitución por su término asociado, retornando la lista de términos con las sustituciones aplicadas. Por ejemplo, utilizando la sustitución `s` definida anteriormente, podríamos hacer

  ```
  > apply s [ Var "X", Fun "wtf" [ Var "Z" ], Num 42 ]
  [ Fun "foo" [], Fun "wtf" [ Var "Y" ], Num 42 ]
  ```

- **(7 puntos)** Escriba la función Haskell

  ```
  unify :: Term -> Term -> Maybe Substitution
  ```

  que implemente el Algoritmo de Unificación de Prolog descrito en el libro de texto y discutido en clase, de manera que recibiendo dos términos produzca de ser posible la sustitución que los unifique. Así, podría ser utilizada como

  ```
  > unify (Var "X") (Num 42)
  Just [ (Var "X", Num 42) ]
  > unify (Fun "foo" []) (Fun "foo" [ Num 42 ])
  Nothing
  > unify (Fun "foo" [ Var "X", Num 42 ]) (Fun "foo" [ Var "Y", Var "Y" ])
  Just [ (Var "X", Var "Y"), (Var "Y", Num 42) ]
  > unify (Fun "foo" [ Num 42 ]) (Fun "foo" [ Num 42 ])
  Just []
  ```

Puede utilizar cualquier función del preludio estándar Haskell. Puede escribir su solución empleando recursión directa o funciones de orden superior, según prefiera. No es necesario implantar el *occurs check*.

Aplicar una sustitución a una lista de términos, es lo mismo que aplicar la sustitución individualmente a cada término, así que la función `apply` no es más que una aplicación directa de listas por comprensión tal que para cada término de la lista se aplica una función auxiliar que en efecto aplica la sustitución.

La función auxiliar opera por recursión directa considerando:

(a) Si no hay nada que sustituir en una variable, la variable queda indicada.

(b) Si la sustitución incluye una variable, y en el término aparece dicha variable, entonces se realiza la sustitución recursivamente sobre el término asociado a la variable.

(c) Cualquier sustitución sobre un functor, no es más que construir el functor pero aplicando la sustitución a sus argumentos.

(d) Los números no son susceptibles de sustitución, así que se retornan inmediatamente y sin modificación.

```
apply :: Substitution -> [Term] -> [Term]
apply s ts = [ apply' s t | t <- ts ]
  where apply' []             (Var y)    = (Var y)
        apply' ((Var x, t):s) (Var y)    = if (x == y) then apply' s t
        apply' s              (Fun n ts) = Fun n (apply s ts)
        apply' _              (Num i)    = Num i
```

El algoritmo de unificación de Robinson, sin *occurs check*, sobre dos términos es tal que:

(a) Un número unifica con otro con sustitución vacía, siempre que sean el mismo número.

(b) Dos variables sin instanciar unifican una con la otra, con sustitución trivial en la cual una variable se sustituye por otra.

(c) Una variable sin instanciar unifica con cualquier otro término, con sustitución trivial. Nótese que debe considerarse el caso simétrico.

(d) Dos functores unifican si tienen el mismo nombre y si pueden unificarse sus argumentos. Para esto introducimos una función auxiliar encargada de la unificación de los argumentos, que también opera de forma directa:

   i. Si ambas listas de argumentos son vacías, unifican con sustitución trivialmente vacía.

   ii. Si alguna lista es vacía y la otra no, no unifican.

   iii. Se procesa un argumento de cada lista para determinar si unifican, produciendo una sustitución `s`. Se aplica dicha unificación al resto de los argumentos de ambas listas y se intenta unificar el resto de los argumentos, produciendo una sustitución `s'`, que se concatena con `s` para producir la sustitución final.

```
unify :: Term -> Term -> Maybe Substitution
unify (Num i)    (Num j)    = if i == j then Just [] else Nothing
unify (Var x)    (Var y)    = Just [(Var x,Var y)]
unify (Var x)         y     = Just [(Var x,y)]
unify      x     (Var y)    = Just [(Var y,x)]
unify (Fun a xs) (Fun b ys) = if (a == b) then unifyArgs xs ys
  where unifyArgs []     []     = Just []
        unifyArgs []     _      = Nothing
        unifyArgs _      []     = Nothing
        unifyArgs (x:xs) (y:ys) =
          case unify x y of
            Nothing -> Nothing
            Just s  -> case unifyArgs (apply s xs) (apply s ys) of
                         Nothing -> Nothing
                         Just s' -> Just (s ++ s')
```

3. **Programación Lógica.** La función `map` de Haskell recibe una función *unaria* y una lista de valores cuyos tipos son compatibles con dicha función, produciendo una lista con los resultados de aplicarla sobre cada elemento, esto es

   `map f` $[x_0, x_1, \ldots, x_n] \Rightarrow [f(x_0), f(x_1), \ldots, f(x_n)]$

   Sin embargo, la función `map` de Scheme es mucho más general pues recibe una función *n-aria* y n listas de igual longitud m, produciendo una lista de longitud n con el resultado de aplicar la función a los elementos correspondientes de cada lista, esto es

   `(map f` $(x_{00}\, x_{01}\, \ldots\, x_{0m})\ (x_{10}\, x_{11}\, \ldots\, x_{1m}) \ldots (x_{n0}\, x_{n1}\, \ldots\, x_{nm})$`)`

   tiene como resultado

   `(` $f(x_{00}, x_{10}, \ldots, x_{n0})\, f(x_{01}, x_{11}, \ldots, x_{n1})\, \ldots\, f(x_{0m}, x_{1m}, \ldots, x_{nm})$ `)`

   (a) **(6 puntos)** Escriba el predicado Prolog `genmap/3` que implante el comportamiento de la función `map` como en Scheme. Esto es, el predicado tiene la forma

   ```
   genmap(+Predicado,+Argumentos,-Resultados)
   ```

   donde `Predicado` es el nombre de algún *functor* Prolog, `Argumentos` es una lista de listas, conteniendo los argumentos necesarios para ser aplicados al *functor* de manera que `Resultados` se instancie con la aplicación del functor a los argumentos correspondientes. El predicado debe fallar si no hay la misma cantidad de elementos en las listas. Por ejemplo,

   ```
   ?- genmap(foo,[[1,2],[3,4],[X,Y]],R).
   R = [foo(1,3,X),foo(2,4,Y)]
   ?- genmap(bar,[[1],[2,3]],R).
   no
   ?- genmap(baz,[[4,2]],R).
   R = [baz(4),baz(2)]
   ```

   Puede utilizar cualquier predicado estándar de Prolog. Su predicado debe triunfar *únicamente* una vez.

```
% No puedo hacer genmap recursivo, así que uso un predicado auxiliar.
genmap(Predicado,Argumentos,Resultados) :-
  domap(Predicado,Argumentos,Resultados).
% Si se acabaron los argumentos, termino la aplicación; en caso contrario
% tomo un juego de argumentos, construyo el functor y lo agrego a la lista
% para continuar recursivamente con el resto.
domap(_,Argumentos,[]) :-
  noquedan(Argumentos), !.
domap(Predicado,Argumentos,[Aplicado|Rest]) :-
  argumentos(Argumentos,Args,RestArgs),
  Aplicado =.. [Predicado|Args],
  domap(Predicado,RestArgs,Rest).
% No quedan argumentos, si la lista está compuesta enteramente por listas vacías.
noquedan([]).
noquedan([[]|Rest]) :- noquedan(Rest).
% De una lista de listas, se construyen una lista con los primeros elementos y una
% lista con los restos. Note que los dos últimos argumentos son acumuladores.
argumentos([],[],[]).
argumentos([[Arg|RestArgs]|NextArgs], [Arg|MoreArgs], [RestArgs|Rest]) :-
  argumentos(NextArgs,MoreArgs,Rest).
```