Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2009

Examen III

(40 puntos)

Nombre:

Carnet:

1. **(20 puntos)** Considere cuidadosamente las siguientes preguntas. Cada una presenta cinco alternativas, debe seleccionar sólo una de éstas. Cada respuesta correcta tiene un valor de **cuatro (4) puntos.** Se aplicará factor de corrección, **restando dos (2) puntos** por cada respuesta incorrecta.

   (a) El segundo mecanismo para implantar excepciones considerado en el libro de texto es el de mantener una lista de pares de direcciones. Con esta alternativa, ¿cuáles de las siguientes características de la implantación son postuladas por el autor?

   i. Se realiza una búsqueda binaria cada vez que se entra a un `try`.
   ii. Cuando se propaga una excepción fuera de la subrutina, la búsqueda debe hacerse con la dirección de retorno de la función en lugar del contador de programa.
   iii. El costo de manejar un `throw` aumenta.
   iv. El costo de manejar un `throw` disminuye.
   v. El método no es aplicable directamente a lenguajes con compilación separada.

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

   (b) Considere las siguientes definiciones Haskell

```
rail p f g x =
  if p x
    then []
    else f x : rail p f g (g x)
shot = rail (==0) id (-1)
```

   Ahora considere las siguientes afirmaciones:

   i. El tipo de `rail` es `(t -> Bool) -> (t -> a) -> (t -> t) -> t -> [a]`
   ii. `(head.shot)` es de tipo `[Integer] -> Integer`
   iii. El tipo de `shot` es `[Integer] -> Integer`
   iv. `shot` es una función de orden superior.
   v. `(-1)` es una función currificada

   ¿Cuáles afirmaciones son ciertas?

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

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

```
procedure foo (bar, baz : int)
begin
  bar := bar + baz
  baz := bar + baz
end
begin
  var qux : int;
  qux := 5;
  foo(qux,qux);
  write(qux)
end
```

Considere *dos* corridas del mismo programa:

- En la primera, el procedimiento `foo` es llamado pasando `bar` por valor y `baz` por valor/resultado.
- En la segunda, el procedimiento `foo` es llamado pasando *todos* los parámetros por referencia.

¿Cuál sería la salida del programa en estos dos casos, respectivamente?

i. 20 y 20
ii. 10 y 20
iii. **15 y 20**
iv. 15 y 10
v. 20 y 20

(d) 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 **inválidas**?

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 y iii.
C. Sólo i, v y vi.
D. Sólo i, iii, v y vi.
E. **Sólo i, iii y v.**

(e) En relación con la implementación de corrutinas e iteradores, ¿cuáles de las siguientes afirmaciones son ciertas?

i. Cada corrutina debe contar con su propia pila de ejecución.
ii. Permitiendo la declaración de corrutinas solamente en el nivel más externo de anidamiento estático se evita la necesidad de pilas cactus.
iii. Los iteradores reales (como en Clu, Ruby o Python) pueden ser implementados mediante corrutinas.
iv. En ausencia de corrutinas, los iteradores reales (como en Clu, Ruby o Python) pueden ser implementados sobre una sola pila de ejecución.

A. Sólo i y iii.
B. Sólo i y iv.
C. Sólo i, ii y iii.
D. Sólo i, ii y iv.
E. **Las cuatro afirmaciones son ciertas.**

2. **Programación Funcional de Orden Superior.** A continuación se especifican dos funciones a implantar, mostrando ejemplos de uso en Haskell y en Scheme. Ud. debe implantar una de las funciones en Haskell y la otra en Scheme. La implantación en Haskell **debe** utilizar funciones de orden superior (`map` y `foldr`) e incluir la firma de la función, mientras que la implantación en Scheme **debe** realizar el trabajo de construcción de listas recursivamente.

   (a) **(5 puntos)** La función `inits` produce una lista con todos los segmentos iniciales de la lista recibida como parámetro. Esto es, en Haskell,

   ```
   > inits [1,2,3,4]
   [[],[1],[1,2],[1,2,3],[1,2,3,4]]
   ```

   mientras que en Scheme,

   ```
   > (inits '(1 2 3 4))
   (() (1) (1 2) (1 2 3) (1 2 3 4))
   ```

   En Haskell:

   ```
   inits :: [a] -> [[a]]
   inits xs = foldr f [[]] xs
               where f x xss = [] : map (x:) xss
   ```

   En Scheme:

   ```
   (define inits (lambda (l)
       (cond
           ((null? l) '(()))
           (#t (let ((fijo  (car l))
                     (resto (cdr l)))
               (cons '() (map (lambda (i) (cons fijo i)) (inits resto))))))))
   ```

   (b) **(5 puntos)** La función `nodups` elimina los duplicados adyacentes de la lista recibida como parámetro. Esto es, en Haskell

   ```
   > nodups [1,2,2,3,3,3,1,1,4,5]
   [1,2,3,1,4,5]
   ```

   mientras que en Scheme,

   ```
   > (nodups '(1 2 2 3 3 3 1 1 4 5))
   (1 2 3 1 4 5)
   ```

   En Haskell:

   ```
   nodups :: Eq a => [a] -> [a]
   nodups = foldr prepend []
              where prepend x []     = [x]
                    prepend x (y:xs) = if (x == y) then (y:xs) else (x:y:xs)
   ```

   En Scheme:

   ```
   (define nodups (lambda (l)
       (cond
           ((null? l) l)
           ((null? (cdr l)) l)
           ((eqv? (car l) (car (cdr l))) (nodups (cdr l)))
           (#t (cons (car l) (nodups (cdr l)))))))
   ```

3. **Programación Lógica**

(a) **(6 puntos)** *Mergesort* es una estrategia eficiente para ordenar una lista basada en dividirla en mitades iguales, ordenar las mitades por separado y luego combinar ambas mitades ordenadas. Así, puede contarse con el predicado `ordena/2` tal que

```
?- ordena([5,2,3,4,1],Ordenada).
Ordenada = [1,2,3,4,5]
```

Provea una implantación de los predicados `ordena/2`, `divide/3` y `combina/3` que produzca *exactamente* una solución. Asuma que las listas a procesar son homogéneas y constituidas solamente por números.

(b) **(4 puntos)** Generalice la solución anterior para tener el predicado `ordena/3` que incorpore el *predicado* a utilizar para comparar, de manera tal que pueda usarse (`<` compara números y `@<` compara átomos).

```
?- ordena([5,2,3,4,1],<,Ordenada).
Ordenada = [1,2,3,4,5]
?- ordena([foo,bar,baz],@<,Ordenada).
Ordenada = [bar,baz,foo].
```

Nota: si lo prefiere puede escribir **una** sola implantación para ambas preguntas resolviendo la parte (b) primero y luego escribiendo el predicado faltante para la parte (a).

```
% Solución específica para la parte (a) utilizando
% el predicado </2 específico para ordenar números.

ordena(Numeros,Ordenados) :- ordena(Numeros,<,Ordenados).

% Solución general para la parte (b)

ordena([],_,[]).                                    % Trivialmente ordenada.
ordena([X],_,[X]).                                  % Trivialmente ordenada.
ordena(Desordenada,Predicado,Ordenada) :-
    Desordenada = [_,_|_],                          % Asegura unicidad, cut NO SIRVE.
    divide(Desordenada,Parte1,Parte2),              % El enunciado
    ordena(Parte1,Predicado,Ordenada1),             %    escrito tal cual
    ordena(Parte2,Predicado,Ordenada2),             %       pero en Prolog
    combina(Predicado,Ordenada1,Ordenada2,Ordenada). %          conservando las relaciones.

divide([],[],[]).                                   % Longitud par, no sobra nada.
divide([X],[X],[]).                                 % Longitud impar, sobra uno.
divide([X,Y|Resto], [X|Mitad1], [Y|Mitad2]) :-      % Dos o más, uno para cada mitad.
    divide(Resto,Mitad1,Mitad2).

combina(_,[],L,L).                          % Agotada la primera lista.
combina(_,L,[],L) :- L \= [].               % Agotada la segunda lista.
                                            % La primera no puede haberse agotado :-)

combina(Predicado,[X|T1],[Y|T2],[X|T]) :-   % Si p(X,Y), X precede a Y.
    Check =.. [Predicado,X,Y],              % Preparo el predicado de ordenamiento.
    call(Check), !,                         % Ejecutar. Si triunfa, impedir backtracking.
    combina(Predicado,T1,[Y|T2], T).        % Combinar el resto sin olvidar Y.
combina(Predicado,[X|T1],[Y|T2],[Y|T]) :-   % Si ¬p(X,Y), Y precede a X.
     combina(Predicado,[X|T1],T2,T).        % Combinar el resto sin olvidar X.
```