Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2008

Examen I

(30 puntos)

Nombre:

Carnet:

1. **(5 puntos)** Considere cuidadosamente las siguientes cuestiones y seleccione exactamente una respuesta de las alternativas que se presentan. Cada cuestión contestada correctamente vale **un (1) punto** pero una cuestión contestada incorrectamente **resta medio punto**. Si Ud. selecciona más de una opción, la pregunta se considera contestada **incorrectamente**.

   (a) En un lenguaje con alcance dinámico **siempre** es necesario

      i. Un recolector de basura.
      No, pues resulta posible tener un lenguaje con alcance dinámico sin recolector de basura.

      ii. Utilizar la cadena estática cuando se construyen clausuras.
      No, pues en un lenguaje con alcance dinámico la cadena estática es inútil.

      iii. **Diferir verificaciones semánticas hasta la ejecución.**
      Como hay que esperar hasta la ejecución para determinar cuáles variables participan en los cómputos, es necesario en ese momento hacer las verificaciones de tipos de datos y establecer la validez en la cantidad y tipo de los argumentos pasados a funciones.

      iv. Operar sin variables globales.
      No, pues en un lenguaje con alcance dinámico se considera que las variables globales están en un registro de activación inicial que corresponde al programa principal.

   (b) Una variable local

      i. Tiene una ubicación relativa diferente dentro de cada activación de la subrutina que le contiene.
      No, pues precisamente las variables locales tienen una ubicación relativa idéntica dentro de cada activación de la subrutina que le contiene, para poder acceder a ellas con desplazamientos constantes a partir del apuntador a la base del registro de activación.

      ii. Tiene una ubicación fija durante la ejecución del programa.
      No, pues las variables locales son almacenadas en la pila de ejecución o en el *heap* en caso de clausuras, y en ambos casos su ubicación depende del flujo de ejecución.

      iii. Sólo es accesible si el lenguaje tiene alcance estático.
      No, pues una variable puede estar en el ambiente de referencia y ser global o local.

      iv. **Es accesible desde subrutinas anidadas a través de la cadena estática.**
      Si una variable es local a un procedimiento que a su vez contiene procedimientos anidados, cualquiera de estos últimos dispondrá de la cadena estática para alcanzar el registro de activación de la rutina que les contiene, y allí utilizar un desplazamiento constante para utilizar la variable local.

(c) En C, la expresión `*p++ = (*q++)+1` es equivalente a

Nota: esta expresión es típica en la copia de arreglos utilizando `q` para apuntar al origen y `p` al destino. Esta expresión particular copia lo apuntado por `q` más uno en la dirección apuntada por `p`, y después se incrementan tanto `p` como `q` para avanzar a las posiciones siguientes de los respectivos arreglos.

i. `*p = *(q+=1); p++`
No, pues el apuntador `q` se incrementa *antes* de hacer la copia.

ii. `*(p+=1) = *(q+=1)+1`
No, pues *ambos* apuntadores se incrementan *antes* de hacer la copia.

iii. `*(p+=1)++ = *q; q += 1`
No, pues el apuntador `p` se incrementa *antes* de hacer la copia.

iv. `*p = *q; ++(*p); p += 1; ++q`
Lo apuntado por `q` se almacena en la dirección de `p`. Luego se incrementan los contenidos de lo apuntado por `p`. Luego se incrementa el apuntador `p`. Finalmente se incrementa el apuntador `q`.

(d) La diferencia entre un módulo como administrador y un módulo como tipo estriba en que:

i. El espacio de nombres del primero solamente contiene funciones, mientras que el segundo contiene funciones y tipos de datos.
No, pues los espacios de nombres de ambos pueden contener funciones, tipos de datos, variables y, dependiendo del lenguaje, constantes.

ii. El primero sólo puede tener espacio de nombres abierto o cerrado según sea necesario, mientras que el segundo sólo puede tener espacio de nombres cerrado.
No, pues la manera en que se exportan e importan símbolos desde y hacia los módulos es independiente del modelo de abstracción administrador o tipo de datos.

iii. **En el primero los datos se pasan a las funciones como argumentos explícitos, mientras que en el segundo se pasan implícitamente.**
Libro de texto, página 130.

iv. Sólo con el segundo es posible crear tipos de datos abstractos.
No, pues con cualquiera de los dos se puede hacer un tipo de datos abstracto.

(e) En un lenguaje que usa el modelo de valor

i. Una asignación de la forma `a := b` siempre produce un *alias*.
No, pues en un lenguaje con modelo de valor las asignaciones son copias de datos, de modo que `a` y `b` serían ubicaciones *diferentes* que almacenan dos copias del mismo valor.

ii. Cualquier expresión puede utilizarse como *l-value*.
No, sólo pueden utilizarse como *l-values* expresiones que denoten direcciones de memoria.

iii. No es posible construir referencias.
No, pues es posible utilizar un valor como referencia si el lenguaje provee los operadores necesarios para tomar una referencia (`&` en C/C++, \ en Perl...) y seguirla (* en C/C++, -> en Perl...)

iv. **Las variables no son más que contenedores con nombre.**
Libro de texto, página 239.

2. Recursión:

(a) **(4 puntos)** Considere la siguiente solución al problema de las Torres de Hanoi escrita en C:

```
void hanoi(int N, int Inicio, int Final, int Auxiliar)
{
    if (N == 1) {
        printf("Mover desde %d hasta %d\n",Inicio,Final);
    } else {
        hanoi(N-1,Inicio,Auxiliar,Final);
        printf("Mover desde %d hasta %d\n",Inicio,Final);
        hanoi(N-1,Auxiliar,Final,Inicio);
    }
}
```

Una de esas llamadas corresponde a recursión de cola. Identifíquela y elimínela.

La segunda llamada a `hanoi` corresponde a recursión de cola, pues a su retorno no hay ningún cómputo pendiente. Para eliminarla, es necesario convertir esa llamada en un proceso iterativo:

```
void hanoi(int N, int Inicio, int Final, int Auxiliar)
{
    int t;
    while (N > 0) {
    if (N == 1) {
        printf("Mover desde %d hasta %d\n",Inicio,Final);
        return;
    } else {
        hanoi(N-1,Inicio,Auxiliar,Final);
        printf("Mover desde %d hasta %d\n",Inicio,Final);
        N = N - 1;
        t = Inicio;
        Inicio = Auxiliar;
        Auxiliar = t;
    }
}
```

Nota: este caso es idéntico a la transformación del MCD de recursivo a iterativo visto en clase (y suministrado como ejemplo), salvo que en lugar de inducir operaciones aritméticas, se cambian variables de posición para satisfacer los argumentos de la llamada recursiva.

(b) **(4 puntos)** Considere la siguiente función en Haskell para invertir listas

```
invertir        :: [a] -> [a]
invertir []     =  []
invertir (x:xs) =  invertir xs ++ [x]
```

Reescriba la función para introducir recursión de cola.

La técnica estándar para introducir recursión de cola consiste en acumular el trabajo hacia la llamada recursiva. En este caso, se utiliza una función auxiliar que comienza trabajando con la lista original y una lista vacía, y va tomando el primer elemento de la lista izquierda, y lo coloca de primero en la lista acumulada, invirtiendo las posiciones hasta procesar toda la lista original.

```
invertir xs = ritrevni xs []
              where
                ritrevni []     rs = rs
                ritrevni (y:ys) rs = ritrevni ys (y:rs)
```

Nota: este caso es idéntico a la transformación del factorial vista en clase (y suministrada como ejemplo) salvo que en lugar de multiplicar y usar el 1 como acumulador inicial, se construyen listas agregando elementos a una lista vacía como acumulador inicial.

3. **(8 puntos)** Suponga un lenguaje *imperativo* con iteradores reales, que son declarados como una rutina cualquiera y donde la producción de valores se indica mediante la instrucción `yield`. Suponga que el lenguaje permite manipular listas de la siguiente manera, i.e.

   - El primer elemento de una lista está en la posición 0.
   - `a.init(i)` retorna una lista con los elementos iniciales de la lista `a` desde la 0-ésima hasta la `i`-ésima posición inclusive.
   - `a.tail(i)` retorna la lista `a` a partir del `i`-ésimo elemento hasta el último elemento.
   - `a.length` retorna la longitud de la lista `a`.
   - `a ++ b` concatena las listas `a` y `b`.
   - `a[i]` retorna el `i`-ésimo elemento de la lista `a`.

   Escriba un iterador `permutaciones(a : lista)` que produzca las permutaciones de la lista, retornándolas una a una con cada invocación. **Pista:** si quiero producir una lista sin el `i`-ésimo elemento de la lista `a`, entonces escribo

   ```
   a_sin_i-esimo := a.init(i-1) ++ a.tail( i+1 )
   ```

   El algoritmo típico para calcular las permutaciones de una lista se define recursivamente como:

   (a) Una lista de uno o menos elementos es su única permutación.

   (b) En cualquier otro caso:

      i. Se fija un elemento de la lista.
      ii. Se obtienen las permutaciones de la lista sin el elemento fijado.
      iii. Se producen tantas listas como permutaciones, utilizando el elemento fijado como inicial.
      iv. Se repite seleccionando otro elemento de la lista para fijar, hasta haberlos probado todos.

   Este algoritmo puede combinarse trivialmente haciendo uso *consistente* de los tipos de datos (listas se combinan con listas).

   ```
   Lista permutaciones(Lista a)
   {
     Lista   fijo, resto, p;
     Integer i;
     if (a.length <= 1) {
       yield a
     } else {
       for (i=0; i < a.length; i++) {
         fijo = a.tail(i).head(0);
         resto = a.head(i-1) ++ a.tail(i+1);
         while (p = permutaciones(resto)) {
           yield fijo ++ p;
         }
       }
     }
   }
   ```

   Nótese que la definición de las funciones `head` y `tail` permite obtener de manera muy simple una lista que tenga solamente el elemento i-ésimo simplemente escribiendo `a.tail(i).head(0)`, pues esto produce una lista, mientras que `a[i]` produce un elemento, en consecuencia `a[i] ++ lista` es una operación con tipos inconsistentes.

4. Considere el siguiente programa escrito en pseudocódigo

```
int u = 42;
int v = 69;
int w = 17;
proc add( z : int )
    u := v + u + z
proc bar( fun : proc)
    int u := w;
    fun(v)
proc foo( x : int, w : int)
  int v := x;
  bar(add)
main
  foo(u,13)
  print(u)
end;
```

¿Qué imprime el programa?

(a) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **estático**.

(b) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **dinámico** y ***deep binding***.

(c) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **dinámico** y ***shallow binding.***

En **todos** los casos asuma que las clausuras se construyen en el momento en que las funciones son **pasadas como parámetros.**

Nota: si solamente muestra los resultados (aunque sean correctos) no obtendrá puntos; es **imprescindible** exhibir los contenidos de la pila de ejecución hasta el momento en que se invoca la función `add` incluyendo las cadenas dinámica y estática así como las clausuras construidas.