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

Carnet:_____

Nombre:_____

Examen I

(25 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	**Total**
12 puntos	6 puntos	7 puntos	**25 puntos**

Espacio para contestar la Pregunta 3

Pregunta 1 - 12 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 dos (2) puntos. Tres subpreguntas incorrectas eliminan una correcta.

1.1. En un lenguaje interpretado con alcance dinámico **siempre** es necesario.

a) Utilizar la cadena dinámica cuando se construyen clausuras.

b) Almacenar todos los objetos en el *heap* y en la pila de ejecución.

c) Diferir la mayoría de las verificaciones semánticas hasta la ejecución del programa.

d) Todas las anteriores.

1.2. Las constantes obvias (*manifest constants*) corresponden a valores que pueden calcularse estáticamente, mientras que las constantes de elaboración (*elaboration time constants*) no. Suponga un lenguaje que ofrece ambos tipos de constantes, y que las diferencia con dos palabras reservadas `const` y `readonly`, respectivamente. Se tiene un programa en el lenguaje que usa `const` y `readonly` para definir constantes en el alcance de un procedimiento *recursivo*. Entonces, la máxima eficiencia de espacio y tiempo se logra cuando:

a) Las `readonly` se almacenan en la pila, pero las `const` se almacenan en área estática.

b) Las `const` se almacenan en la pila, pero las `readonly` se almacenan en área estática.

c) Ambas se almacenan en la pila.

d) Ambas se almacenan en memoria estática.

1.3. Una variable local a un procedimiento:

a) Nunca tendría vida global.

b) Nunca estaría en una clausura del procedimiento.

c) Nunca se almacenaría en el *heap*.

d) Nunca podría ser evaluada estáticamente.

1.4. Considere un lenguaje imperativo, de propósito general, con alcance estático y sin clausuras. ¿Cuál de las siguientes verificaciones no puede hacerse de forma estática?

a) Verificar que ninguna expresión produzca desbordamiento (*overflow*).

b) Verificar que una subrutina reciba el número correcto de argumentos

c) Verificar que toda excepción pueda ser atrapada.

d) Verificar que una variable reciba un valor cónsono con su tipo.

1.5. Un lenguaje permite escribir funciones con nombres arbitrarios (e.g. `this_is_how_we_sum`) y asociarlas con los operadores del lenguaje (e.g. `+`) con algún mecanismo sintáctico. El compilador verifica estáticamente si la función definida concuerda en cantidad de argumentos con lo esperado por el operador. Este es un caso de:

a) Polimorfismo de subtipos (*subtype polymorphism*).

b) Polimorfismo paramétrico (*parametric polymorphism*).

c) Sobrecarga (*overloading*).

d) *Aliasing*.

1.6. El ambiente de ejecución de un lenguaje con almacenamiento en *heap* requiere un manejador de memoria para administrar el espacio adicional. Considere las siguientes afirmaciones en relación a cualquier manejador de memoria:

I. *Best Fit* siempre es mejor que *First Fit* para controlar la fragmentación externa.

II. La fragmentación interna depende de la selección de tamaños para los bloques.

III. La fragmentación interna puede combatirse compactando bloques de memoria adyacentes durante la liberación de memoria.

¿Cuáles afirmaciones son ciertas?

a) II solamente.

b) I y II solamente.

c) I y III solamente.

d) II y III solamente.

Pregunta 2 - 6 puntos

Considere el siguiente programa escrito en pseudocódigo:

```
int foo = 1
int bar = 2

proc fudge(int qux)
  bar := foo + bar + qux

proc B(proc wee, int bar)
  if bar > 1
    B(wee,bar-1)
  wee(bar)
  print(bar)

proc A(int baz, int foo)
  int bar = baz
  B(fudge,foo)
  print(bar)

main
  A(foo,2)
  print(bar)
end
```

Ejecute el programa aplicando las reglas de alcance y construcción de clausuras para cada uno de los casos especificados en la siguiente tabla. Escriba en la columna **Salida del Programa** los resultados emitidos durante cada corrida.

Alcance	**Asociación**	**Salida del Programa**
Estático	Profunda (*Deep*)	1 2 1 7
Dinámico	Profunda (*Deep*)	1 5 8 2
Dinámico	Superficial (*Shallow*)	4 6 1 2

Pregunta 3 - 7 puntos

Suponga que Ud. está usando Haskell para implantar un lenguaje con alcance dinámico. Cada símbolo del lenguaje será un `String` y ya existe un tipo de datos abstracto `Info` que representa toda la información necesaria para cada símbolo. Combinando esos tipos de datos con los tipos lista y tupla convencionales de Haskell, complete las definiciones (**un (1) punto cada una**)

```
type AL = [(String,Info)]

type RT = [(String,[Info])]
```

tales que `AL` corresponda a una Lista de Asociaciones y `RT` a una Tabla Central de Referencia. Una vez definidos ambos tipos, provea la implementación de las funciones

- `enterScopeAL` que incorpora en la Lista de Asociaciones actual la lista de nuevas asociaciones al entrar a un alcance, produciendo el ambiente combinado (**un (1) punto**).

  ```
  enterScopeAL :: AL- > [(String,Info)] -> AL
  enterScopeAL t n = n ++ t
  ```

  Una definición alternativa con estilo Haskell más elegante sería

  ```
  enterScopeAL :: AL- > [(String,Info)] -> AL
  enterScopeAL = flip (++)
  ```

- `enterScopeRT` que incorpora en la Tabla Central de Referencia actual la lista de nuevas asociaciones al entrar a un alcance, produciendo el ambiente combinado (**cuatro (4) puntos**).

  ```
  enterScopeRT :: RT -> [(String,Info)] -> RT
  enterScopeRT t [] = t
  enterScopeRT t bs = foldl addOrReplace t bs
    where addOrReplace []          (s,i)  = [(s,[i])]
          addOrReplace ((s,ob):rs) (ns,i) =
            if s == ns then (s,i:ob) : rs
                       else (s,ob)   : addOrReplace rs (ns,i)
  ```

Nota: solamente puede usar funciones del preludio Haskell.