Taller de Programación Declarativa

Modelos y Paradigmas de Programación

Escuela de Ingeniería de Sistemas y Computación

Universidad del Valle

Profesor Juan Francisco DIAZ FRIAS*

A continuación Usted encontrará una serie de ejercicios que debe resolver **individualmente**. No se admite ninguna consulta con sus compañeros de otros grupos de trabajo, pero sí con el profesor. Esa es la hipótesis básica sobre la cual se fundamenta esta tarea. Cualquier violación a esta regla será considerada un intento de copia y causará la anulación del taller.

La solución a todos y cada uno de los puntos del taller se debe almacenar en un archivo `.oz` tal como se describe en cada punto. Usted debe entregar como solución al examen, el conjunto de esos archivos empaquetado y comprimido en un solo archivo cuyo nombre debe ser análogo a `taller1JFDiaz.zip` o `taller1JFDiaz.tgz` o `taller11JFDiaz.tar.gz` o `taller1JFDiaz.rar` donde en lugar de JFDiaz van las iniciales de los nombres y el primer apellidos suyo. Al desempaquetar este archivo, se deben encontrar solamente los archivos siguientes: `max.oz`, `factList.oz`, `sum.oz`, `alltail.oz`, `convolucion.oz`, `fibo.oz`, `add.oz`, `newton.oz` con el contenido descrito abajo. Adicionalmente debe encontrarse un archivo `practica1.pdf`, con el desarrollo del taller que no corresponde a código Oz.

No envíe ningún archivo adicional, así lo haya utilizado para sus pruebas. Su solución debe ser enviada a través del Campus Virtual en el enlace correspondiente al Taller de Programación Declarativa según las fechas indicadas allí. El sistema no permitirá que se envíen soluciones después de dicha fecha.

1. Suponga que una función tiene una sentencia **`if`** en la cual se omite la clausula **`else`**. Entonces, se

*juanfco.diaz@correounivalle.edu.co

lanzará una excepción, cuando se invoque a la función y esa condición sea falsa. Explique por qué este comportamiento es correcto para las funciones y por qué esto no ocurre para los procedimientos.

2. La siguiente función retorna el máximo número en una lista de enteros.

```
fun {Max L}
  fun {MaxLoop L M}
    case L of nil then M
      [] H|T then
      if M > H then {MaxLoop T M}
      else {MaxLoop T H} end
    end
  end
in
  if L == nil then error
  else {MaxLoop L.2 L.1} end
end
```

   Reescriba la función utilizando notación procedimental. Cada declaración y aplicación de funciones debe ser sustituida por una declaración y aplicación de procedimientos.

   Su solución debe guardarla en un archivo de nombre `max.oz`.

3. Escriba una función que reciba como parámetro un entero N y retorne una lista incremental de factoriales, comenzando en *1!* hasta *N!*. Implemente la función de manera que la computación sea recursiva.

   Ej. `{FactList 4}` retorna `[1 2 6 24]`, que corresponde a la lista de factoriales `[1! 2! 3! 4!]`.

   Su solución debe guardarla en un archivo de nombre `factList.oz`.

4. Considere la siguiente implementación de la sumatoria de un número.

```
fun {Sum N}
    if N == 0 then 0
    else N+{Sum N-1}
    end
end
```

   Explique por qué esta implementación no es iterativa. Reescriba la función para que cálcule la sumatoria de `N` de forma iterativa. Justifique que su solución es iterativa y calcula correctamente la suma de los `N` primeros números naturales.

   Su solución debe guardarla en un archivo de nombre `sum.oz`.

5. Escriba el procedimiento `{ForAllTail Xs P}` que aplica el procedimiento `P` a cada sublista no vacía de `Xs`. La aplicación del procedimiento `{ForAllTail [X1 ... Xn] P}` corresponde a la secuencia de sentencias:

```
{P [X1 X2 ... Xn]}
{P [X2 ... Xn]}
 ...
{P [Xn]}
```

Su solución debe guardarla en un archivo de nombre `alltail.oz`.

6. *Convolución con recursión de cola.* Escriba una función que reciba dos listas $[x_1\ x_2\ ...\ x_n]$ y $[y_1\ y_2\ ...\ y_n]$ y devuelva su convolución simbólica, $[x_1\#y_n\ x_2\#y_{n-1}\ ...\ x_n\#y_1]$. La función debe ser recursiva por la cola y no realizar más de n invocaciones recursivas.

   Su solución debe guardarla en un archivo de nombre `convolucion.oz`.

7. La función de Fibonacci está definida de la siguiente manera:

$$\begin{aligned} fib(n) &= 1 \qquad \text{Si } n < 2 \\ fib(n) &= fib(n-1) + fib(n-2) \qquad \text{Si } n \geq 2 \end{aligned}$$

   - Implemente esta función en la forma intuitiva (computación recursiva). ¿Hasta qué valor de `N` corre en su computador `{Fib N}` ?
   - Implemente esta función iterativamente. Ejecute `{Fib N+1}`, `{Fib 10*N}`, `{Fib 100*N}` donde `N` es el valor encontrado en el ítem anterior.

   Su solución debe guardarla en un archivo de nombre `fibo.oz`.

8. Escriba una función `{Add B1 B2}` que retorna el resultado de sumar los números binarios `B1` y `B2`. Un número binario es representado como una lista de dígitos binarios. El dígito más significativo es el primer elemento de la lista. Por ejemplo:

```
{Add [1 1 0 1 1 0] [0 1 0 1 1 1]} -> [1 0 0 1 1 0 1]
```

   Asuma que las dos listas tienen el mismo número de elementos y defina las funciones auxiliares que considere necesarias.

   Su solución debe guardarla en un archivo de nombre `add.oz`.

9. **Método de Newton Genérico**

   El objetivo de este ejercicio es implementar el método de Newton para hallar raíces de una función cualquiera. Una función estará representada por una fórmula. Una fórmula puede ser un flotante, un átomo (que representa una variable), una tupla binaria (con etiquetas posibles: `s`, `r`, `m`, `d`, `e`, para la suma, resta, multiplicación, división y exponenciación respectivamente) o la tupla unaria con etiqueta `l` (la letra *ele*) para el logaritmo natural.

   Así por ejemplo la función $f(x) = 5k + \ln(3)/(8-x)^x$ podría ser representada en *MOzArt* así:
   `s(m(5.0 k) d(l(3.0) e(r(8.0 x) x)))`

a) Implemente una función `{Derivar F A}` que dada una fórmula `F` y un átomo `A` (que indica cual será la variable de la función), retorne su derivada[1] con respecto a esa variable; todos los demás átomos de `F` deben ser considerados como constantes.

Ej. `{Derivar s(k m(3.0 x)) x}` retorna `s(0.0 s(m(0.0 x) m(1.0 3.0))).`

b) Implemente una función `{Evaluar F A V}` que dada una fórmula `F`, un átomo `A` (que indica cual es la variable) y un valor `V` flotante, calcule la evaluación de la función. Nota: La fórmula de entrada solamente tendrá un tipo átomo, es decir que no dependerá de varias variables, además que siempre podrá ser evaluada.

Ej. `{Evaluar d(2.0 x) x 4.0}` retorna `0.5`.

c) Implemente una función `{Limpiar F}` que dada una fórmula `F`, retorne una fórmula equivalente pero que no contenga ceros no necesarios.

Ej. `{Limpiar s(0.0 s(m(0.0 x) m(3.0 1.0)))}` retorna `m(3.0 1.0)` ó simplemente `3.0`.

d) Implemente una función `{Raiz F X0 BuenaAprox}` que dada una fórmula `F`, valor inicial `X0` y una función binaria `BuenaAprox` retorne una raíz de la función usando el método de Newton. La función `{BuenaAprox F1 V}` recibe una fórmula `F1` y un valor `V` y retorna **`true`** (**`false`**) cuando la evaluación de la fórmula `F1` es sufientemente buena (o no) como para no continuar buscando soluciones mejores.

Su solución debe guardarla en un archivo de nombre `newton.oz`.

[1] Tenga en cuenta las reglas del Cuadro 1.

$$
\begin{aligned}
c &\overset{'}{\to} 0 \qquad (c \text{ es constante}) && (1)\\
x &\overset{'}{\to} 1 && (2)\\
f+g &\overset{'}{\to} f'+g' && (3)\\
f-g &\overset{'}{\to} f'-g' && (4)\\
f\cdot g &\overset{'}{\to} f'\cdot g + f\cdot g' && (5)\\
\frac{f}{g} &\overset{'}{\to} \frac{f'\cdot g - f\cdot g'}{g^2} && (6)\\
\ln(f) &\overset{'}{\to} \frac{f'}{f} && (7)\\
f^g &\overset{'}{\to} f^g * \left(\frac{f'\cdot g}{f} + g'\cdot \ln f\right) && (8)
\end{aligned}
$$

Cuadro 1: Reglas de derivación usadas. Tenga en cuenta que f y g son funciones de la variable de derivación. También tenga en cuenta que $f'(x) = 0$ si f es una constante.