PLP - Primer Parcial - 2^{do} cuatrimestre de 2006

Este examen se aprueba obteniendo al menos **70 puntos**. Poner nombre, apellido y número de órden en cada hoja, y numerarlas. Se puede utilizar todo lo definido en las prácticas y todo lo que se dió en clase, colocando referencias claras.

Ejercicio 1 - Programación funcional (40 puntos)

Consideremos la siguiente representación de matrices:

```
data CompMatrix = Mat Int Int (Int → Int → Int)
```

Que representa una matriz por medio de sus dimensiones (cantidad de filas y columnas, en ese órden) y una función que dados dos índices, devuelve el valor que hay en la posición indicada. La función puede indefinirse o dar cualquier valor si se la llama con parámetros fuera de rango (las filas y columnas se numeran entre 1 y la cantidad respectiva).

Por ejemplo, la siguiente función, dada una cantidad de filas y columnas, devuelve la matriz de la dimensión indicada que contiene todos unos:

```
todosUnosM :: Int → Int → CompMatrix
todosUnosM n m = Mat n m (λ_ _ → 1)
```

a) **(3 puntos)** Escribir una función `idM` que dado un entero N, devuelva la matriz identidad de $N \times N$ [1].

```
idM :: Int → CompMatrix
```

b) **(3 puntos)** Escribir una función `corrM` que dados dos enteros N y M, devuelva la matriz de $N \times M$ cuyos elementos son los primeros $N \cdot M$ naturales escritos de izquierda a derecha y de arriba hacia abajo.

```
corrM :: Int → CompMatrix
```

Por ejemplo `corrM 2 3` debería devolver la representación de la matriz:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

c) **(4 puntos)** Escribir la función `addM` que representa la suma de matrices. Se puede asumir que las matrices recibidas tienen idéntica dimensión. [2]

```
addM :: CompMatrix → CompMatrix → CompMatrix
```

d) **(5 puntos)** Escribir la función `mulM` que representa el producto de matrices. Se puede asumir que la cantidad de columnas del primer parámetro es igual a la cantidad de filas del segundo parámetro. [3]

[1] La matriz identidad es la que tiene 1s en la diagonal principal y 0s en el resto de las posiciones. La diagonal principal es la que va de la esquina superior izquierda (posición (1,1)) a la inferior derecha (posición (N,N))

[2] La suma de matrices se define como una matriz de mismas dimensiones y cada posición es la suma de los valores en la misma posición de los sumandos.

[3] El producto de dos matrices A de $N \times K$ y B de $K \times M$ es una matriz de $N \times M$ que en la posición (i,j) tiene el valor $\sum_{e=1}^{K} A_{ik} \cdot B_{kj}$.

```
mulM :: CompMatrix → CompMatrix → CompMatrix
```

e) **(5 puntos)** Dado el tipo de la matriz por extensión, representada mediante la lista de las filas, escribir la función `compressM` que dada una matriz por comprensión, devuelve su representación por extensión.

```
type ExtMatrix = [[Int]]
extendM :: CompMatrix → ExtMatrix
```

Ejemplo:

```
compressM (corrM 2 3) → [[1,2,3],[4,5,6]]
```

f) **(10 puntos)** Escribir, **sin recursión explícita sobre listas** la función `compressM` que es la reversa de `extendM`, dada una matriz por extensión devuelve su representación por comprensión.

```
compressM :: ExtMatrix → CompMatrix
```

g) **(5 puntos)** Sea `IndMatrix` el tipo de las matrices por inducción:

```
data IndMatrix = Hor IndMatrix IndMatrix | Ver IndMatrix IndMatrix | Single Int
```

El constructor `Single` representa una matriz de un solo elemento y el constructor `Hor`/`Ver` representa dos matrices de igual cantidad de filas/columnas pegadas (la primera a la izquierda y la segunda a la derecha)/(la primera arriba y la segunda abajo). Por ejemplo, el siguiente código:

```
ejMat = Hor
        (Hor
                (Ver (Single 1) (Single 2))
                (Ver (Single 3) (Single 4))
        )
        (Ver (Single 5) (Single 6))
```

representa la matriz:

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

Dar el tipo y la implementación de un esquema de recursión (*fold*) para `IndMatrix`.

h) **(5 puntos)** Usando el esquema del punto anterior, **sin recursión explícita ni pattern matching**, escribir la función `extendIndM` que dada una matriz por inducción devuelve una matriz por extensión.

```
extendIndM :: IndMatrix → ExtMatrix
```

Ejercicio 2 - PCF (30 puntos)

La idea de este ejercicio es extender PCF para soportar la existencia de clases de tipos parecidas a las de Haskell. En este caso una clase de tipos será simplemente un conjunto de tipos que comparten operaciones (como se definen dichas operaciones escapa a las necesidades del ejercicio). Para definir el comportamiento deseado lo ejemplificaremos (en los ejemplos se asumirá PCF extendido con el tipo *float* y *bool*).

letclass $Num = \{nat,\ float\}$ in $((\widetilde{\lambda}x{:}Num.\widetilde{\lambda}y{:}Num.x +_{Num} y)\ 5\ 7)$

letclass $Num = \{nat,\ float\}$ in $((\widetilde{\lambda}x{:}Num.\widetilde{\lambda}y{:}Num.x +_{Num} y)\ 5{,}5\ 7{,}5)$

Ambas expresiones son válidas, la primera reduce a 12 y la segunda a 13,0. Notar que estamos asumiendo que $+_{Num}$ existe sin proveer un mecanismo para definirlo.
Por otro lado,

letclass $Num = \{nat,\ float\}$ in $((\widetilde{\lambda}x{:}Num.\widetilde{\lambda}y{:}Num.x +_{Num} y)\ true\ 7)$

no es válida ya que *bool* no está en la clase Num y por lo tanto no puede aplicársele la función. Por último:

letclass $Num = \{nat,\ float,\ bool\}$ in $((\widetilde{\lambda}x{:}Num.\widetilde{\lambda}y{:}Num.x +_{Num} y)\ true\ 7)$

es válida, ya que en este caso *bool* sí está en la clase Num (notar que las clases para nosotros son un mero grupo definido sintácticamente sobre el cual no ponemos ninguna restricción sobre operaciones ofrecidas, cosa que sí pide Haskell).
El sistema de tipos para esta extensión no se modifica. Sí se agrega sintaxis de la siguiente manera:

$M ::= ... \mid$ letclass $S = \{\sigma_1, ..., \sigma_k\}$ in $M \mid \widetilde{\lambda}x{:}S.M$

Donde S pertenece a un conjunto de cadenas que representan los nombres de las clases de tipos. Notar que el constructor $\widetilde{\lambda}$ agregado es distinto al λ existente ya que en λ lo que viene luego de los dos puntos debe ser un tipo y en $\widetilde{\lambda}$ este caso debe ser un nombre de clase. Se asume que ambos conjuntos (tipos y clases) son disjuntos.
Se pide para la extensión dada, definir:

a) **(14 puntos)** Reglas de tipado

b) **(8 puntos)** Semántica operacional small-step

c) **(8 puntos)** Semántica operacional big-step

Ejercicio 3 - Inferencia de tipos (30 puntos)

Utilizando el árbol de inferencia, inferir el tipo de las siguientes expresiones o demostrar que no son tipables. En caso de ser tipables, dar una cortísima descripción de lo que hace la expresión.

a) **(8 puntos)** $\lambda x.\ \lambda y.\ \lambda z.\ x\ (y\ z)$

b) **(8 puntos)** $\lambda x.\ \lambda y.\ x\ (y\ x)$

c) **(14 puntos)** Extender el algoritmo de inferencia para soportar la inferencia de tipado de PCF con árboles binarios como fue visto en la clase práctica.

La sintaxis de esta extensión es la siguiente:

$\sigma ::= ... \mid AB_\sigma$

$M ::= ... \mid Nil_\sigma \mid Bin_\sigma(M, N, O) \mid Case_{AB_\sigma}\ M\ of\ Nil \to N\ ; Bin(m, n, o) \to O$

Y sus reglas de tipado, las siguientes:

$$\frac{}{\Gamma \rhd Nil_\sigma : AB_\sigma} \qquad \frac{\Gamma \rhd M : AB_\sigma \quad \Gamma \rhd O : AB_\sigma \quad \Gamma \rhd N : \sigma}{\Gamma \rhd Bin_\sigma(M, N, O) : AB_\sigma}$$

$$\frac{\Gamma \rhd M : AB_\sigma \quad \Gamma \rhd N : \tau \quad \Gamma \cup \{m : AB_\sigma, n : \sigma, o : AB_\sigma\} \rhd O : \tau}{\Gamma \rhd Case_{AB_\sigma}\ M\ of\ Nil \to N\ ; Bin(m, n, o) \to O : \tau}$$