Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Mar-07

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Se pidió a un estudiante que demostrara la siguiente propiedad:

> Demostrar que para toda lista `yss` no vacía y para todo elemento `x`, se cumple que
>
> ```
> concat (addFirst x yss) = x : concat yss
> ```
>
> siendo
>
> ```
> concat :: [[a]] ->[a]
> concat [] = []
> concat (xs:xss) = xs ++ concat xss
>
> addFirst x (ys:yss) = (x:ys) : yss
> ```

y su respuesta fue la siguiente:

> La demostración se hará por inducción en la estructura de la lista `yss`. Dado que la lista es no vacía, el caso base ya fue probado y sólo se demostrará el caso inductivo. Para ello, planteamos la hipótesis y tesis inductiva.
>
> **HI)** `concat (addFirst x yss) = x : concat yss`
>
> **TI)** `concat (addFirst x (ys:yss) = x : concat (ys:yss)`
>
> Partiendo de la expresión izquierda de la **TI** llegamos a la parte izquierda, lo cual completa la demostración.
>
> ```
> concat (addFirst x (ys:yss))
>   =  (addFirst.1)
> concat ((x:ys):yss))
>   =  (concat.2)
> (x:ys) ++ concat yss
>   =  ((++).2)
> x : (ys ++ concat yss)
>   =  (concat.2)
> x : (concat (ys:yss)
> ```

a) Escriba al menos tres tipos distintos para la función `addFirst`. ¿Puede encontrar un tipo τ para `addFirst` de tal manera que los anteriores sean casos particulares de τ? Explique.

b) Determine si la respuesta dada por el estudiante es correcta, justifique su respuesta, y en caso de ser incorrecta, dé una versión correcta de la misma.

Ejercicio 2 Lea el siguiente párrafo y discuta en sus propios términos los conceptos involucrados, las consecuencias de lo expuesto, y el grado de concordancia con su propia experiencia:

> "La programación es una disciplina que maneja, fundamentalmente, dos clases distintas de entidades. Por un lado están los datos, representados de alguna manera adecuada para su manipulación, y por el otro están los programas, que se encargan de manipular datos para producir nuevos datos. Reconocer esto es escencial para convertirse en un buen programador."

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Mar-07 *(equivalencias cursada 1999)* fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Se pidió a un estudiante que demostrara la siguiente propiedad:

> Demostrar que para toda lista `yss` no vacía y para todo elemento `x`, se cumple que
>
> ```
> concat (addFirst x yss) = x : concat yss
> ```
>
> siendo
>
> ```
> concat :: [[a]] ->[a]
> concat [] = []
> concat (xs:xss) = xs ++ concat xss
>
> addFirst x (ys:yss) = (x:ys) : yss
> ```

y su respuesta fue la siguiente:

> La demostración se hará por inducción en la estructura de la lista `yss`. Dado que la lista es no vacía, el caso base ya fue probado y sólo se demostrará el caso inductivo. Para ello, planteamos la hipótesis y tesis inductiva.
>
> **HI)** `concat (addFirst x yss) = x : concat yss`
>
> **TI)** `concat (addFirst x (ys:yss) = x : concat (ys:yss)`
>
> Partiendo de la expresión izquierda de la **TI** llegamos a la parte izquierda, lo cual completa la demostración.
>
> ```
> concat (addFirst x (ys:yss))
>   =  (addFirst.1)
> concat ((x:ys):yss))
>   =  (concat.2)
> (x:ys) ++ concat yss
>   =  ((++).2)
> x : (ys ++ concat yss)
>   =  (concat.2)
> x : (concat (ys:yss)
> ```

a) Escriba al menos tres tipos distintos para la función `addFirst`. ¿Puede encontrar un tipo τ para `addFirst` de tal manera que los anteriores sean casos particulares de τ? Explique.

b) Determine si la respuesta dada por el estudiante es correcta, justifique su respuesta, y en caso de ser incorrecta, dé una versión correcta de la misma.

Ejercicio 2 Escriba las siguientes funciones, y muestre que funcionan con los ejemplos dados.

a) `altdoble :: [Int] ->[Int]`, que doble los números de las posiciones impares de la lista de entrada. Por ejemplo, `altdoble [2,5,3,7,7,3] = [4,5,6,7,14,3]`.

b) `altupper :: [Char] ->[Char]`, que alterne mayúsculas y minúsculas en la lista dada. Por ejemplo, `altdoble ''Hola Mundo!'' = ''HoLa mUnDo!''`.

c) `altnums :: [[Int]] ->[Int]`, que reemplace las listas en posiciones impares por su longitud, y las que están en posiciones pares por la suma de sus elementos. Por ejemplo, `altnums [ [1,2], [1,2], [2,3], [2,3] ] = [2,3,2,5]`.

d) Generalice las definiciones anteriores en una función `altmap`, que aplique una función `f` a los elementos impares de una lista, y una función `g` a los pares.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Mar-28

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Se pidió a un estudiante que demostrara la siguiente propiedad:

> Demostrar que para toda lista `yss` no vacía y para todo elemento `x`, se cumple que
>
> ```
> concat (addFirst x yss) = x : concat yss
> ```
>
> siendo
>
> ```
> concat :: [[a]] ->[a]
> concat [] = []
> concat (xs:xss) = xs ++ concat xss
>
> addFirst x (ys:yss) = (x:ys) : yss
> ```

y su respuesta fue la siguiente:

> La demostración se hará por inducción en la estructura de la lista `yss`. Dado que la lista es no vacía, el caso base ya fue probado y sólo se demostrará el caso inductivo. Para ello, planteamos la hipótesis y tesis inductiva.
>
> **HI)** `concat (addFirst x yss) = x : concat yss`
>
> **TI)** `concat (addFirst x (ys:yss) = x : concat (ys:yss)`
>
> Partiendo de la expresión izquierda de la **TI** llegamos a la parte izquierda, lo cual completa la demostración.
>
> ```
> concat (addFirst x (ys:yss))
>   =  (addFirst.1)
> concat ((x:ys):yss))
>   =  (concat.2)
> x : ys ++ concat yss
>   =  (concat.2)
> x : concat (ys:yss)
> ```

a) Escriba al menos tres tipos distintos para la función `addFirst`. ¿Puede encontrar un tipo τ para `addFirst` de tal manera que los anteriores sean casos particulares de τ? Explique.

b) Determine si la respuesta dada por el estudiante es correcta, justifique su respuesta, y en caso de ser incorrecta, dé una versión correcta de la misma.

Ejercicio 2 Escriba las siguientes funciones, y muestre que funcionan con los ejemplos dados.

a) `sumaltdoble :: [Int] ->Int`, que suma los números de la lista de entrada, pero a aquellos en las posiciones pares de la lista los duplica primero. Por ejemplo, `altdoble [2,5,3,7,7,3] = 4+5+6+7+14+3`.

b) `altupper :: [Char] ->[Char]`, que alterne mayúsculas y minúsculas en la lista dada. Por ejemplo, `altdoble ''Hola Mundo!'' = ''HoLa mUnDo!''`.

c) `altnums :: [[Int]] ->Int`, que multiplique la longitud de las listas en posiciones pares, y la suma de los elementos de las que están en posiciones impares. Por ejemplo, `altnums [ [1,2], [1,2], [2,3], [2,3] ] = 2*(3*(2*5))`.

d) Generalice las definiciones anteriores en una función `foldalt`, que realice el fold de la lista de entrada, pero aplicando una función `f` a los elementos pares de la misma, y una función `g` a los impares.

Exprese las funciones de los ítems anteriores usando `foldalt`.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Jun-13

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Dadas las siguientes definiciones:

```
data Tip a = Tip a | Fork (Tip a) (Tip a)

foldT :: (a -> b) -> (b -> b -> b) -> Tip a -> b
foldT f g (Tip x) = f x
foldT f g (Fork t1 t2) = g (foldT f g t1) (foldT f g t2)

minT :: Tip a -> a
minT = foldT id min

replaceT :: b -> Tip a -> Tip b
replaceT v = foldT (const (Tip v)) Fork

repmin t = replaceT (minT t) t
```

a) Dar tres tipos distintos para la función `repmin`. ¿Puede dar un tipo que sea más general que todos los posibles para ella? ¿Cómo? Explique.

b) Mostrar mediante un ejemplo que la de ejecución de `repmin` requiere visitar cada nodo del árbol dos veces.

c) Una versión de `repmin` que visite cada nodo sólo una vez se puede obtener a partir del siguiente código, escribiendo una eureka que involucre a `rm` y derivando una versión recursiva para esta última.

```
repmin' t = let (m,t') = rm m t in t'
```

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Jun-20

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Dadas las siguientes definiciones:

```
data Tip a = Tip a | Fork (Tip a) (Tip a)

foldT :: (a ->b) ->(b ->b ->b) ->Tip a ->b
foldT f g (Tip x) = f x
foldT f g (Fork t1 t2) = g (foldT f g t1) (foldT f g t2)

minT :: Tip a ->a
minT = foldT id min

replaceT :: b ->Tip a ->Tip b
replaceT v = foldT (const (Tip v)) Fork

repmin t = replaceT (minT t) t
```

a) Dar tres tipos distintos para la función `repmin`. ¿Puede dar un tipo que sea más general que todos los posibles para ella? ¿Cómo? Explique.

b) Mostrar mediante un ejemplo que la ejecución de `repmin` requiere visitar cada nodo del árbol dos veces.

c) Una versión de `repmin` que visite cada nodo sólo una vez se puede obtener a partir del siguiente código, escribiendo una especificación para `rm` y derivando una versión recursiva para esta última.

```
repmin' t = let (m,t') = rm m t in t'
```

Escriba una especificación de `rm` que involucre a `replaceT` y `minT` de tal manera que las definiciones de `repmin'` y `repmin` sean equivalentes. Luego derive una versión recursiva para `rm`. Muestre que la versión resultante de `repmin'` recorre el árbol sólo una vez.

Ejercicio 2 Para representar el lambda-cálculo como un tipo algebraico, puede utilizarse la siguiente definición:

```
type Vble = String
data Lam = Var Vble | Lambda Vble Lam | Ap Lam Lam
```

Entre las operaciones típicas del lambda-cálculo, se encuentran la que permite calcular el conjunto de variables libres de un término, y la que permite substituir una variable libre por un término dado. Para representarlas, se pueden utilizar las siguientes definiciones en Haskell:

```
freeVbles :: Lam ->[Vble]
freeVbles (Var x) = [x]
freeVbles (Lambda x e) = freeVbles e // [x]
freeVbles (Ap e1 e2) = freeVbles e1 `union` freeVbles e2

subst :: Vble ->Lam ->Lam ->Lam
subst x e (Var x') = if x==x' then e else (Var x')
subst x e (Lambda x' e') =  if x==x' then Lambda x' e'
                                     else Lambda x' (subst x e e')
subst x e (Ap e1 e2) = Ap (subst x e e1) (subst x e e2)
```

a) Defina una función que permita obtener `freeVbles` y `(subst x e)` como instancias.

b) Defina las funciones `freeVbles` y `(subst x e)` como instancias de la definida en el inciso anterior.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Jul-11

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

a) Hacer una función que calcule `n` pisos del triángulo de Tartaglia (también llamado de Pascal), donde cada número es la suma de los dos que tiene encima. Ejemplo:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
...
```

b) Utilizar `(tartaglia n)` para calcular el número combinatorio `n` tomados de a `i`, que responde a la fórmula

$$\left(\begin{array}{c} n \\ i \end{array} \right) = \left(\begin{array}{c} n-1 \\ i-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ i \end{array} \right)$$

Ejercicio 2 Dadas las siguientes definiciones:

```
from :: Int ->[Int]
from n = n : from (n+1)

data Tip a = Tip a | Fork (Tip a) (Tip a)

change ks t = snd (change' ks t)

change' (k:ks) (Tip _) = (ks, Tip k)
change' ks (Fork t1 t2) =   let(ks', t1') = change' ks t1
                               (ks'', t2') = change' ks' t2
                            in(ks'', Fork t1' t2')

renumber t = change (from 1) t
```

a) Dé el tipo más general posible para `renumber`. ¿Por qué es posible dar un tipo más general? Explique algunas ventajas del polimorfismo.

b) Mostrar mediante un ejemplo que la ejecución de `renumber` requiere producir nodos de una lista que son consumidos inmediatamente. ¿Podría ejecutar `renumber` en un lenguaje estricto? ¿Por qué?

c) Derivar una versión de `renumber` que no genere nodos de listas, y que pueda ser utilizada en un lenguaje estricto. Usar la siguiente Eureka:

```
change'' k t = proc (change' (from k) t)
proc (x:xs, t) = (x,t)
```

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Oct-10

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

A un estudiante se le pidió que dado el tipo `LamExp` para representar expresiones del λ-cálculo, definiera una función `formaNormal`, que dada una expresión, la redujera a su β-forma normal. El código que este estudiante escribió fue el siguiente:

```
data LamExp = Var Int | Lam Int LamExp | App LamExp LamExp

formaNormal :: LamExp ->LamExp
formaNormal (Var x) = Var x
formaNormal (Lam x e) = Lam x (formaNormal e)
formaNormal (App (Lam x e1) e2) = subst x e2 e1
formaNormal (App e1 e2) = App (formaNormal e1) (formaNormal e2)
```

Sin embargo, el JTP no estaba convencido de que este código realmente redujera a forma normal cualquier λ-expresión, por lo que le solicitó a Ud. que lo verificara, y en caso de que no fuera correcto, lo modificase para que lo fuera.

a) Calcule el valor de la expresión

```
let id = Lam 1 (Var 1)
    e = App (App id id) (Var 2)
 in formaNormal e
```

b) Determine si el código presentado para la función `formaNormal` es correcto (o sea, siempre calcula la forma normal de su argumento). En caso afirmativo, justifique, y en caso negativo, corrija el código para que funcione correctamente (si este fuera el caso, ¡evite recorrer la expresión más de una vez!).

c) ¿Qué puede decir del uso de la recursión en el código de `formaNormal`? ¿Se podría representar `formaNormal` como un `fold` tradicional? Si responde que se podría, defínala como `fold`; si responde que no, explique por qué, defina una función que trabaje según el principio de `fold` pero que corrija el problema, y defina `formaNormal` en base a esta función.

Ejercicio 2 Considere las siguientes definiciones:

```
subst :: (a ->b ->c) ->(a ->b) ->a ->c
subst f g x = f x (g x)

id :: a ->a
id x = x

(>->) :: (a ->b) ->(b ->c) ->a ->c
(g >->f) x = let y = g x in f y

addid f x = f id x

dup f x = f x x
```

a) Determine el tipo más general de `addid` y `dup`. Explique que significa "tipo más general", y por qué puede obtenerlo.

b) Demuestre que `subst >->addid = dup`.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Nov-07

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

Considere el lenguaje de las fórmulas de la Lógica de Primer Orden, definido por la siguiente gramática:

$$
\begin{array}{rcll}
< folexp > & := & < atom > & | \quad \neg < folexp > \\
 & | & < folexp > \wedge < folexp > & | \quad < folexp > \vee < folexp > \\
 & | & (\forall < var >)(< folexp >) & | \quad (\exists < var >)(< folexp >) \\
< atom > & := & < pred > (< vars >) & \\
< var > & := & < var > \mid < var >, < vars > & \\
< pred > & := & P \mid Q \mid R \mid \ldots & \\
< var > & := & x \mid y \mid z \mid \ldots &
\end{array}
$$

Así, por ejemplo, $(\forall x)(P(x) \vee Q(x, x))$ es una fórmula de este lenguaje.

Sabemos que los cuantificadores $\forall$ y $\exists$ son *binders* que ligan la variable que los sigue, y que su alcance es la expresión que se encuentra a continuación entre paréntesis. De esta manera, en la expresión anterior, x es la variable ligada por el $\forall$ y $P(x) \vee Q(x)$ es el alcance del mismo.

a) Identifique las variables libres y ligadas en las siguientes fórmulas, y los binders a los que se ligan:

- $P(x) \wedge (\exists x)(Q(x) \vee (\forall x)(R(x) \wedge P(y)))$
- $(\forall y)(\exists z)(P(x, y, z) \vee (\forall y)(Q(z, y)))$

b) Defina una función FV que dada una expresión retorne el conjunto de sus variables libres.

c) Defina una noción de sustitución de una variable libre por una fórmula de tal manera que se evite la captura de variables.

d) Defina una noción de α-conversión para este lenguaje, y escriba al menos dos expresiones α-equivalentes a las fórmulas del primer inciso.

e) Escriba un tipo algebraico que sirva para representar las fórmulas de este lenguaje, y una función `fv` que represente a la función FV del inciso b).

f) Si no hizo `fv` utilizando un esquema de recursión, inténtelo de nuevo. ¡Y no se olvide de dar los tipos de las funciones que define!

Ejercicio 2 Dadas las siguiente definiciones:

```
rep :: Int ->[a] ->[a]
rep 0 xs = []
rep n xs = xs ++ rep (n-1) xs

rep' :: Int ->[a] ->[a]
rep' 0 xs = []
rep' n xs = rep' (n-1) xs ++ xs
```

a) Demuestre que, si `n`$\neq$`0`, `rep n xs = rep (n-1) xs ++ xs`.

b) Demuestre que `rep = rep'`.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Nov-28

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

Un *Tone* es un aparato que consiste de una pirámide de celdas interconectadas, por donde pueden pasar bolitas. Cada celda tiene dentro un 'interruptor', que se modifica cada vez que una bolita pasa por la celda, cambiando la interconexión entre celdas. El mecanismo funciona tirando una bolita por la celda superior de la pirámide, la cual va atravesando las celdas interconectadas, hasta llegar a la base. Cada vez que una bolita pasa por una celda, el interruptor de ésta cambia de estado, conectando la celda actual con otra diferente. Así, la siguiente bolita que pase por esa celda será desviada hacia otro lado, y alcanzará un lugar diferente de la base. En el caso de representar un Tone plano, tendríamos un mecanismo como el que se grafica en la figura 1.

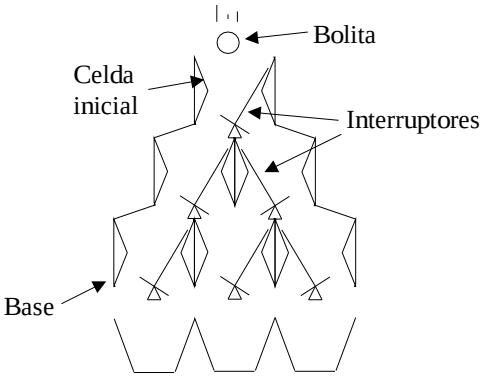

Figura 1: Tone representable en el plano.

Para modelar un Tone en Haskell, se pueden definir los tipos abstractos `Tone`, `Celda` y `Dir`, que representen al aparato, a las celdas del aparato, y a las diferentes direcciones en que se puede encontrar un interruptor, respectivamente. Para estos tipos de datos, se proveerán las siguientes funciones:

- `toneInicial :: Int ->Tone`, función total que devuelve diferentes tones, según el número suministrado como argumento.
- `celdaInicial :: Tone ->Celda`, función total que dado un tone, devuelve la celda inicial del mismo.
- `celdaEnLaBase :: Tone ->Celda ->Bool`, función total que dado un tone y una celda, determina si la misma se encuentra en la base del tone o no.
- `proximaCelda :: Tone ->Celda ->Celda`, función parcial que dado un tone `t` y una celda `c`, retorna la celda a la cual está conectada `c`, según el estado de su interruptor en `t` (o sea, dice a qué celda iría una bolita que pasa por `c`). No está definida si la celda se encuentra en la base.
- `cambiarCelda :: Celda ->Tone ->Tone`, función total que dado una celda `c` y un tone `t`, devuelve el tone resultante de cambiar el estado de `c` en `t`, como si una bolita la hubiera atravesado.
- `direccion :: Tone ->Celda ->Dir`, función total que dado un tone `t` y una celda `c`, retorna la dirección en la que apunta el interruptor de `c` en `t`.
- `eqDir :: Dir ->Dir ->Bool`, función total que dadas dos direcciones, determina si son iguales.
- `celdas :: Tone ->[ Celda ]`, función total que devuelve una lista con todas las celdas del tone dado, en orden arbitrario.

Usando estas funciones pueden definirse funciones tales como `arrojarBolita`, `simular` y `todasHacia`.

```
arrojarBolita :: Tone ->Tone
arrojarBolita t = tiroBasico (celdaInicial t) t

simular :: Int ->[ Tone ]
simular = arrojarBolitas . toneInicial

todasHacia :: Dir ->Tone ->Int
todasHacia = cuantasHasta . estanTodasPara

-- Auxiliares
tiroBasico p t = let t' = cambiarCelda p t
                 in if celdaEnLaBase p t
                    then t'
                    else tiroBasico (proximaCelda p t) t'

arrojarBolitas = iterate arrojarBolita

estanTodasPara :: Dir ->Tone ->Bool
estanTodasPara d t = verificar d t (celdas t)
verificar d t = chequeo d . obtenerDirs t
chequeo d = foldr (\d' b ->b && eqDir d d') True
obtenerDirs = map . direccion

cuantasHasta :: (Tone ->Bool) ->Tone ->Int
cuantasHasta p = length . takeWhile (not . p) . arrojarBolitas
```

a) Implemente los tipos abstractos `Tone`, `Celda` y `Dir` para representar tones como el mostrado en la figura 1. O sea, dé tipos de datos que implementen los tipos pedidos, y también implemente las funciones sobre estos tipos.

b) Muestre que las implementaciones dadas en el inciso anterior respetan las condiciones de totalidad o parcialidad indicadas en la descripción.

c) Demuestre que

`cuantasHasta p t = if not (p t) then 1 + cuantasHasta p (arrojarBolita t) else 0.`

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2001-Dic-19

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1 Dado el tipo

```
data Arbol a = Empty | Nodo (Arbol a) a (Arbol a)
```

y considerando el árbol

```
aej = Nodo (Nodo Empty 2 (Nodo Empty 3 Empty))
           1
           (Nodo (Nodo Empty 5 Empty)
                 4
                 (Nodo Empty 6 Empty))
```

a) Escribir una función `nivelCompletoMasProfundo :: Arbol a ->[a]` que dado un árbol, devuelva la lista de los elementos que se encuentran en el nivel más profundo que contenga la máxima cantidad de elementos posibles. Por ejemplo `nivelCompletoMasProfundo aej = [2,4]`.

b) Escribir una función `frontera :: Arbol a ->[a]` que dado un árbol, retorne todos los elementos que residen en nodos que no tienen ningún descendiente. Por ejemplo `frontera aej = [3,5,6]`.

c) Escribir una función `trunchar :: Arbol a ->Arbol a` que elimine todas las ramas necesarias para transformar el árbol en un árbol completo con la máxima altura posible. Por ejemplo, `trunchar aej = Nodo (Nodo Empty 2 Empty) 1 (Nodo Empty 4 Empty)`.

d) Escribir una función `nivelMax :: Arbol a ->Int` que dado un árbol, devuelva la profundidad del nivel completo más profundo. Por ejemplo `nivelMax aej = 2`.

Ejercicio 2

a) Determine los tipos de las funciones `atFirst` y `atLast`, siendo sus definiciones

```
atFirst x xss = [] : map (x:) xss

atLast [] x = []
atLast [ys] x = [ys, ys++[x]]
atLast (ys:yss) x = ys : atLast yss x
```

b) Demostrar que para toda lista `yss` y para todos elementos `x` y `x'`, se cumple que `atFirst x (atLast yss x') = atLast (atFirst x yss) x'`.

c) Demostrar que `initsR = initsL`, siendo

```
initsR :: [a] ->[[a]]
initsR [] = [[]]
initsR (x:xs) = [] : map (x:) (initsR xs)

initsL :: [a] ->[[a]]
inistL = foldl atLast [[]]
```

(**Ayuda:** Divida su demostración en dos partes, de tal manera que una sea una inducción sencilla y la otra use algún resultado auxiliar.)

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2002-Abr-24

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

El objetivo de esta evaluación es que considere la noción de máquina abstracta (que se explicará a continuación) y que la traduzca en un lenguaje que conoce (Haskell), mostrando de esta manera que efectivamente sabe programar en un lenguaje funcional y que puede expresar operaciones no triviales con él y manipularlas adecuadamente.

Considere el lenguaje de las expresiones aritméticas definido por la siguiente gramática:

$$e \quad := \quad n \mid e + e \mid e - e$$

(¡si no sabe gramáticas, no desespere! Son simplemente una forma cómoda (inductiva) de decir que las constantes numéricas son expresiones, y que la suma y resta de expresiones es una nueva expresión – a estas alturas debería conocer algo muuuy parecido.)[1] Para evitar líos con los paréntesis, se asumen las convenciones clásicas de precedencia de operaciones, y que al escribir una expresión aritmética se pueden usar paréntesis para evitar ambigüedades; esto se conoce en el área de teoría de la computación como *sintaxis abstracta*. Ejemplos de expresiones aritméticas son:

$$5 \qquad (4-2)+3 \qquad 2+3 \qquad (4-2)+(5-2)$$

La expresiones aritméticas pueden representarse en Haskell definiendo un tipo algebraico `Expr`.

Para calcular el valor de una expresión aritmética, se pueden usar diversas formas; nosotros vamos a considerar dos: el método recursivo directo y la compilación a bytecodes sobre una máquina abstracta.

El método directo consiste en escribir una función `eval :: Expr -> Int`, por recursión sobre la estructura de `Expr`; esta forma es la implementación en Haskell de un método para dar significado a estructuras sintácticas, conocido como *semántica denotacional*, que también debería preguntarse si conoce o no, y como podría hacer para conocerla...

La compilación a bytecodes consiste en definir cierta "máquina", junto con una serie de instrucciones de como hacerla funcionar, y traducir la expresión aritmética a una secuencia de tales instrucciones (conocidas como "bytecodes"), de tal manera que ejecutar los bytecodes en la máquina retorna el resultado esperado. Para las expresiones aritméticas, basta con usar una pila de enteros como máquina (definida mediante un tipo abstracto `Pila`, que se describe más adelante), y los bytecodes como secuencia de instrucciones del siguiente conjunto:

$$\text{PUSH } n \qquad \text{ADD} \qquad \text{SUB}$$

(se llaman bytecodes pues se pueden representar de manera compacta utilizando un byte por cada instrucción, más dos bytes para los argumentos). El significado de cada instrucción es una serie de acciones a realizar sobre la máquina:

- PUSH n apila el número n
- ADD desapila los dos primeros elementos y apila su suma
- SUB desapila los dos primeros elementos y apila su resta (resta el primer elemento desapilado, y, al segundo, x, apilando $x - y$).

La ejecución de una secuencia de bytecodes consiste en realizar las acciones indicadas por cada instrucción de la secuencia sobre una pila vacía, y luego retornar el elemento en el tope de la pila.

La compilación de una expresión aritmética se define mediante la siguiente función recursiva:

$$\begin{array}{lcl} \mathcal{C}(n) & = & \text{PUSH } n \\ \mathcal{C}(e_1 + e_2) & = & \mathcal{C}(e_1), \mathcal{C}(e_2), \text{ADD} \\ \mathcal{C}(e_1 - e_2) & = & \mathcal{C}(e_1), \mathcal{C}(e_2), \text{SUB} \end{array}$$

donde la coma representa la secuenciación de instrucciones. Por ejemplo, la expresión $\mathcal{C}((4-2)+3)$ da como resultado la secuencia PUSH 4,PUSH 2,SUB,PUSH 3,ADD.

Los ejercicios que tiene que realizar lo guiarán en la definición de los dos mecanismos de evaluación de expresiones, y en la prueba de que ambos son equivalentes.

Las pilas utilizadas como máquinas en los ejercicios se definen como un tipo abstracto de datos, `Pila a`, cuyas operaciones son:

[1] ¡Ojo! El consejo de no desesperar sólo vale para este examen... Al terminar debería preguntarse por qué no sabe gramáticas, y como podría hacer para remediarlo, pues son extremadamente importantes. Consulte a su docente amigo.

- `empty`, que representa a una pila vacía.
- `isEmpty` p, que testea si la pila p está vacía o no.
- `push` a p, que representa la pila resultante de apilar el elemento a en la pila p.
- `pop` p, que representa la pila resultante de desapilar el primer elemento de p.
- `top` p, que representa el primer elemento de p.

Las operaciones `pop` p y `top` p no están definidas si p es la pila vacía (son operaciones parciales). Se sabe que las operaciones de una pila satisfacen las siguientes propiedades:

`top` (`push` a p) $= a$
`pop` (`push` a p) $= p$
`isEmpty empty` $=$ `True`
`isEmpty` (`push` a p) $=$ `False`
`push` (`top` p) (`pop` p) $= p$, si p no está vacía

Ejercicio 1 Este ejercicio define el significado denotacional de una expresión aritmética y el resultado de compilar la misma en una secuencia de bytecodes.

a) Defina tipos de datos para representar expresiones aritméticas (`Expr`), instrucciones de la máquina (`Instr`), y secuencias de bytecodes (`ByteCode`).

Explique la conveniencia de definirlos de esa manera y no de otra. Explique también qué es un tipo algebraico.

b) Escriba la representación en `Expr` de las expresiones dadas como ejemplo al principio de este texto.

c) Defina una función `eval :: Expr -> Int`, que devuelva el resultado de evaluar una expresión aritmética. Muestre la reducción de `eval` aplicada a la expresión más grande del item anterior.

¿Qué orden de reducción usó? ¿Por qué? Explique las características del orden utilizado.

d) Defina una función `compile :: Expr -> ByteCode`, que compile una expresión aritmética e en la secuencia correspondiente de bytecodes indicada por $\mathcal{C}(e)$. Muestre la reducción de `compile` aplicada a la misma expresión usada en el item anterior.

Ejercicio 2 Este ejercicio define el significado de una secuencia de bytecodes.
Defina funciones:

a) `execInstr :: Instr -> Pila -> Pila`, que implemente la acción de una instrucción.

b) `execute :: ByteCode -> Pila -> Pila`, que implemente la acción de una secuencia de bytecodes.

c) `run :: ByteCode -> Int`, que implemente la evaluación de los bytecodes correspondientes a la compilación de una expresión aritmética.

Describa como piensa definir sus funciones. (O sea, no alcanza con dar el código pelado para que el ejercicio esté completo...)

Ejercicio 3 Este ejercicio prueba que el valor calculado por los bytecodes resultado de la compilación de una expresión aritmética coincide con el valor denotacional de la misma, probando de esta manera que la implementación de expresiones aritméticas usando máquinas abstractas es correcta.
¡No olvide que debe proceder paso a paso! Empezar diciendo "caso base" invalida el ejercicio...

a) Demuestre que `execute [i]` = `execInstr i`.

b) Demuestre que para todo `p`, `execute (is ++ is') p` = `execute is' (execute is p)`.

c) Demuestre que para todo `p`, `execute (compile e) p` = `push (eval e) p`.
(**Happy Hour:** ¡Hay que usar los items a) y b) en los casos de la inducción!)

d) Demuestre que `eval e = run (compile e)`.
(**Happy Hour:** Fácil, fácil. Mire los items anteriores y la definición de `run`...)

Ejercicio 4 Implemente el tipo abstracto `Pila` y demuestre que las leyes de las mismas se satisfacen en dicha implementación. ¿Es realmente abstracto el código que escribió? ¿Por qué?

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2002-May-22

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

Considere las siguientes definiciones y funciones:

```
data T a = E | T (T a) a (T a)

inorder t = iaux t []
iaux E xs = xs
iaux (T a x b) xs = iaux a (x: iaux b xs)

addTLast x E = T E x E
addTLast x (T a y b) = T a y (addTLast x b)

addTFirst x E = T E x E
addTFirst x (T a y b) = T (addTFirst x a) y b
```

a) Demuestre que para cualquier `t :: T a` finito y cualquier `x :: a`,

```
inorder (addTLast x t) = inorder t ++ [x]
```

Atención: es posible que precise algún resultado auxiliar, que deberá enunciar y demostrar.

b) Demuestre que para cualquier `t :: T a` finito y cualquier `x :: a`,

```
inorder (addTFirst x t) = x : inorder t
```

Atención: este resultado sale como corolario de una propiedad más general sobre `iaux`...

c) ¿Qué diferencias observa entre ambas demostraciones en cuanto al uso de la inducción? Explique.

d) Escriba una función `addTAll :: a -> T a -> T a` que agregue un elemento `x :: a` como hijo de cada hoja del árbol. Por ejemplo,

```
addTAll 3 (T (T E 2 E) 1 E) = T (T (T E 3 E) 2 (T E 3 E)) 1 (T E 3 E)
```

e) Derive una función `inter` que cumpla que para cualquier `t :: T a` finito y cualquier `x :: a`,

```
inorder (addTAll x t) = inter x (inorder t),
```

Ayuda: utilice la siguiente eureka

a) `inter x (iaux t []) = iaux (addTAll x t) []`

b) `inter x (iaux t (z:zs)) = iaux (addTAll x t) (z: inter x zs)`

¿Qué tipo tiene `inter`? ¿Es el único posible? Explique.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2002-Jun-19

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Ejercicio 1

La forma tradicional de entender a los tipos de datos de un lenguaje funcional es como conjuntos de valores con características comunes. Sin embargo, existen formas alternativas de intepretar a los tipos. Una de ellas es la de entender a los tipos como las *proposiciones* de la lógica proposicional. En esta visión, por ejemplo, la formación de pares se entiende como la conjunción (o sea, el tipo `(P,Q)` representa a la proposición $P \wedge Q$), el tipo `data Either a b = Left a | Right b` se entiende como la disyunción (o sea, el tipo `Either P Q` representa a la proposición $P \vee Q$), y el tipo de las funciones representa a la implicación (o sea, `P -> Q` representa a la proposición $P \Rightarrow Q$).

Esta analogía permite entender a los elementos de un tipo dado como una prueba de que la proposición correspondiente es verdadera; de esta manera, probar que una proposición es verdadera es equivalente a encontrar un elemento del tipo correspondiente. Por ejemplo, para probar que la proposición $P \Rightarrow P$ es verdadera para todo P, construímos un elemento del tipo `a -> a`; así, la función `\x -> x` es una *prueba* de que $P \Rightarrow P$ es verdadera para todo P. De la misma manera, para probar que la proposición $P \Rightarrow P \vee Q$ es verdadera para todos P y Q, construímos un elemento del tipo `a -> Either a b` – por ejemplo, `\x -> Left x`.

Para que la identificación entre proposiciones y tipos fuera completa, debería poder definirse un tipo que represente la negación de una proposición. Ello se consigue de la siguiente manera. Suponga que se extiende el lenguaje Haskell de tal manera de poder definir un tipo de datos *sin constructores*, por ejemplo, mediante la siguiente cláusula:

```
data Empty = {}
```

(¡observe que no se podrán construir elementos definidos para este tipo!). En la visión de tipos como conjuntos de valores, el tipo `Empty` representa al conjunto vacío de valores definidos, y en la visión de tipos como proposiciones de la lógica, este tipo representa a una contradicción (o sea, una proposición que siempre es falsa, o de manera equivalente, que no tiene pruebas).

Habiendo definido el tipo `Empty`, podemos proponer una definición tentativa para la negación:

```
type No a = (a -> Empty).
```

De esta manera, el tipo `No P` representa a la proposición $\neg P$. Los elementos de tipo `No a` son funciones que dado un elemento de tipo `a` retornan un elemento del conjunto vacío. A pesar de que esta forma de explicarlo da la sensación que no se puede construír ninguna función de tipo `No a`, el siguiente ejemplo muestra que eso no es cierto: `\(a,na) -> na a :: No (a, No a)`. Recordando la interpretación de cada constructor de tipos como proposiciones, podemos ver que este último tipo representa a la proposición $\neg(P \wedge \neg P)$, cualquiera sea P, de la cual sabemos que es verdadera (no es cierto que una proposición y su negación pueden ser ciertas al mismo tiempo).

a) Para cada una de las proposiciones enunciadas, establezca cuál es el tipo que la representa.

- **a)** $(P \wedge Q) \Rightarrow P$
- **b)** $(P \vee P) \Rightarrow P$
- **c)** $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$
- **d)** $\neg\neg\neg P \Rightarrow \neg P$

b) "Demuestre" cada una de las proposiciones anteriores (o sea, encuentre elementos del tipo correspondiente).

(**Ayuda:** expanda totalmente cada `No` a su definición como función.)

(**Ayuda adicional:** en el último ítem, puede resultar útil pensar primero en construír una prueba de $\neg\neg P$ a partir de una prueba de P)

c) Suponga que cambiamos la definición de `No` por `data No a = No (a->Empty)`. Reescriba las definiciones dadas en el ítem anterior para que los tipos coincidan con esta nueva definición.

d) Explique que significado tendría una prueba de la proposición $P \vee \neg P$ en el sentido de prueba dado antes. ¿Podría construír tal prueba? ¿Por qué?

Ejercicio 2

a) Determine el tipo de `addLast`, siendo la definición

```
addLast [ys] x = [ys++[x]]
addLast (ys:yss) x = ys : addLast yss x
```

Explique si hay otros tipos para `addLast`, y que relación tienen con el que usted dió.

b) Demostrar que para toda lista `yss` no vacía y para todo elemento `x`, se cumple que `concat (addLast yss x) = concat yss ++ [x]` siendo

```
concat :: [[a]] ->[a]
concat = foldr (++) []
```

Ejercicio 3

a) Sintetice una definición de `f` en términos de `map`, `filter` y `concat`.

```
f xs = [ x^2 + y^2
       | x <- xs, even x,
         y <- reverse xs ]
```

b) Sintetice una versión recursiva de `f` a partir de la obtenida en el ítem anterior.

Ejercitación Programación Funcional

Compilación: 3 Sept 2002

Ejercicios de la mesa de: 2002-Jul-17

fcional@sol.info.unlp.edu.ar

Importante: Practicar utilizando problemas tomados en finales pasados puede ser un buen ejercicio para medir cuánto se ha aprendido de la materia. Sin embargo, debe tenerse en cuenta que:

- Algunos finales pueden resultar demasiado extensos o difíciles; en estos casos probablemente se hayan hecho aclaraciones o ayudas durante la mesa de final, o corregido dejando ejercicios opcionales.
- ¡Aprender la solución de todos y cada uno de los finales no garantiza conocer a fondo la materia! Cada examen final es diferente a los demas, aunque evalua lo mismo: saber aplicar los conceptos dictados en el curso.

Por último, recuerde que los finales se realizan a libro abierto, que se deben definir o demostrar todas las funciones y propiedades utilizadas durante su resolución (excepto las vistas en clase o en la bibliografía, a las cuales se debe hacer referencia). Además el objetivo es medir cuánto se sabe del tema, por lo que puede ser conveniente no limitarse a dar las resoluciones puntuales de cada inciso sino también explicar lo que se hace, ilustrar con ejemplos, discutir alternativas, etc.

Un árbol binario de búsqueda se puede definir sobre el tipo algebraico de los árboles binarios, abstrayendo los constructores, y permitiendo utilizar solamente las operaciones para crear un árbol vacío, insertar un elemento a un árbol, y consultar si un elemento pertenece a un árbol. Esto se puede implementar en Haskell de la siguiente manera:

```
module BST (BST, empty, insert, member) where
  data BST a = E | T (BST a) a (BST a)

  empty :: BST a
  empty = E

  insert :: a -> BST a -> BST a
  insert x E           = N E x E
  insert x (T t1 y t2) = if x < y
                         then T (insert x t1) y t2
                         else if x > y
                               then T t1 y (insert x t2)
                               else T t1 y t2

  member :: a -> BST a -> Bool
  member _ E           = False
  member x (T t1 y t2) = if x < y
                         then member x t1
                         else if x > y
                               then member x t2
                               else True
```

Ejercicio 1

Con la implementación dada, la función `member` debe realizar, en el peor de los casos, $2d$ comparaciones, siendo d la profundidad del árbol en cuestión. Si se llevase cuenta de un elemento *candidato*, que pudiera ser el elemento buscado, podría reducirse el número de comparaciones a $d + 1$. Esto se implementa con la siguiente función:

```
  memberF :: a -> BST a -> Bool
  memberF x t = membF x t Nothing

  membF _ E Nothing     = False
  membF x E (Just y)    = x==y
  membF x (T t1 y t2) c = if x < y
                          then membF x t1 c
                          else membF x t2 (Just y)
```

Sin embargo, no está claro que ambas funciones realicen el mismo trabajo. Por ello, se le encomienda a usted que pruebe tal hecho, y mejore la función `membF`, para lo cual debe completar los siguientes items:

a) Escriba tres tipos distintos para la función `membF`. Escriba un tipo del cual los tres anteriores sean casos particulares. Describa la característica del sistema de tipos que le permite esto.

b) Demuestre la siguiente propiedad

$$P(\texttt{x},\texttt{t},\texttt{z}) \equiv$$
$$(\texttt{membF x t Nothing = member x t})$$
$$\wedge\ ((\forall y \in t.z < y)(\Rightarrow \texttt{membF x t (Just z) =(x==z) || member x t}))$$

Ayudas:

- Asuma que `t` es un árbol binario de búsqueda (o sea, que todos los elementos del hijo izquierdo (derecho) son menores (mayores) que el elemento en la raíz, y que los subárboles también son árboles de búsqueda). Observe que esto se cumple por ser `BST` un tipo abstracto.
- Debe demostrar *ambas* propiedades juntas, o no podrá hacerlo.
- Las siguientes igualdades le serán útiles:

  A) si `x<z` es falso, entonces `(if x >z then e else True) = (x==z) || e`

  B) `False || e = e`

  C) `e || False = e`

  D) `e || (if b then e1 else e2) = if b then e||e1 else e||e2`

c) La función `membF` es ineficiente, pues crea constructores del tipo `Maybe` los cuales son eliminados inmediatamente por el pattern matching del caso recursivo. Derive una solución que no tenga esta característica, utilizando la siguiente eureka:

```
membFNothing x t = membF x t Nothing
membFJust x t y = membF x t (Just y)
```

Ejercicio 2

El código dado para la función `insert` examina cada nodo en el camino desde la raíz al elemento a insertar, y siempre crea copias de los mismos (o sea, en el lado derecho *siempre* aparece `T a' y' b'`, para algún `a'`, `b'` y `y'`). En el caso de que el elemento ya exista, tal duplicación no es necesaria, ya que se hubiera podido retornar el mismo árbol (en lugar de uno idéntico a él). Esta optimización puede obtenerse mediante una función `insert' :: a -> BST a -> Maybe (BST a)` que retorne `Nothing` en los casos donde el árbol a devolver sea el mismo que el original. Entonces `insert` puede escribirse como:

```
insert x t = case (insert' x t) of
               Nothing -> t
               Just t' -> t'
```

a) Complete la implementación de `insert'` basada en el siguiente código:

```
insert' x E = ...
insert' x (T a y b) = if x > y
                        then fmapM ... (insert' x a)
                        else if x < y
                               then ...
                               else ...

fmapM f mt = case mt of
               Nothing -> mt
               Just t  -> Just (f t)
```

b) (adicional) Implemente una versión de `insert'` que realice solamente $d + 1$ comparaciones en el peor caso.