Práctica 3

Programación Funcional, UNQ

Representación de datos con funciones

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*
- *No dude en manifestar observaciones y críticas sobre los ejercicios de esta práctica, que con gusto serán recibidas por los docentes.*
- *Los ejercicios del anexo pueden obviarse, pero recuerde que aportan una compresión más profunda sobre los temas que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

1. Lambda Booleans

Dada las siguientes definiciones que representan a los booleanos mediante funciones:

```
type BoolLam = a -> a -> a

trueLam  = \x y -> x
falseLam = \x y -> y
```

Y considerando las siguientes operaciones simples sobre esta representación de los booleanos:

```
ifThenElseLam = \x -> x
notLam = \x -> ifThenElseLam x falseLam trueLam
```

Definir las siguientes operaciones (que se comportan como sus contrapartes booleanas):

a) `orLam :: BoolLam -> BoolLam -> BoolLam`

b) `andLam :: BoolLam -> BoolLam -> BoolLam`

2. Lambda Pairs

Dada la siguiente definición de `pairLam` que representa a los pares mediante funciones (e.g. `(pairLam 1 True)` es una expresión que denota el par `(1,True)`):

```
data Either a b = Left a | Right b

type Projector = a -> b -> Either a b
left  = \x y -> Left x
right = \x y -> Right y

type PairLam a b :: a -> b -> Projector -> Either a b
pairLam = \x y p -> p x y
```

Definir las siguientes operaciones (que se comportan como las proyecciones de las componentes del par):

a) `fstLam :: PairLam a b -> Either a b`, (e.g. `fstLam (pairLam 1 True)` debería retornar `(Left 1)`).

b) `sndLam :: PairLam a b -> Either a b`, (e.g. `fstLam (pairLam 1 True)` debería retornar `(Right True)`).

3. Lambda Sets

Defina las siguientes operaciones para conjuntos representados por extensión (como `[a]`) y alternativamente por comprensión (como predicado `a -> Bool`) cuando sea posible.

a) `union :: Set a -> Set a -> Set a`

b) `intersect :: Set -> Set a -> Set a`

c) `complement :: Set a -> Set a`

d) `cardinal :: Set a -> Nat`

4. Fuzzy Sets

Un conjunto difuso (o fuzzy set en inglés) indica para cada elemento `x` un nivel de certeza de que `x` pertenezca al conjunto (i.e., un número real entre 0 y 1, donde 0 indica que con certeza el elemento no pertenece al conjunto y 1 indica que con certeza el elemento sí pertenece al conjunto). Considere una representación de fuzzy sets mediante funciones:

```
type Fuzzy :: a -> Float
```

Y defina la siguientes operaciones:

a) `belongs :: Fuzzy a -> a -> Float`

b) `complement :: Fuzzy a -> Fuzzy a`

c) `union :: Fuzzy a -> Fuzzy a -> Fuzzy a`

d) `intersect :: Fuzzy -> Fuzzy a -> Fuzzy a`

5. Church Numerals*

Dada la siguiente representación de números mediante funciones (donde un número `n` se representa mediante `n` composiciones de una función `f`, es decir, la repetición de `n` aplicaciones sucesivas de la función `f`):

```
type Numeral a = (a -> a) -> a -> a

zero = \f x -> x  -- 0 aplicaciones de la funcion f

succ n = \f x -> f (n f x)  -- 1 aplicacion mas de la funcion f
```

Defina la siguientes operaciones:

a) `church :: Nat -> Numeral Nat`, que retorna el numeral que representa a un número dado.

b) `unchurch :: Numeral Nat -> Nat`, que retorna el entero representado por un numeral.

c) `add :: Numeral a -> Numeral a -> Numeral a`, que retorna el numeral que representa la suma de otros dos.

d) `mul :: Numeral a -> Numeral a -> Numeral a`, que retorna el numeral que representa el producto de otros dos.