Programación Funcional

Ejercicios de Práctica Nro. 10

Inducción/Recursión IV

Aclaraciones:

- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las 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 evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Ejercicio 1) Dada la siguiente representación de un lenguaje de expresiones numéricas con variables:

```
data NExp = Var Variable
          | NCte Int
          | NBOp NBinOp NExp NExp

data NBinOp = Add | Sub | Mul | Div | Mod | Pow
type Variable = String
```

y el TAD **`Memoria`** cuya interfaz es la siguiente:

- **`enBlanco :: Memoria`**, que describe una memoria vacía.
- **`cuantoVale :: Variable -> Memoria -> Maybe Int`**, que describe el número asociado a la variable dada en la memoria dada.
- **`recordar :: Variable -> Int -> Memoria -> Memoria`**, que la memoria resultante de asociar el número dado a la variable dada en la memoria dada.
- **`variables :: Memoria -> [Variable]`**, que describe las variables que la memoria recuerda.

- implementar las siguientes funciones:
  i. **`evalNExp :: NExp -> Memoria -> Int`**, que describe el número resultante de evaluar la expresión dada a partir de la memoria dada.
  ii. **`cfNExp :: NExp -> NExp`**, que describe una expresión con el mismo significado que la dada, pero simplificada y reemplazando las subexpresiones que no dependan de la memoria por su expresión más sencilla. La resolución debe ser exclusivamente *simbólica*.

- demostrar la siguiente propiedad:

```
evalNExp . cfNExp = evalNExp
```

Ejercicio 2) Dada la siguiente representación de un lenguaje de expresiones booleanas:

```
data BExp = BCte Bool
          | Not BExp
          | And BExp BExp
          | Or BExp BExp
          | ROp RelOp NExp NExp

data RelOp = Eq   | NEq   -- Equal y NotEqual
           | Gt   | GEq   -- Greater y GreaterOrEqual
           | Lt   | LEq   -- Lower y LowerOrEqual
```

- implementar las siguientes funciones:
  i. **`evalBExp :: BExp -> Memoria -> Bool`**, que describe el booleano que resulta de evaluar la expresión dada a partir de la memoria dada.
  ii. **`cfBExp :: BExp -> BExp`**, que describe una expresión con el mismo significado que la dada, pero reemplazando las subexpresiones que no dependan de la memoria por su expresión más sencilla. La resolución debe ser exclusivamente *simbólica*.
- demostrar la siguiente propiedad:
  i. **`evalBExp . cfBExp = evalBExp`**

Ejercicio 3) Dada la siguiente representación de un lenguaje imperativo simple:

```
data Programa = Prog Bloque
type Bloque = [Comando]
data Comando = Assign Nombre NExp
             | If BExp Bloque Bloque
             | While BExp Bloque
```

- implementar las siguientes funciones:
  i. **`evalProg :: Programa -> Memoria -> Memoria`**, que describe la memoria resultante de evaluar el programa dado a partir de la memoria dada.
  ii. **`evalBlq :: Bloque -> Memoria -> Memoria`**, que describe la memoria resultante de evaluar el bloque dado a partir de la memoria dada.
  iii. **`evalCom :: Comando -> Memoria -> Memoria`**, que describe la memoria resultante de evaluar el comando dado a partir de la memoria dada.
  iv. **`optimizeCF :: Programa -> Programa`**, que describe un programa con el mismo significado que el dado, pero aplicando

constant folding sobre las expresiones y descartando los comandos que no serán ejecutados.

- demostrar las siguientes propiedades:

  i. para todo **c.** para todo **cs.**
  `evalBloque (c:cs) = evalBloque cs . evalComando c`

  ii. para todo cs_1. para todo cs_2.
  `evalBloque (cs1++cs2) = evalBloque cs2 . evalBloque cs1`

  iii. para todo **be.** para todo cs_1. para todo cs_2.
  `evalComando (If be cs1 cs2) = evalComando (If (Not be) cs2 cs1)`

  iv. para todo **x.** para todo ne_1. para todo ne_2.
  si (para todo **v.** para todo **mem.**
  `evalNExp ne2 (recordar x v mem) = evalNExp ne2 mem`)
  entonces `evalBloque [Assign x ne1, Assign x ne2] = evalComando (Assign x ne2)`
  AYUDA: el antecedente solamente establece que **x** no aparece en ne_2 y por lo tanto no influye en el resultado

- demostrar la siguiente propiedad:
  `evalProg . optimizeCf = evalProg`

Ejercicio 4) Dada la siguiente representación del lenguaje Rowbstones:

```
data DirR = Oeste | Este
data ExpR a = Lit a
            | PuedeMover Dir
            | NroBolitas
            | HayBolitas
            | UnOp  UOp (ExpR a)
            | BinOp BOp (ExpR a) (ExpR a)

data UOp = No | Siguiente | Previo
data BOp = YTambien | OBien | Mas | Por

type ProgramaR = ComandoR
data ComandoR = Mover DirR
              | Poner
              | Sacar
              | NoOp
              | Repetir (ExpR Int) ComandoR
              | Mientras (ExpR Bool) ComandoR
              | Secuencia ComandoR ComandoR
```

y el TAD `TableroR` cuya interfaz es la siguiente:

- **`tableroInicial :: Int -> TableroR`**, que describe un tablero inicial.
- **`mover :: DirR -> TableroR -> TableroR`**, que describe el tablero a partir del tablero dado donde el cabezal se movió hacia la dirección dada. Esta operación es parcial, pues debe poderse mover en la dirección dada (ver **`puedeMover`**).
- **`poner :: TableroR -> TableroR`**, que describe el tablero donde se agregó una bolita de la celda actual del tablero dado.
- **`sacar :: TableroR -> TableroR`**, que describe el tablero donde se sacó una bolita de la celda actual del tablero dado. Esta operación es parcial, pues debe haber bolitas en la celda actual (ver **`hayBolitas`**).
- **`nroBolitas :: TableroR -> Int`**, que describe la cantidad de bolitas en la celda actual del tablero dado.
- **`hayBolitas :: TableroR -> Bool`**, que indica si hay bolitas en la celda actual del tablero dado.
- **`puedeMover :: DirR -> TableroR -> Bool`**, que indica si el cabezal se puede mover hacia la dirección dada en el tablero dado.
- **`boom :: String -> TableroR -> a`**, que describe el resultado de realizar una operación de consulta inválida sobre un tablero.

a. implementar las siguientes funciones:

   i. **`evalExpRInt :: ExpR Int -> TableroR -> Int`**, que describe el número que resulta de evaluar la expresión dada en el tablero dado.
   NOTA: en caso de que la expresión no pueda computar un número, debe ser considerada una operación inválida.

   ii. **`evalExpRBool :: ExpR Bool -> TableroR -> Bool`**, que describe el booleano que resulta de evaluar la expresión dada en el tablero dado.
   NOTA: en caso de que la expresión no pueda computar un booleano, debe ser considerada una operación inválida.

   iii. **`expRTieneTipoInt :: ExpR Int -> Bool`**, que indica si la expresión dada no falla cuando se ejecuta **`evalExpRInt`**.

   iv. **`expRTieneTipoBool :: ExpR Bool -> Bool`**, que indica si la expresión dada no falla cuando se ejecuta **`evalExpRBool`**.

   v. **`evalR :: ComandoR -> TableroR -> TableroR`**, que describe el tablero resultante de evaluar el comando dado en el tablero dado.

   vi. **`cantSacar :: ComandoR -> Int`**, que describe la cantidad de constructores Sacar que hay en el comando dado.

   vii. **`seqN :: Int -> ComandoR -> ComandoR`**, que describe la secuencia de comandos que reitera el comando dado la cantidad de veces dada.

   viii. **`repeat2Seq :: ComandoR -> ComandoR`**, que describe el comando resultante de reemplazar, en el comando dado, todas las apariciones del constructor Repetir aplicado a un literal por la

secuencia de comandos que reitera el comando correspondiente la cantidad de veces dada por el literal.

b. dadas las siguientes definiciones

```
sequence :: [(a -> a)] -> a -> a
sequence []     x = x
sequence (f:fs) x = f (sequence fs x)

evalMany :: Int -> ProgramaR -> TableroR -> TableroR
evalMany 0 p t = t
evalMany n p t = evalR p (evalMany (n-1) p t)

evalList :: ProgramaR -> [TableroR] -> [TableroR]
evalList p []     = []
evalList p (t:ts) = evalR p t : evalList p ts
```

demostrar las siguientes propiedades:

i. para todo **p**. para todo **ts1**. para todo **ts2**.
```
evalList p (ts1 ++ ts2)
          = evalList p ts1 ++ evalList p ts2
```
ii. para todo **n**.
```
eval (Repetir (Lit n) Poner) = eval (seqN n Poner)
```
iii. para todo **n**.
```
eval (Repetir (Lit n) Poner)
          = (sequence . replicate n) poner
```
iv. para todo **n**. para todo **p**. para todo **t**.
```
evalMany n p t = evalR (Repetir n p) t
```
v. para todo **n**. para todo **p**
```
evalMany n p = many n (evalR p)
```
vi. `evalR . repeat2Seq = evalR`

vii. para todo **n**.
```
cantSacar (repeat2Seq (Repeat (Lit n) Sacar))
  = length (replicate n Sacar)
```