Logikai és funkcionális programozás – funkcionális programozás modul

A laborfeladatok írásához a Clean nyelvet használtuk – a program ingyenesen letölthető a ftp://ftp.cs.kun.nl/pub/Clean illetve a http://www.cs.kun.nl/~clean link-ekről.

Példaprogramok:

1. egy szám faktoriálisának a kiszámítása:

```
Start = fakt 6

// a függvény típusának a definiciója
fakt :: Int -> Int
fakt 0 = 1
fakt n | n>0 = n * fakt (n-1)
```

2. egy lista legnagyobb elemének a meghatározása:

```
Start = maxEl [10,3,2,4]

// egy lista a bemenő argumentum
maxEl :: [Int] -> Int
maxEl [x] = x
maxEl [hx:tx]
  | hx > mT = hx
  | otherwise = mT
 where mT = maxEl tx
```

3. Euler algoritmusa egy pozitív szám négyzetgyökének a meghatározására:

   az algoritmus az $x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right)$ iterációt implementálja.

```
Start = root 15.0

root:: Real -> Real
root x = until goodEnough improve 1.0
  where improve y = 0.5*(y + x/y)
        goodEnough y = y*y ~=~ x

// operátordefiníció
(~=~) infix 5 :: Real Real -> Bool
(~=~) a b = a-b < h && b - a < h
   where h = 0.00000001
```

4. Beszúrásos rendezés:

```
Start = IIsortF [2, 1, 5, 7, 4, 3]
```

```
// a második argumentum a hívásnál lesz beírva
IIsortF = foldr Insert []

IIsort:: [a] -> [a] | Ord a
IIsort x = foldr Insert [] x

Insert:: a [a] -> [a] | Ord a
Insert e [] = [e]
Insert e [f:v]
  | e<f = [e,f:v]
  | otherwise = [f] ++ (Insert e v)
```

5. Generáljuk az összes prímszámot:

```
// lambda-függvény
Start = filter (\ x = (divisors x) == [1,x] ) [1..]

// az argumentumának az osztóit adja vissza
divisors :: Int -> [Int]
divisors x = filter (\ v = (x rem v) ==0) [1..x]
```

6. Generáljuk a pitagórászi számhármasokat n-ig:

```
Start = pitTrio 500

pitTrio :: Int -> [(Int,Int,Int)]
pitTrio n = map thirdNum (pit2 n)
  where thirdNum (x,y) = (x,y,rtInt(x*x+y*y))

// PIT-I számpárosokat térít vissza
pit2 :: Int -> [(Int,Int)]
pit2 n = [ (x,y) \\ x<-[1..n], y<-[1..n] | x<y &&
isSq(x*x+y*y)]

// egészeken definiált négyzetgyök
rtInt :: Int -> Int
rtInt x = toInt (sqrt (toReal x) )

// teljes négyzet tesztje
isSq :: Int -> Bool
isSq x = (x == y*y)
  where y = rtInt x
```

7. Prímszámok generálása szűréssel (a lusta kiértékelés – lazy evaluation – miatt fut le a program):

```
Start = sieve [2..]

// _KIszűri a lista első elemének többszöröseit
```

laborfeladat ....

meg mindig, de rovidke ... a komment sok

Laborfeladatok

A laborfeladatoknál felhasználható a http://www.cs.ubbcluj.ro/~csatol lapon elérhető dokumentáció a funkcionális programozással kapcsolatban.

Követelmények:
1. feladatok megoldásánál írjunk komment-eket;
2. definiáljuk a típusát minden függvénynek;
3. próbáljuk a futási időt optimálni – listákat egy alkalommal járjuk csak be, ne generáljunk fölöslegesen változókat;

Feladatok

a. Írjunk programot, mely egy számot egy pozitív egész hatványra emel. ....(1pt)

b. Írjunk programot, mely visszaadja egy lista két legnagyobb elemét.............(1pt)

c. Írjunk egy programot a komplex számokkal való műveletek végzésére.
1. A programban írjuk felül az alapműveleteket (+, -, *, /), a negációt, az egyenlőségjelet (==) ....(2pt)
2. Írjuk felül a **zero**, **one**, **abs**, **toString** függvényt (használjuk az ***instance*** parancsot)..(2pt)
3. Definiáljunk egy függvényt – lásd **mkQ** az előadásból – amely egy komplex számot hoz létre (1) a komplex síkból, (2) polár koordinátákból...................(2pt)
4. írjunk kódot a fenti műveletek tesztelésére (használjuk a TUPLE kiírást). A program kimenete párok, ahol megjelennek a műveletek argumentumai, illetve az eredmény (lásd másodfokú egyenlet, ea.pdf). ....(2pt)

d. A – komplex modul használatával – írjunk programot az utazóügynök feladatra. A program bemenete egy lista komplex számokkal, mely tartalmazza a városok síkbeli koordinátáit. A kimenet egy lista, melyben a városokat az adott sorrendben bejárva a legkisebb lesz a megtett út..(5pt)

Összesen 15 pont.

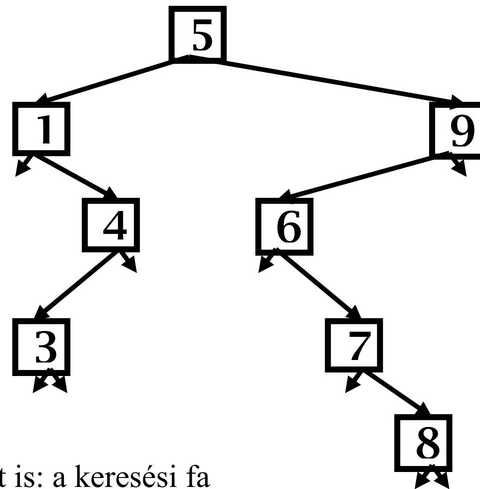
Opcionális feladat – műveletek bináris fákkal

Bináris fákat keresések gyorsítására használják. Használjuk a bináris fa algebrai típusdefiníciót

```
::Tree a = Node a (Tree a) (Tree a)
         |  Leaf
```

Írjunk egy függvényt, mely egy egészekből álló lista elemeiből egy bináris keresési fát épít. Egy listából fát építeni úgy lehet, hogy iteratívan beszúrunk egy már létező bináris – kezdetben üres – fába elemeket. A függvény típusa

```
treeFromList :: [Int] -> Tree Int
```



A keresési fa bal részében a gyökér-elemnél kisebb elemek vannak, a jobb oldalon az annál nagyobb elemek – két egyforma elemet nem tárolunk a fában. Egy példa erre a transzformációra a mellékelt ábra, amikor a

`[5, 1, 4, 3, 9, 6, 7, 9, 8]`

listát alakítjuk keresési fává. Az ábrán a nyilak jelzik az al-fákat. A kis nyilak jelzik azt, hogy a lista véget ért (`Leaf` típusú konstruktor). Az átalakítás folyamán a leveleket egészítettük ki új elemekkel a listából. Írjuk meg az inverz műveletet is: a keresési fa elemeinek listába fűzését.
....(4pt)

Definiáljuk a `toString` függvényt, amely megjeleníti a bináris fát. Egy lehetséges mód az alábbi:

```
(5 (1 _ ; (4 (3 _;_);_));(9 (6 _;(7 _;(8 _;_)));_))
```

....(1pt)

Amennyiben a keresési fa nem kiegyensúlyozott, a keresés nem hatékony. A kiegyensúlyozottság az al-fák mélységének (**`depth`** függvény az előadás során) függvénye: ha a fa bal és jobb ágának a különbsége nagyobb, mint kettő, akkor a fa nem kiegyensúlyozott. Írjunk egy – rekurzív – függvényt, amely a fába úgy tesz be elemeket, hogy a fa kiegyensúlyozott legyen. Egy mód erre az, hogy, amennyiben szükséges, a beszúrandó elemet a fa gyökerében tároljuk. (dokumentáljuk a megoldást)
....(5pt)

Írjunk egy függvényt, mely egy kiegyensúlyozatlan fából egy kiegyensúlyozott fát hoz létre. A kritérium itt is a keresési fa alegységei mélységének a különbsége – mely kettőnél kisebb kell, hogy legyen. (dokumentáljuk a megoldást)
....(5pt)

Összesen 15 pont.

A laborfeladatok megoldásai:

a. Írjunk programot, mely egy számot egy pozitív egész hatványra emel. ....(1pt)

```
poW :: Real Int -> Real
poW x 1 = x
poW x y = x*poW( x y-1)
```

b. Írjunk programot, mely visszaadja egy lista két legnagyobb elemét. ....(1pt)

```
twoLargest :: [Int] -> [Int]
twoLargest [x,y]
  | x>y = [x,y]
  = [y,x]

twoLargest [hX:rX]
  | m1R < hX = [hX,m1R]
  | m2R < hX = [m1R,hX]
  = [m1R,m2R]
  where [m1R:m2List]  = twoLargest rX
        m2R = hd m2List
```

c. Írjunk egy programot a komplex számokkal való műveletek végzésére.

```
// komplex típus
::Z = {v::Real,i:: Real}
```

1. A programban írjuk felül az alapműveleteket (+, -, *, /), a negációt, az egyenlőségjelet (==) ....(2pt)

```
instance + Z
    where (+) a b = {v=(a.v+b.v),i=(a.i+b.i)}
instance - Z
    where (-) a b = {v=(a.v-b.v),i=(a.i-b.i)}
instance * Z
    where (*) a b = {v=(a.v*b.v-
a.i*b.i),i=(a.v*b.i+a.i*b.v)}
instance / Z
    where (/) a b = {v=(a.v*b.v+a.i*b.i)/mb,
                     i=(a.i*b.v-a.v*b.i)/mb}
            where mb = b.v*b.v + b.i*b.i
instance == Z
    where (==) a b = ((a.v==b.v)&&(a.i==b.i))
```

2. Írjuk felül a **zero**, **one**, **abs**, **toString** függvényt (használjuk az ***instance*** parancsot)...(2pt)

```
instance zero Z
    where zero = {v=0.0,i=0.0}
instance one Z
    where one = {v=1.0,i=0.0}
instance toString Z
    where toString z = (toString z.v) +++ "+" +++ (toString
z.i) +++ "i\n"
instance abs Z
    where abs z = {v=(sqrt(z.v*z.v + z.i*z.i)),i=0.0}
```

3. Definiáljunk egy függvényt – lásd **mkQ** az előadásból – amely egy komplex számot hoz létre (1) a komplex síkból, (2) polár koordinátákból..................(2pt)

```
mkZ :: (Real,Real) -> Z
mkZ (a,b)  = {v=a,i=b}
mkPZ:: (Real,Real) -> Z
mkPZ (r,theta) = { v=r*cos(theta), i=r*sin(theta)}
```

4. írjunk kódot a fenti műveletek tesztelésére (használjuk a TUPLE kiírást). A program kimenete párok, ahol megjelennek a műveletek argumentumai, illetve az eredmény (lásd másodfokú egyenlet, ea.pdf). ....(2pt)

```
Start = (toString a) +++ " es " +++ (toString b) +++ " -
RE\na+b: "
        +++ toString(a + b) +++ " \na-b: "
        +++ toString(a - b) +++ " \na*b: "
        +++ toString(a * b) +++ " \na/b: "
        +++ toString(a / b)
    where a = mkZ(1.0,3.0)
        b = mkZ(-3.0,2.0);
```

d. A – komplex modul használatával – írjunk programot az utazóügynök feladatra. A program bemenete egy lista komplex számokkal, mely tartalmazza a városok síkbeli koordinátáit. A kimenet egy lista, melyben a városokat az adott sorrendben bejárva a legkisebb lesz a megtett út..(5pt)

```
// permutáció generáló függvényt (a tetszőleges típus)
pperm :: [a] ->[[a]]
pperm []  = [[]];
pperm [a] = [[a]];
pperm [hd:tt] = [ iat i hd list \\ i <- [0..lTail], list <-
pPrev ]
      where lTail = length tt
              pPrev = pperm tt

// függvény, mely egy lista adott helyére beszúr egy elemet
iat:: Int a [a] -> [a]
iat 0 x l     = [x:l]
iat n x []    = [x]
iat p x [h:t] = [h:(iat (p-1) x t)];

// segédfüggvény mely a valós abszolút értéket ad
abs_r::Z -> Real
abs_r z = b.v
    where b = abs(z)

// a lista „hossza”
dList::[Z] -> Real
dList []      = abort "hiba!"
dList [x]     = 0.0
dList [h1:h2:t] = abs_r(h1-h2) + (dList [h2:t])

// függvény, mely kiválasztja a legrövidebb utat.
```

```
mList:: [[Z]] -> [Z]
mList [[]]  = abort "hiba!"
mList [x]   = x
mList [h:t]
     | hL < tL = h
     = mT
   where hL = dList h
         mT = mList t
         tL = dList mT

// a program futtatása
Start = mList (pperm lV)
    where lV = [mkZ(0.0,4.0),zero,one,mkZ(2.0,4.0),
                mkZ(2.0,3.0),mkZ(4.0,0.0)]
```