Introduction to Functional Programming

Revision

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
zsv@elte.hu

Overview

Basic types, function definitions, recursion

```
isum :: Int → Int
isum x
| x == 0 = 0
= (x rem 10) + isum (x / 10)

Start = isum 123 // 6

power :: Int Int → Int
power x n
| n==0 = 1
| n>0 = x * power x (n-1)

Start = power 2 5 // 32
```

Composition, omitting values

```
twiceof :: (a → a) a → a
twiceof f x = f (f x)

Twice :: (t→t) → (t→t)
Twice f = f o f

// omitting values
f :: Int Int → Int
f _ x = x

Start = f 4 5 // 5
```

Nr to list, double recursion

```
p :: Int → [Int]
p x = digits x []
digits :: Int [Int] → [Int]
digits 0 l = l
digits x l = digits (x/10) [x rem 10 : l]

pali :: Int → Bool
pali x = y == reverse y
   where y = p x
Start = pali 12321 // True

fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
Start = fib 5 // 8
```

Lists, patterns, recursion

```
hd, tl, drop, take, ++, flatten, reverse,
length, last, init, isMember, removeDup
rewriteFlatten :: [[Int]] → [Int]
rewriteFlatten [] = []
rewriteFlatten [x : xs] = x ++ rewriteFlatten xs

triplesum :: [Int] → Int
triplesum [x, y, z] = x + y + z

sim :: [Int] → Bool
sim x = x == reverse x

not_five :: [Int] → [Int]
not_five  []    =  []
not_five  [5:t]  = not_five t
not_five  [h:t]  = [h : not_five t]
```

List recursion

```
tails :: [[Int]] → [[Int]]
tails [] = []
tails [x : xs] = [ tl x : tails xs]
Start = tails [[1, 2, 3], [3, 4], [5, 7, 8, 9]]

tailsd :: [[Int]] → [[Int]]
tailsd [] = []
tailsd [x : xs] = [ (drop 1 x) : tailsd xs]
Start = tailsd [[1, 2, 3], [3, 4], [5, 7, 8, 9]]

tails x = map tl x
```

Higher order functions

```
ins0 :: [[Int]] → [[Int]]
ins0 [] = []
ins0 [x:xs] = [ 0, x : ins0 xs]
// or
ins0 [x:xs] = [ [0] ++ x : ins0 xs]

f lists = map (λx = [0] ++ x) lists

f lists = map ((++)[0]) lists
```

```
// removes x from a list
remove :: Int [Int] → [Int]
remove x [] = []
remove x [y:ys]
| x == y      = remove x ys
| otherwise = [y : remove x ys]
Start = remove 3 [1,2,1,1,2,3,2,1,3] // [1,2,1,1,2,2,1]

// removeDuplicates l returns the list l with all duplicates removed
removeDuplicates :: [Int] → [Int]
removeDuplicates []       = []
removeDuplicates [x:xs] = [x : removeDuplicates (remove x xs)]
Start = removeDuplicates [1,2,1,2,3,1,2,4,2,3] // [1,2,3,4]
```

foldr, map

```
// add the numbers of 1..N (N positive) using foldr
addn :: Int → Int
addn n
| n < 0 = abort "for n negative there is no sum defined"
= foldr (+) 0 [1..n]

// compute n! factorial using foldr
f :: Int → Int
f n = foldr (*) 1 [1..n]

// compute 1*1 + 2*2 + ... + n*n for n positive using map and foldr
sumsqr :: Int → Int
sumsqr n = foldr (+) 0 (map (λx = x*x)  [1..n])
```

foldr, map, filter, takeWhile

```
// rewrite flatten using foldr
flat :: [[Int]] → [Int]
flat x = foldr (++) [] x

sqrs_lambda :: [Int] → [Int]
sqrs_lambda y = map (λx = x*x) y

// compute the double of the positive elements of a list
double x = 2*x
f2 :: [Int] → [Int]
f2 list = map double (filter ((<)0) list)

// numbers of a list up to the first 0 encountered then divide by 2
div2 x = x/2
f :: [Int] → [Int]
f list = map div2 (takeWhile ((≠)0) list)
```

higher order functions

```
// Replicate n>0 times the element of a list
replicate :: Int Int → [Int]
replicate 0 x = []
replicate n x = [x : replicate (n-1) x]

f :: Int [Int] → [[Int]]
f n list = map (replicate n) list

//compute the sum of the sublist using foldr
f :: [[Int]] → [Int]
f lists = map (foldr (+) 0) lists

g l = filter isEven l

l = iterate (λ x = x/10) 54321 // [54321,5432,543,54,5,0,0...]
```

list comprehensions, tuples

```
zip, search, fst, snd, unzip
// generate the list [[1],[2,2],[3,3,3],[4,4,4,4],...,[10,..,10]]
l :: [[Int]]
l = [[y \\ x←[1..y]] \\ y←[1..10]]

// 5 Pythagoras numbers
l = take 5 [(a,b,c)\\c←[1..],b←[1..c],a←[1..b] |a*a+b*b==c*c]

// parallel processing
l =[(x,y)\\ x←[1..] & y←['a'..'z']]

divisors nr = [x \\ x ← [1..nr] | nr rem x == 0]
```

tuples

```
// sum of the list of tuples [(1,1), (2,2), (3,3)] -> (6,6)
sumtup l = (sum (fst x), sum (snd x))
where x = unzip l

triplesum l = [(fst a, snd a, fst a + snd a) \\ a ← l]

triplesum1 l = [(x,y,x+y) \\ x ← (fst a) & y ← (snd a)]
where a = unzip l

triplesum2 l = [(x,y,x+y) \\ x ← (map fst l) & y ← (map snd l)]

tri l = [(x,y,z) \\ x ← fst3 l & y ← snd3 l & z ← thd3 l]

tri1 (a,b,c) = [(x,y,z) \\ x ← a & y ← b & z ← c]
```

sieve, quick sort

```
sieve :: [Int] → [Int]
sieve [p:xs] = [p: sieve [ i \\ i ← xs | i rem p ≠ 0]]
Start = take 100 (sieve [2..])

qsort :: [a] → [a] | Ord a
qsort [] = []
qsort [c : xs] = qsort [x \\ x ← xs | x <  c] ++ [c] ++
                 qsort [x \\ x ← xs | x >= c]
```

sorting by insertion

```
// inserting in already sorted list
Insert :: a [a] → [a] | Ord a
Insert e [] = [e]
Insert e [x : xs]
| e ≤ x = [e , x : xs]
| otherwise = [x : Insert e xs]
Start = Insert 5 [2, 4 .. 10] // [2,4,5,6,8,10]

mysort :: [a] → [a] | Ord a
mysort [] = []
mysort [a:x] = Insert a (mysort x)
Start = mysort [3,1,4,2,0] // [0,1,2,3,4]

mergesort :: [a] [a] → [a] | Ord a
```

records, arrays

```
:: Point = {   x       ::  Real
             , y       ::  Real
             , visible ::  Bool
             }

IsVisible :: Point → Bool
IsVisible {visible = True} = True
IsVisible _                = False

hide p = { p & visible = False }
xcoordinate p = p.x

MyArray :: {Int}
MyArray = {1,3,5,7,9}
Start = MyArray.[2]

MapArrays f a = {f e \\ e <-: a}
```

trees, search trees

```
:: Tree a = Node a (Tree a) (Tree a)
          | Leaf
nrT :: (Tree a) → Int
nrT Leaf = 0
nrT (Node x le ri) = 1 + nrT le + nrT ri

traversing a tree inorder, preorder, postorder

treesort :: ([a]→ [a]) | Eq, Ord a
treesort = collect o listtoTree
```

Algebraic types

```
// Triple branches
:: Tree4 a = Node4 a (Tree4 a) (Tree4 a) (Tree4 a)
           | Leaf4

// Rose-tree - tree with variable, multiple branches
// No leaf constructor, node with no branches
:: Tree5 a = Node5 a [Tree5 a]

// Every node has one branch = list
:: Tree6 a = Node6 a (Tree6 a)
            | Leaf6
```

Map, foldr on trees

```
:: BTree a = Bin (BTree a) (BTree a)
           | Tip a

mapbtree :: (a → b) (BTree a) → BTree b
mapbtree f (Tip x) = Tip (f x)
mapbtree f (Bin t1 t2) = Bin (mapbtree f t1) (mapbtree f t2)

foldbtree :: (a a → a) (BTree a) → a
foldbtree f (Tip x) = x
foldbtree f (Bin t1 t2) = f (foldbtree f t1) (foldbtree f t2)

aBTree = Bin (Bin (Bin (Tip 1) (Tip 1))
                  (Bin (Tip 3) (Tip 3))) (Tip 5)

Start = mapbtree inc aBTree
Start = foldbtree (+) aBTree // 13
```

ADT

```
:: Bag a
newB    ::     (Bag a)                      // empty bag
isempty ::     (Bag a) → Bool
insertB :: a   (Bag a) → Bag a  | Eq a   // insert an element
removeB :: a   (Bag a) → Bag a  | Eq a   // remove an element
```

Bag

```
:: Bag a :== [(Int,a)]

insertB :: a (Bag a) → Bag a | Eq a
insertB e [] = [(1,e)]
insertB e [(m,x):t]
| e == x = [(m+1,x):t]
= [(m,x)] ++ insertB e t

removeB :: a (Bag a) → Bag a | Eq a
removeB e [] = []
removeB e [(m,x):t]
| e == x && (m-1) == 0 = t
| e == x = [(m-1,x):t]
= [(m,x)] ++ removeB e t
```

instances, classes

```
instance +    (a,b)   | + a & + b
instance -    (a,b)   | - a & - b
instance *    (a,b)   | * a & * b
instance /    (a,b)   | / a & / b
instance zero (a,b)   | zero a & zero b
instance one  (a,b)   | one a & one b
instance ¬    (a,b)   | ¬ a & ¬ b

class Delta a | *,-,fromInt a
delta1 :: a a a → a | Delta a
```

instances of predefined classes

```
instance + String
where
    (+) s1 s2 = s1 +++ s2
Start = "Hello" + " world!" // "Hello world!"

instance + (a,b) | + a & + b
where
    (+) (x1,y1) (x2,y2) = (x1+x2,y1+y2)

Start = (1,2) + (3,4) // (4,6)
```

classes

```
class PlusMinx a
 where
       (+¬)  infixl 6   :: !a   !a       →        a
       (-¬)  infixl 6   :: !a   !a       →        a
       zerox            :: a

instance PlusMinx Char
 where
       (+¬) :: !Char !Char → Char
       (+¬) x y =  toChar (toInt(x) + toInt(y))
       (-¬) x y =  toChar (toInt(x) - toInt(y))
       zerox = toChar 0

Start = 'a' +¬ 'e'
```

Map

```
module Map
import StdEnv

// The (Maybe a) type represents a collection of at most one element
:: Maybe a =  Just a
            | Nothing

// Binary trees
::  Tree a = Leaf | Node a (Tree a) (Tree a)

// Single tree - roses
:: Tree1 a = Node1 a [Tree1 a]
```

Map

// the type constructor of class Map

```
class Map t :: (a → b) (t a) → t b

instance Map []
where Map f xs = map1 f xs
      map1 :: (a → b) [a] → [b]
      map1 f [] = []
      map1 f [x:xs] = [f x : map1 f xs]

instance Map Maybe
where Map f mb = mapMaybe f mb
      mapMaybe :: (a → b) (Maybe a) → Maybe b
      mapMaybe f Nothing  = Nothing
      mapMaybe f (Just x) = Just (f x)
```

Map

```
instance Map Tree
where Map f tr = mapTree  f tr
mapTree        :: (a → b) (Tree a) → Tree b
mapTree f Leaf        = Leaf
mapTree f (Node x le ri) = Node (f x) (mapTree f le) (mapTree f ri)

instance Map Tree1
where Map f tr = mapTree1  f tr
mapTree1                     :: (a → b) (Tree1 a) → Tree1 b
mapTree1 f (Node1 elem ls) = Node1 (f elem) (map (mapTree1 f) ls)

instance Map ((,) a)
where
   Map :: (a → b) (c,a) → (c,b)
   Map f (x,y) = (x,f y)
```

Map

```
t1 :: Tree Int // binary tree
t1 = Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
Start = Map inc t1

a1 :: Tree1 Int // rose
a1 = Node1 1 [Node1 2 [Node1 3 [], Node1 4 [], Node1 5
     [Node1 6 []]]]
Start = Map inc a1
Start = Map inc [1..10]
Start :: Maybe Int
Start = Map inc (Just 4)
Start = Map inc Nothing
Start = Map inc (1.5, 2)
```