Introduction to Functional Programming

Zsók Viktória

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

Overview

1. Tuples
2. List comprehensions
3. Infinite lists

Warm-up exercises 2

Write a function or an expression for the following:

1. compute 5! factorial using `foldr` => 120
2. rewrite `flatten` using `foldr` (for the following list [[1,2], [3, 4, 5], [6, 7]] => [1,2,3,4,5,6,7])
3. using `map` and `foldr` compute how many elements are altogether in the following list [[1,2], [3, 4, 5], [6, 7]] => 7
4. using `map` extract only the first elements of the sublists in [[1,2], [3, 4, 5], [6, 7]] => [1,3,6]

Solutions 2

Write a function or an expression for the following:

1. compute 5! factorial using `foldr` => 120
2. rewrite `flatten` using `foldr` (for the following list [[1,2], [3, 4, 5], [6, 7]] => [1,2,3,4,5,6,7])
3. using `map` and `foldr` compute how many elements are altogether in the following list [[1,2], [3, 4, 5], [6, 7]] => 7
4. using `map` extract only the first elements of the sublists in [[1,2], [3, 4, 5], [6, 7]] => [1,3,6]

```
1. Start = foldr (*) 1 [1..5]
2. Start = foldr (++) [] [[1,2], [3, 4, 5], [6, 7]]
3. Start = foldr (+) 0 (map length [[1,2], [3, 4, 5], [6, 7]])
4  Start = map hd [[1,2], [3, 4, 5], [6, 7]]
```

Tuples

```
(1,'f')              :: (Int,Char)
("world",True,2)     :: (String,Bool,Int)
([1,2],sqrt)         :: ([Int],Real→Real)
(1,(2,3))            :: (Int,(Int,Int))
// any number 2-tuples pair, 3-tuples,
// no 1-tuple (8) is just an integer, no need for ()

fst :: (a,b) → a
fst (x,y) = x
Start = fst (10, "world") // 10

snd :: (a,b) → b
snd (x,y) = y
Start = snd (1,(2,3))     // (2,3)

f :: (Int, Char) → Int
f (n, x) = n + toInt x
Start = f (1,'a') // 98
```

Tuples

```
splitAt :: Int [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)

Start = splitAt 3 ['hello'] // (['h','e','l'],['l','o'])

search :: [(a, b)] a -> b | == a
search [(x, y):ts] s
| x == s = y
| otherwise = search ts s

Start = search [(1,1), (2,4), (3,9)] 3 // 9
```

Zipping

```
zip2 :: [a] [b] → [(a, b)]
zip2 [] [] = []
zip2 [] ys = []
zip2 xs [] = []
zip2 [x : xs] [y : ys] = [(x, y) : zip2 xs ys]

Start = zip2 [1,2,3] ['a'..'c'] // [(1,'a'),(2,'b'),(3,'c')]

zip :: ([a], [b]) → [(a, b)]
zip (x, y) = zip2 x y

Start = zip ([1,2,3], ['a'..'c']) // [(1,'a'),(2,'b'),(3,'c')]
```

List comprehensions

```
Start :: [Int]
Start = [x * x \\ x ← [1..10]] // [1,4,9,16,25,36,49,64,81,100]

// expressions before double backslash
// generators after double backslash
// i.e. expressions of form x <- xs x ranges over values of xs
// for each value value the expression is computed

Start = map (λx = x * x) [1..10] // [1,4,9,16,25,36,49,64,81,100]

// constraints after generators

Start :: [Int]
Start = [x * x \\ x ← [1..10] | x rem 2 == 0]   // [4,16,36,64,100]
```

List comprehensions

```
// nested combination of generators
// coma combinator - generates every possible combination of the
// corresponding variables, last variable changes faster
// for each x value y traverses the given list

Start :: [(Int,Int)]
Start = [(x,y) \\ x <- [1..2], y <- [4..6]]
        // [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]

// parallel combinator of generators is &

Start = [(x,y) \\ x <- [1..2] & y <- [4..6]]
        // [(1,4),(2,5)]

// multiple generators with constraints

Start = [(x,y) \\ x <- [1..5], y <- [1..x] | isEven x]
        // [(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

List comprehensions - equivalences

```
mapc :: (a → b) [a] → [b]
mapc f y = [ f x \\ x ← y ]

filterc :: (a → Bool) [a] → [a]
filterc p y = [ x \\ x ← y | p x ]

zipc :: [a] [b] → [(a, b)]
zipc x y = [(a, b) \\ a ← x & b ← y]

Start = zipc [1,2,3] [10, 20, 30] // [(1,10),(2,20),(3,30)]

// functions like sum, reverse, isMember, take
// are hard to write using list comprehensions
```

Generating infinite list

```
// generating infinite list
Start = [2..]  // [2,3,4,5,...]
Start = [1,3..]  // [1,3,5,7,...]

fromn :: Int → [Int]
fromn n = [n : fromn (n+1)]

Start = fromn 8  // [8,9,10,...]

// intermediate result is infinite
Start = map ((^)3) [1..]

// final result is finite
Start = takeWhile ((>) 1000) (map ((^)3) [1..])
// [3,9,27,81,243,729]
```

Infinite lists - repeat

```
// generating infinite list with repeat from StdEnv
repeat :: a → [a]
repeat x = list
  where list = [x:list]

Start = repeat 5 // [5,5,5,...]

repeatn ::  Int a → [a]
repeatn n x = take n (repeat x)

Start = repeatn 5 8 // [8,8,8,8,8]
```

Infinite lists - iterate

```
// generating infinite list with iterate from StdEnv
iterate :: (a -> a) a -> [a]
iterate f x = [x: iterate f (f x)]

Start = iterate inc 5 // [5,6,7,8,9,...]

Start = iterate ((+) 1) 5 // [5,6,7,8,9,...]

Start = iterate ((*) 2) 1 // [1,2,4,8,16,...]

Start = iterate (\ x= x/10) 54321 // [54321,5432,543,54,5,0,0...]

Start = reverse ( map (\x=x rem 10) (takeWhile ((<)0)
                                     (iterate (\ x= x/10) 54321)))
        // [5,4,3,2,1]
```

Warm-up exercises 3

Write a function for the following:

1. fibonnacci n
2. count the occurrences of a number in a list
3. write a list comprehension for the squares of the elements of a list

Solutions 3

```
fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

Start = fib 5  // 8

fib2 :: Int → Int
fib2 n = fibAux n 1 1

fibAux 0 a b         = a
fibAux i a b
| i > 0 = fibAux (i-1) b (a+b)

Start = fib2 5  // 8
```

Solutions 3

```
CountOccurrences :: a [a] → Int | == a
CountOccurrences a [x : xs] = f a [x : xs] 0
where
      f a [] i = i
      f a [x : xs] i
        | a == x = f a xs i+1
                 = f a xs i

Start = CountOccurrences 2 [2, 3, 4, 2, 2, 4, 2, 1] // 4

Start = [x*x \\ x ← [1..5]] // [2, 4, 6, 8, 10]
```