Introduction to Functional Programming

Sorting

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

Overview

Quick sort

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

Start = qsort [2,1,5,3,6,9,0,1] // [0,1,1,2,3,5,6,9]
```

Sorting lists

```
// the sort function is in StdEnv
sort :: [a] → [a] | Ord a
Start = sort [3,1,4,2,0] // [0,1,2,3,4]

// inserting in an already sorted list, insert is in StdEnv
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]

Insert 3 (Insert 1 (Insert 4 (Insert 2 (Insert 0 [] ))))
```

Merge

```
// merge is in StdEnv
merge :: [a] [a] → [a] | Ord a
merge [] ys = ys
merge xs [] = xs
merge [x : xs] [y : ys]
| x ≤ y = [x : merge xs [y : ys]]
| otherwise = [y : merge [x : xs] ys]

Start = merge [2,5,7] [1,5,6,8] // [1,2,5,5,6,7,8]
Start = merge [] [1,2,3] // [1,2,3]
Start = merge [1,2,10] [] // [1,2,10]
Start = merge [2,1] [4,1] // [2,1,4,1]
Start = merge [1,2] [1,4] // [1,1,2,4]
```

Merge - renamed pattern

```
merge :: [a] [a] → [a] | Ord a
merge [] ys = ys
merge xs [] = xs
merge p=:[x : xs] q=:[y : ys]
| x ≤ y = [x : merge xs q]
| otherwise = [y : merge p ys]

Start = merge [2,5,7] [1,5,6,8] // [1,2,5,5,6,7,8]
Start = merge [] [1,2,3] // [1,2,3]
Start = merge [1,2,10] [] // [1,2,10]
Start = merge [2,1] [4,1] // [2,1,4,1]
Start = merge [1,2] [1,4] // [1,1,2,4]
```

Mergesort

```
msort :: [a] → [a] | Ord a
msort xs
| len ≤ 1 = xs
| otherwise = merge (msort ys) (msort zs)
where
    ys = take half xs
    zs = drop half xs
    half = len / 2
    len = length xs

Start = msort [2,9,5,1,3,8] // [1,2,3,5,8,9]
```

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 = iterate1 (λ x= x/10) 54321 // [54321,5432,543,54,5,0,0...]
```

Prime numbers

```
divisible :: Int Int → Bool
divisible x n = x rem n == 0

denominators :: Int → [Int]
denominators x = filter (divisible x) [1..x]

prime :: Int → Bool
prime x = denominators x == [1,x]

primes :: Int → [Int]
primes x = filter prime [1..x]

Start = primes 100 // [2,3,5,7,...,97]
```

Prime numbers

```
sieve :: [Int] → [Int]
sieve [p:xs] = [p: sieve [ i \\ i ← xs | i rem p ≠ 0]]

Start = take 100 (sieve [2..])
```

Review

- Functions, operators, basic type
- Lists, generators
- Higher order functions
- Tuples
- Recursive functions, infinite lists

Exercise

```
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
```