

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 Lists

2 Higher order functions



init, last, flatten

init selects everything but the last element (compare with last).

```
init :: [a] → [a]
init [x] = []
init [x : xs] = [x : init xs]
```

```
last :: [a] → [a]
last [x] = x
last [x : xs] = last xs
```

```
flatten :: [[a]] → [a]
flatten [] = []
flatten [x : xs] = x ++ flatten xs
```



Comparing and ordering lists

Equality of lists (operators are also functions written between the arguments)

$(=) :: [a] [a] \rightarrow \text{Bool} \mid = a$

$(=) [] [] = \text{True}$

$(=) [] [y : ys] = \text{False}$

$(=) [x : xs] [] = \text{False}$

$(=) [x : xs] [y : ys] = x = y \ \&\& \ xs = ys$



Ordering lists

Lexicographical ordering (dictionary ordering)

E.g. $[2, 3] < [3, 0]$ or $[10, 1] < [10, 2]$

$(<) :: [a] [a] \rightarrow \text{Bool} \mid <, = a$

$(<) [] [] = \text{False}$

$(<) [] _ = \text{True}$

$(<) _ [] = \text{False}$

$(<) [x : xs] [y : ys] = x < y \mid \mid (x = y \ \&\& \ xs < ys)$



Other comparisons

Once we have $<$ and $==$ all others can be defined.

$(<>) \ x \ y = \text{not } (x == y)$

$(>) \ x \ y = y < x$

$(>=) \ x \ y = \text{not } (x < y)$

$(<=) \ x \ y = \text{not } (y < x)$



Higher order functions

```
map :: (a → b) [a] → [b]
map f [] = []
map f [x:xs] = [f x : map f xs]
```

```
Start = map inc [1, 2, 3]           // [2, 3, 4]
Start = map double [1, 2, 3]       // [2, 4, 6]
```

// lambda expressions

```
Start = map (λx = x*x+2*x+1) [1..10] // [4,9,16,25,36,49,64,81,100,121]
```

// mapfun [f,g,h] x = [f x, g x, h x]

```
mapfun [] x = []
mapfun [f : fs] x = [f x : mapfun fs x]
```

```
Start = mapfun [inc, inc, inc] 3 // [4, 4, 4]
```



Filtering

// function parameters

```
filter :: ( a → Bool) [a] → [a]
```

```
filter p [] = []
```

```
filter p [x : xs]
```

```
| p x = [x : filter p xs]
```

```
| otherwise = filter p xs
```

```
Start = filter isEven [1..10] // [2,4,6,8,10]
```

```
odd x = not (isEven x)
```

```
Start = odd 23 // True
```

```
Start = filter (not o isEven) [1..100] // [1,3,5,...,99]
```



Filtering

```
filter p [] = []  
filter p [x : xs]  
| p x = [x : filter p xs]  
| otherwise = filter p xs  
Start = filter isEven [2,4,6,7,8,9] // [2, 4, 6, 8]
```

```
takeWhile :: (a → Bool) [a] → [a]  
takeWhile p [] = []  
takeWhile p [x : xs]  
| p x = [x : takeWhile p xs]  
| otherwise = []  
Start = takeWhile isEven [2,4,6,7,8,9] // [2, 4, 6]
```

```
dropWhile p [] = []  
dropWhile p [x : xs]  
| p x = dropWhile p xs  
| otherwise = [x : xs]  
Start = dropWhile isEven [2,4,6,7,8,9] // [7, 8, 9]
```



Folding

`foldr op e [] = e`

`foldr op e [x : xs] = op x (foldr op e xs)`

`foldr (+) 0 [1,2,3,4,5] → (1 + (2 + (3 + (4 + (5 + 0)))))`

`Start = foldr (+) 10 [1, 2, 3] // 16`

`product [] = 1`

`product [x:xs] = x * product xs`

`product x = foldr (*) 1 x`



Compositions, function parameters

// function composition

`twiceof :: (a → a) a → a`

`twiceof f x = f (f x)`

`Start = twiceof inc 0 // 2`

// Evaluation:

`twiceof inc 0`

`→ inc (inc 0)`

`→ inc (0+1)`

`→ inc 1`

`→ 1+1`

`→ 2`

`Twice :: (t→t) → (t→t)`

`Twice f = f o f`

`Start = Twice inc 2 // 4`

`f = g o h o i o j o k` is nicer than `f x = g(h(i(j(k x))))`



Partial parameterization

Calling a function with fewer arguments than it expects.

```
plus x y = x + y
successor :: (Int → Int)
successor = plus 1
Start = successor 4 // 5
succ = (+) 1
Start = succ 5 // 6
```

// the function adding 5 to something

```
Start = map (plus 5) [1,2,3] // [6,7,8]
```

```
plus :: Int → (Int→Int)
accepts an Int and returns the successor function of type Int→Int
```

Currying: treats equivalently the following two types
 $\text{Int Int} \rightarrow \text{Int}$ and $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$



Writing equivalences

```
product [] = 1  
product [x:xs] = x * product xs
```

```
product = foldr (*) 1
```

```
and [] = True  
and [x:xs] = x && and xs
```

```
and = foldr (&&) True
```

```
sum [] = 0  
sum [x:xs] = x + product xs
```

```
sum = foldr (+) 0
```



Iteration

// compute f until p holds

```
until p f x
```

```
| p x = x
```

```
| otherwise = until p f (f x)
```

```
Start = until ((<)10) ((+)2) 0 // 12
```

// iteration of a function

```
iterate :: (t → t) t → [t]
```

```
iterate f x = [x : iterate f (f x)]
```

```
Start = iterate inc 1 // infinite list [1..]
```

