CS 558 — 3 September 2024

Note Title 2024-09-03

Syntax for binary operators

$a \to b \to c$

Alphanumeric names

div mod

Prefix by default

div 7 2 ↪ 3

To use them infix, we need backquotes

7 `div` 2

Non-alphanumeric names # : ==

Infix by default

"abc" # "def"

Can use them as prefix by enclosing in parentheses:

(++) "abc" "def"

Higher-order functions over lists "Functionals"

Helper function:

```
isCap :: Char -> Bool
isCap c = 'A' <= c && c <= 'Z'
```

```
keepOnlyCaps :: [Char] -> [Char]
keepOnlyCaps [] = []
keepOnlyCaps (c : cs)
  | isCap c   = c : keepOnlyCaps cs
  | otherwise = keepOnlyCaps cs
```

Example evaluation. keepOnlyCaps ['A', 'b', 'X']

= 'A' : keepOnlyCaps ['b', 'X']

= 'A' : keepOnlyCaps ['X']

= 'A' : 'X' : keepOnlyCaps []

= 'A' : 'X' : []

Higher-order function filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

(a -> Bool): function argument "predicate"

```
filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Now we can define keepOnlyCaps in this way instead:

```
keepOnlyCaps :: [Char] -> [Char]
keepOnlyCaps cs = filter isCap cs
```

OR

```
keepOnlyCaps = filter isCap
```

```
keepAllButCaps = filter (not . isCap)
```

List of all even positive integers:

```
filter (\n -> n `mod` 2 == 0) [1..]
```

\ a function which, given some argument n (a number) computes if it is even

---

$[n*2 \mid n \leftarrow [1,2,3]]$

$*2$

$[2,4,6]$

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

```
map (\n -> n*2) [1,2,3]
```

(*2)

```
map (*2) [1,2,3]
= (*2) 1 : map (*2) [2,3]
= 2 : map (*2) [2,3]
= 2 : (*2) 2 : map (*2) [3]
= 2 : 4 : map (*2) [3]
= 2 : 4 : (*2) 3 : []
= 2 : 4 : 6 : []
```

```
map isCap ['X','b']
= [True, False]
```

```
map isCap :: [Char] -> [Bool]
```

Aggregation / Combing / Reducing / Folding

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

(a -> b -> b): combing operator / aggregating function; b: starting value; [a]: list to be aggregated; b: result of aggregation

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

example implement ~~sst~~ summation using foldr

```
Sum :: [Int] -> Int
Sum = foldr (+) 0
```

```
sum [1,2,3]
= foldr (+) 0 [1,2,3]
= (+) 1 (foldr (+) 0 [2,3])
= (+) 1 ((+) 2 (foldr (+) 0 [3]))
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
= (+) 1 ((+) 2 ((+) 3 0))
= (+) 1 ((+) 2 3)
= (+) 1 5
= 6
```