higher order functions

• take functions as arguments
• return functions as results
• or both

\[
\text{doubleL} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{double} \ [\text{x : xs}] = [2 \times \text{x : doubleL} \ [\text{xs}]]
\]
\[
\text{trebleL} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{trebleL} \ [\text{x : xs}] = [3 \times \text{x : trebleL} \ [\text{xs}]]
\]

\[
\text{map} \ f \ [\text{x : xs}] = [f \text{x} | \text{x : xs}]
\]
\[
\text{map} \ f \ [\text{xs}] = [f \ [\text{xs}] = [f \text{x : map} \ f \ [\text{xs}]]
\]

\[
\text{doubleL} \ [\text{x : xs}] = \text{map} \ \text{twice} \ [\text{x : xs}]
\]
\[
\text{where twice} \ x = 2 \times \text{x}
\]
### Higher Order Functions

**map**: 

- **Function**: `(a → b) → [a] → [b]`
- **IN**: Values for which the function can be applied
- **OUT**: The type of values after applying the function

**map** - apply some function to every element of a list thus yielding another list.

---

**doubleL**

```haskell
xs = map twice xs

where twice x = 2 * x
```

**cnvrtC**

```haskell
:: [Char] → [Int]
cnvrtC xs = map ord xs
```

**doubleLambda**

```haskell
xs = map (\x -> 2 * x) xs
```

---

- **cnvrtC**: `char to int`
  - `cnvrtC "Stefan"` gives `[83,116,101,102,97,110]`
  - `cnvrtC "abcd"` gives `[97,98,99,100]`

- **doubleLambda**: `apply function twice`
  - `doubleLambda [2, 7, 3, 12]` gives `[4, 14, 6, 24]`
  - `doubleLambda []` gives `[]`
properties as functions

getDigits :: [Char] -> [Char]
getDigits s = [c | c <- s, isDigit c]

isDigit :: Char -> Bool
isDigit c = (c >= '0' && c <= '9')

getDigits "a 1 2 b 3 c d 7 x y" -> 1 2 3 7

x has a property f if \( f(x) = True \)

isEven :: Int -> Bool
isEven n = (mod n 2 == 0)

isSorted :: [Int] -> Bool
isSorted xs = (xs == qSort xs)

filtering

filter f [] = []
filter f (x : xs)
  | f x = x : filter f xs
  | otherwise = filter f xs

filter isSorted [[2,3,4,5], [], [7,3,6]] -> [[2,3,4,5], []]
**Higher Order Functions**

**foldr1** :: (a -> a -> a) -> [a] -> a

\[foldr1 \xi [e_1, e_2, e_3, ..., e_n] = [e_1 \xi (e_2 \xi (... \xi e_n ...))] = [e_1 \xi (foldr1 \xi [e_2, e_3, ..., e_n])]

---

**foldr1** (+) [1, 2, 3, 4] = 10

**foldr1** (+) [1] = 1

**foldr1** (+) [] = Program error: {foldr1 (instNum v30 Num_+) [ ]}

**foldr1** (||) [True, False, False] = True

**foldr1** (++) ["Dark", "side", "  ", "of"] = "Darkside of"

**foldr1** (*) [1 .. 7] = 5040
higher order functions

- take functions as arguments
- return functions as results
- or both

sequence of processes:
for every process \( i \in P \), \( \text{OUT-process}_i \rightarrow \text{IN-process}_{i+1} \)

function composition

\[
(f \cdot g) x = f(g(x))
\]

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)
\]

- type of \( f \)
- type of \( g \)
- type of \( f \cdot g \)

Prelude> and [(5 == 5), (3 > 5)]
True
Prelude> (not . and) [(5 == 5), (3 > 5)]
False
Prelude> \text{cos (sin pi)}
1.0
Prelude> \text{cos (sin pi)}
1.0
Prelude>
twice :: (a -> a) -> (a -> a)
twice = (\f -> f . f)

Main> succ 110
    111
Main> succ (succ 110)
    112
Main> (twice succ) 110
    112
Main>

... thrice, four-times, ..., n-times

ntimes :: Int -> (a -> a) -> (a -> a)
ntimes n f |
| n > 0   = f . ntimes (n-1) f |
| otherwise       = id

Main> twice succ 110
    112
Main> ntimes 2 succ 110
    112
Main> ntimes 1 succ 110
    111
Main> ntimes 0 succ 110
    110
Main> ntimes 5 succ 110
    115
Main>