higher order functions

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

```
doubleL :: [ Int ] -> [ Int ]
double  xs = [ 2 * x | x <- xs ]
```

```
doubleL :: [ Int ] -> [ Int ]
doubleL [ ]       = [ ]
doubleL [ x : xs ] = [ 2 * x : doubleL xs ]
```

```
trebleL :: [ Int ] -> [ Int ]
trebleL  xs = [ 3 * x | x <- xs ]
```

```
trebleL :: [ Int ] -> [ Int ]
trebleL [ ]       = [ ]
trebleL [ x : xs ] = [ 3 * x : trebleL xs ]
```

sin x
ord x
.....


```
map f  xs = [ f x | x <- xs ]
```

```
map f [ ]       = [ ]
map f [ x : xs ] = [ f x : map f xs ]
```

```
doubleL  xs = map  twice xs
              where twice x = 2 * x
```

function IN OUT

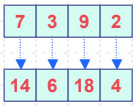
```
map :: (a -> b) -> [a] -> [b]
```

values for which the function can be applied

the type of values after applying the function

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

```
doubleL xs = map twice xs
             where twice x = 2 * x
```




```
-- lambda notation for local function defn
doubleLambda xs = map (\x -> 2 * x) xs
```

```
Main> doubleLambda [2, 7, 3, 12]
[4,14,6,24]
Main> doubleLambda [ ]
[ ]
Main> doubleLambda [1]
[2]
Main>
```

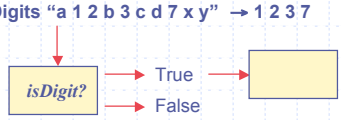
```
cnvrtC :: [ Char ] -> [ Int ]
cnvrtC xs = map ord xs
```

```
Main> cnvrtC "Stefan"
[83,116,101,102,97,110]
Main> cnvrtC ['a', 'b', 'c', 'd']
[97,98,99,100]
```

properties as functions

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

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



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

property f over type t t → Bool

```
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  f  xs = [ x  | x <- xs, f  x ]
```

filter isSorted [[2,3,4,5], [ ], [7,3,6]] → [[2,3,4,5], [ ]]

folding

$\mathbf{foldr1}\ \xi\ [e_1, e_2, e_3, \ldots, e_n] =$
$= [e_1\ \xi\ (e_2\ \xi\ (\ldots\ \xi\ e_n\ \ldots)$
$= [e_1\ \xi\ (\mathbf{foldr1}\ \xi\ [e_1, e_2, e_3, \ldots, e_n])$

$\mathbf{foldr1}\ (+)\ [e_1, e_2, e_3]$
$= e_1\ (+)\ (\mathbf{foldr1}\ (+)\ [e_2, e_3])$
$= e_1\ (+)\ e_2\ (+)\ e_3$

binary function over type a

result

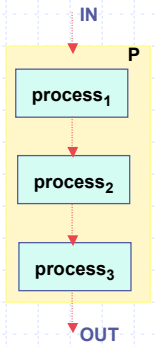
```
foldr1 :: (a -> a ->a) -> [a] -> a
foldr1 f [x]  = x
foldr1 f (x : xs) = f x (foldr1  f  xs)
-- at least one element in the list x
```

```
Main> foldr1 (+) [1,2,3,4]
10
Main> foldr1 (+) [1]
1
Main> foldr1 (+) [ ]
Program error: {foldr1 (instNum_v30 Num_+) [ ]}
Main> foldr1 (||) [True, False, False]
True
Main> foldr1 (++) ["Dark", "side", "  ", "of" ]
"Darkside  of"
Main> foldr1 (*) [1 .. 7]
5040
```

higher order functions

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

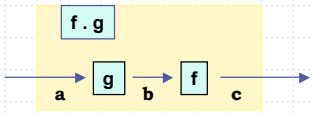
```
cs3 % date
Saturday October 26 14:47:54 BST 2002
cs3 % f | grep p00 | cut -c48-58
 Mon 10:18
 Mon 16:23
 Sat  14:32
 Tue  14:38
 Mon 10:30
cs3 %
```



sequence of processes:

for every $process_i \in P$, $OUT\text{-}process_i \rightarrow IN\text{-}process_{i+1}$

function composition



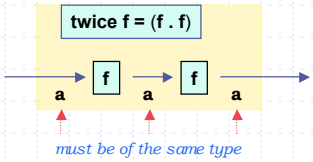

```
(f . g) x = f (g x)

(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

type of f *type of g* *type of f . g*

```
Prelude> and [(5 == 5), (3 > 5)]
False
Prelude> (not . and) [(5 == 5), (3 > 5)]
True
Prelude>cos (sin pi)
1.0
Prelude> (cos . sin) pi
1.0
Prelude>
```

twice -- function on function

```
twice f = (f . f)
```



must be of the same type

(twice) :: (a -> a) -> (a -> a)

```
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
```

identity

```
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>
```