Lecture 6 – Higher-ordered Functions

Violet Ka I Pun

violet@ifi.uio.no

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter `filter (> 5) [1,3,5,7,9] = [7,9]`

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```

filter
```
::
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```

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

all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```
all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```
all
```
::
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```
all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```
all
```
:: (a -> Bool) -> [a] -> Bool
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```
all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```
all
```
:: (a -> Bool) -> [a] -> Bool
```
any
```
any (<10) [1,3,5,7,9,10] = True
any (/=5) [1,2,3,4,5] = True
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```
all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```
all
```
:: (a -> Bool) -> [a] -> Bool
```
any
```
any (<10) [1,3,5,7,9,10] = True
any (/=5) [1,2,3,4,5] = True
```
any
```
::
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

filter
```
filter (> 5) [1,3,5,7,9] = [7,9]
filter (/=3) [1,2,3,4,5] = [1,2,4,5]
```
filter
```
:: (a -> Bool) -> [a] -> [a]
```
all
```
all (<10) [1,3,5,7,9] = True
all (/=5) [1,2,3,4,5] = False
```
all
```
:: (a -> Bool) -> [a] -> Bool
```
any
```
any (<10) [1,3,5,7,9,10] = True
any (/=5) [1,2,3,4,5] = True
```
any
```
:: (a -> Bool) -> [a] -> Bool
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                               ++ [x] ++ qs [y | y <- xs, y>x]
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                              ++ [x] ++ qs [y | y <- xs, y>x]
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                          ++ [x] ++ qs [y | y <- xs, y>x]
  qs :: Ord a => [a] -> [a]
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                              ++ [x] ++ qs [y | y <- xs, y>x]
  qs :: Ord a => [a] -> [a]
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                          ++ [x] ++ qs [y | y <- xs, y>x]
  qs :: Ord a => [a] -> [a]

  gqs f [] = []
  gqs f (x:xs) = qs [y | y <- xs, f y x]
                    ++ [x] ++ qs [y | y <- xs, not(f y x)]
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                          ++ [x] ++ qs [y | y <- xs, y>x]
  qs :: Ord a => [a] -> [a]

  gqs f [] = []
  gqs f (x:xs) = qs [y | y <- xs, f y x]
                    ++ [x] ++ qs [y | y <- xs, not(f y x)]
  gqs ::
  ```
  ```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, e.g.:
- ```
  qs [] = []
  qs (x:xs) = qs [y | y <- xs,y<=x]
                          ++ [x] ++ qs [y | y <- xs, y>x]
  qs :: Ord a => [a] -> [a]

  gqs f [] = []
  gqs f (x:xs) = qs [y | y <- xs, f y x]
                    ++ [x] ++ qs [y | y <- xs, not(f y x)]
  gqs :: (a -> a -> Bool) -> [a] -> [a]
  ```
  ```

e.g.: **lex** `a b =`

Higher-order functions

e.g.: **lex** `a b =` **if** $a<b$ **then** `LT`
else if $a>b$ **then** `GT`
else `EQ`

```
lex a b = if a<b then LT
                 else if a>b then GT
                             else EQ
```

Higher-order functions

e.g.: **lex** `a b =` **if** `a<b` **then** `LT`
else if `a>b` **then** `GT`
else `EQ`

lgh `a b =` **if** `length a` $<$ `length b` **then** `LT`
else if `length a` $>$ `length b` **then** `GT` **else** `EQ`

Higher-order functions

e.g.: **lex** `a b =` **if** `a<b` **then** `LT`
else if `a>b` **then** `GT`
else `EQ`

lgh `a b =` **if** `length a` < `length b` **then** `LT`
else if `length a` > `length b` **then** `GT` **else** `EQ`

- `data Ordering = LT|GT|EQ deriving...`

Higher-order functions

e.g.: **lex** a b = **if** a<b **then** LT
else if a>b **then** GT
else EQ

lgh a b = **if** length a < length b **then** LT
else if length a > length b **then** GT **else** EQ

- data Ordering = LT|GT|EQ deriving...
- Data.List has a function
  sortBy :: (a->a->Ordering)->[a]->[a]

Higher-order functions

e.g.: **lex** `a b =` **if** `a<b` **then** `LT`
else if `a>b` **then** `GT`
else `EQ`

lgh `a b =` **if** `length a` < `length b` **then** `LT`
else if `length a` > `length b` **then** `GT` **else** `EQ`

- `data Ordering = LT|GT|EQ deriving...`
- `Data.List` has a function
  `sortBy :: (a->a->Ordering)->[a]->[a]`
- **sortBy lex** x – sort x by < (lists in x: lexicographical)

Higher-order functions

e.g.: **lex** `a b =` **if** `a<b` **then** `LT`
else if `a>b` **then** `GT`
else `EQ`

lgh `a b =` **if** `length a < length b` **then** `LT`
else if `length a > length b` **then** `GT` **else** `EQ`

- `data Ordering = LT|GT|EQ deriving...`
- `Data.List` has a function
  `sortBy :: (a->a->Ordering)->[a]->[a]`
- **sortBy lex** x – sort x by < (lists in x: lexicographical)
- **sortBy lgh** x – will sort lists in x by their length

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, . . .

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, . . .
- . . . and output
  – for example, sections:

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:
  `([2]++) :: Num a => [a] -> [a]`

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, . . .
- . . . and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) ->  a -> c
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

e.g. `odd x = not(even x)`

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

e.g. `odd x = not(even x)` ....shorter: `odd = not . even`

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

e.g. `odd x = not(even x)` ....shorter: `odd = not . even`

`twice f x = f (f x)` ....shorter:

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output

  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

e.g.
```
odd x = not(even x) .........shorter:  odd = not . even
twice f x = f (f x) ..........shorter:  twice f = f . f
```

Higher-ordered functions

- Functions are first-class citizens in FP
- Functions can be input, ...
- ...and output
  – for example, sections:

```
([2]++) :: Num a => [a] -> [a]
   (++) :: [a] -> ([a] -> [a])
zip [1,2,3] :: Num t => [b] -> [(t, b)]
        zip :: [a] -> ([b] -> [(a, b)])
```

- in- and output

```
f . g = \x -> f ( g x )        ≠ \x -> f g x
. :: (b -> c) -> (a -> b) -> (a -> c)
```

e.g.
```
odd x = not(even x) .........shorter:  odd = not . even
twice f x = f (f x) ..........shorter:  twice f = f . f
.................................. twice = \f -> f . f
```

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw ::
```

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` $(=$ `\x -> (1+) x`$)$:

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`
   $=$ `(2+) ::`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` $($= `\x -> (1+) x`$)$:

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`
   $=$ `(2+) :: Int -> Int`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`
   $=$ `(2+) :: Int -> Int`

2. `tw (4+) 2`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)`
   = `s` ∘ `s`
   = `(2+) :: Int -> Int`

2. `tw (4+) 2` = `10`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`
   $=$ `(2+) :: Int -> Int`

2. `tw (4+) 2` $=$ `10`

   `tw (6+) 2`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` $=$ `\x -> s(s x)`
   $=$ `s` $\circ$ `s`
   $=$ `(2+) :: Int -> Int`

2. `tw (4+) 2` $=$ `10`

   `tw (6+) 2` $=$ `14`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)`
   = `s ∘ s`
   = `(2+) :: Int -> Int`

2. `tw (4+) 2` = `10`

   `tw (6+) 2` = `14`

3. `tw (++ '' HaHa...'') ''Hei!''`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)`
   = `s ∘ s`
   = `(2+) :: Int -> Int`

2. `tw (4+) 2` = `10`

   `tw (6+) 2` = `14`

3. `tw (++ ‘‘ HaHa...’’) ‘‘Hei!’’` = `‘‘Hei! HaHa... HaHa...’’`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1.	`tw s`	=	`\x -> s(s x)`
		=	`s` ∘ `s`
		=	`(2+) :: Int -> Int`
2.	`tw (4+) 2`	=	`10`
	`tw (6+) 2`	=	`14`
3.	`tw (++ `` HaHa...'') ``Hei!''`	=	```` ``Hei! HaHa... HaHa...'' ````
	`` tw (``HaHa...'' ++) ``Hei!'' ````		

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)` = `s ∘ s` = `(2+) :: Int -> Int`
2. `tw (4+) 2` = `10`

   `tw (6+) 2` = `14`
3. `tw (++ ‘‘ HaHa...’’) ‘‘Hei!’’` = `‘‘Hei! HaHa... HaHa...’’`

   `tw (‘‘HaHa...’’ ++) ‘‘Hei!’’` = `‘‘HaHa...HaHa...Hei!’’`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)`
   = `s` ∘ `s`
   = `(2+) :: Int -> Int`

2. `tw (4+) 2` = `10`

   `tw (6+) 2` = `14`

3. `tw (++ `` HaHa...'') ``Hei!''` = ```Hei! HaHa... HaHa...''`

   `tw (``HaHa...'' ++) ``Hei!''` = ```HaHa...HaHa...Hei!''`

4. `tw (3:) [5,6,7]`

In- and output: `tw f x = f (f x)`

```
tw = \f -> \x -> f (f x)
tw :: (t -> t) -> t -> t
```

Given `s = (1+)` (= `\x -> (1+) x`):

1. `tw s` = `\x -> s(s x)` = `s ∘ s` = `(2+) :: Int -> Int`
2. `tw (4+) 2` = `10`

   `tw (6+) 2` = `14`
3. `tw (++ `` HaHa...'') ``Hei!''` = ```Hei! HaHa... HaHa...''``

   `tw (``HaHa...'' ++) ``Hei!''` = ```HaHa...HaHa...Hei!''``
4. `tw (3:) [5,6,7]` = `[3,3,5,6,7]`

List functions ‘element-wise’

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

List functions 'element-wise'

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

FP:
```
f [] = []
f (h:t) = (... h) : f t
```

List functions 'element-wise'

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

FP:
```
f [] = []
f (h:t) = (... h) : f t
```

- Examples:

List functions 'element-wise'

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

FP:
```
f [] = []
f (h:t) = (... h) : f t
```

- Examples:
  - ```
    copy [] = []
    copy (h:t) = h : copy t
    ```
    ```

List functions 'element-wise'

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

FP:
```
f [] = []
f (h:t) = (... h) : f t
```

- Examples:
  - ```
    copy [] = []
    copy (h:t) = h : copy t
    ```
  - ```
    liad1 [] = []
    liad1 (h:t) = h+1 : liad1 t
    ```

List functions ‘element-wise’

IP: `for (int i=0; i<a.length; ++i) a[i] = ...`

FP:
```
f [] = []
f (h:t) = (... h) : f t
```

- Examples:
  - ```
    copy [] = []
    copy (h:t) = h : copy t
    ```
  - ```
    liad1 [] = []
    liad1 (h:t) = h+1 : liad1 t
    ```
  - ```
    lisq [] = []
    lisq (h:t) = h*h : lisq t
    ```
    ```

Functions ‘element-wise’

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

Functions ‘element-wise’

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

Functions ‘element-wise’

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

Functions 'element-wise'

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

Functions 'element-wise'

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

Functions 'element-wise'

- In general:

```
  copy [] = []
copy (h:t) = h : copy t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

Functions 'element-wise'

- In general:

```
  copy [] = []
copy (h:t) = h : copy t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

```
copy = map
```

Functions 'element-wise'

- In general:

```
 copy [] = []
copy (h:t) = h : copy t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

```
copy = map (\x -> x) ::
```

Functions 'element-wise'

- In general:

```
  copy [] = []
copy (h:t) = h : copy t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x) :: [a] -> [a]`

Functions ‘element-wise’

- In general:

```
      copy [] = []
  copy (h:t) = h : copy t
     liad1 [] = []
liad1 (h:t) = h+1 : liad1 t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x) ::` [a] -> [a]

Functions 'element-wise'

- In general:

```
   copy [] = []
 copy (h:t) = h : copy t
   liad1 [] = []
liad1 (h:t) = h+1 : liad1 t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x)` :: [a] -> [a]

  `liad1 = map`

Functions 'element-wise'

- In general:

```
   copy [] = []
 copy (h:t) = h : copy t
   liad1 [] = []
liad1 (h:t) = h+1 : liad1 t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x)` :: [a] -> [a]

  `liad1 = map (\x -> x+1) ::`

Functions ‘element-wise’

- In general:

```
      copy [] = []
  copy (h:t) = h : copy t
     liad1 [] = []
 liad1 (h:t) = h+1 : liad1 t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x) ::` [a] -> [a]

  `liad1 = map (\x -> x+1) :: Num a => [a] -> [a]`

Functions 'element-wise'

- In general:

```
   copy [] = []
copy (h:t) = h : copy t
  liad1 [] = []
liad1 (h:t) = h+1 : liad1 t
   lisq [] = []
 lisq (h:t) = h*h : lisq t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x) :: [a] -> [a]`

  `liad1 = map (\x -> x+1) :: Num a => [a] -> [a]`

Functions 'element-wise'

- In general:

```
      copy [] = []
  copy (h:t) = h : copy t
     liad1 [] = []
 liad1 (h:t) = h+1 : liad1 t
      lisq [] = []
  lisq (h:t) = h*h : lisq t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

  `copy = map (\x -> x) :: [a] -> [a]`

  `liad1 = map (\x -> x+1) :: Num a => [a] -> [a]`

  `lisq = map`

- In general:

```
    copy [] = []
copy (h:t) = h : copy t
   liad1 [] = []
liad1 (h:t) = h+1 : liad1 t
    lisq [] = []
 lisq (h:t) = h*h : lisq t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

```
copy = map (\x -> x) :: [a] -> [a]
liad1 = map (\x -> x+1) :: Num a => [a] -> [a]
lisq = map (\x -> x*x) ::
```

Functions 'element-wise'

- In general:

```
    copy [] = []
    copy (h:t) = h : copy t
    liad1 [] = []
    liad1 (h:t) = h+1 : liad1 t
    lisq [] = []
    lisq (h:t) = h*h : lisq t
```

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (h:t) = f h : map f t
```

e.g. `map f [1,2,3]` gives `[f 1, f 2, f 3]`

- Now:

```
copy = map (\x -> x) :: [a] -> [a]
liad1 = map (\x -> x+1) :: Num a => [a] -> [a]
lisq = map (\x -> x*x) :: Num a => [a] -> [a]
```

More examples of `map`

- Given:

More examples of `map`

- Given:
  - `fst(x,y) = x`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,''one''),(2,''two''),(3,''three'')]`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[1,2,3]`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,``one''),(2,``two''),(3,``three'')]`

  .... `[1,2,3]`
- `map snd [(1,``one''),(2,``two''),(3,``three'')]`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,``one''),(2,``two''),(3,``three'')]`
  .... `[1,2,3]`
- `map snd [(1,``one''),(2,``two''),(3,``three'')]`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,''one''),(2,''two''),(3,''three'')]`
  .... `[1,2,3]`
- `map snd [(1,''one''),(2,''two''),(3,''three'')]`
  .... `[''one'',''to'',''three'']`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[1,2,3]`
- `map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[‘‘one’’,‘‘to’’,‘‘three’’]`
- `concat (map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[1,2,3]`
- `map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[‘‘one’’,‘‘to’’,‘‘three’’]`
- `concat (map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`

More examples of map

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  `.................................................... [1,2,3]`
- `map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  `................................ [‘‘one’’,‘‘to’’,‘‘three’’]`
- `concat (map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`
  `............................................ ‘‘onetwothree’’`

More examples of map

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[1,2,3]`
- `map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[‘‘one’’,‘‘to’’,‘‘three’’]`
- `concat (map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`
  .... `‘‘onetwothree’’`
- `map sq (map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,''one''),(2,''two''),(3,''three'')]`
  .... `[1,2,3]`
- `map snd [(1,''one''),(2,''two''),(3,''three'')]`
  .... `[''one'',''to'',''three'']`
- `concat (map snd [(1,''one''),(2,''two''),(3,''three'')])`
  .... `''onetwothree''`
- `map sq (map fst [(1,''one''),(2,''two''),(3,''three'')])`

More examples of `map`

- Given:
  - `fst(x,y) = x`
  - `snd(x,y) = y`
  - `sq(n) = n*n`
- `map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[1,2,3]`
- `map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)]`
  .... `[‘‘one’’,‘‘to’’,‘‘three’’]`
- `concat (map snd [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`
  .... `‘‘onetwothree’’`
- `map sq (map fst [(1,‘‘one’’),(2,‘‘two’’),(3,‘‘three’’)])`
  .... `[1,4,9]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m`
- `[head m]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` .... `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` .... `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` .... `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` ....

```
[ [(1,2),(1,3),(1,4)],
  [(2,2),(2,3),(2,4)],
  [(3,2),(3,3),(3,4)] ]
```

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` ....
  ```
  [ [(1,2),(1,3),(1,4)],
    [(2,2),(2,3),(2,4)],
    [(3,2),(3,3),(3,4)] ]
  ```

How to get the n^{th} row ?

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)],`
  `[(2,2),(2,3),(2,4)],`
  `[(3,2),(3,3),(3,4)] ]`

How to get the n^{th} row ? ———————— row n m = m !! (n-1)

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)],`
  `[(2,2),(2,3),(2,4)],`
  `[(3,2),(3,3),(3,4)] ]`

How to get the n^{th} row ? ———————— row n m = m !! (n-1)

- `row 2 m`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)],`
  `[(2,2),(2,3),(2,4)],`
  `[(3,2),(3,3),(3,4)] ]`

How to get the n^{th} row ? ———————— row n m = m !! (n-1)

- `row 2 m` .... `[(2,1),(2,2),(2,3),(2,4)]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` .... `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)], [(2,2),(2,3),(2,4)], [(3,2),(3,3),(3,4)] ]`

How to get the nth row ? ———————————— row n m = m !! (n-1)

... kth column?

- `row 2 m` .... `[(2,1),(2,2),(2,3),(2,4)]`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)],`
  `[(2,2),(2,3),(2,4)],`
  `[(3,2),(3,3),(3,4)] ]`

How to get the n[th] row ? ———————— row n m = m !! (n-1)

... k[th] column? ——— col k m = map (\x -> x !! (k-1)) m

- `row 2 m` .... `[(2,1),(2,2),(2,3),(2,4)]`

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` ....
  .. `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)],`
  `[(2,2),(2,3),(2,4)],`
  `[(3,2),(3,3),(3,4)] ]`

How to get the n^{th} row ? ———————— row n m = m !! (n-1)

... k^{th} column? ——— col k m = map (\x -> x !! (k-1)) m

- `row 2 m` .... `[(2,1),(2,2),(2,3),(2,4)]`
- `col 2 m`

Example: matrices

```
m = [ [(1,1),(1,2),(1,3),(1,4)],
      [(2,1),(2,2),(2,3),(2,4)],
      [(3,1),(3,2),(3,3),(3,4)] ]
```

What does the following give?

- `head m` .... `[(1,1),(1,2),(1,3),(1,4)]`
- `[head m]` .... `[[(1,1),(1,2),(1,3),(1,4)]]`
- `tail m` .... `[[(2,1),(2,2),(2,3),(2,4)],[(3,1),(3,2),(3,3),(3,4)]]`
- `map head m` .... `[(1,1),(2,1),(3,1)]`
- `map tail m` .... `[ [(1,2),(1,3),(1,4)], [(2,2),(2,3),(2,4)], [(3,2),(3,3),(3,4)] ]`

How to get the n^th row ? ———————— row n m = m !! (n-1)

... k^th column? ——— col k m = map (\x -> x !! (k-1)) m

- `row 2 m` .... `[(2,1),(2,2),(2,3),(2,4)]`
- `col 2 m` .... `[(1,2),(2,2),(3,2)]`

Variations

- `map f l` has the same length as `l`

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                             else filter p t
  ```
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                              else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                          else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... ?
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                           else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`
  ```

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                            else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                           else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

  - `strip 1 [2,1,3,1,4,4,1,5]`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                             else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

  - `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                              else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

  - `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
  - `strip '1' ''21314415''`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                           else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

  - `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
  - `strip '1' ''21314415''` .... `''23445''`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                            else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

- `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
- `strip '1' ‘‘21314415’’` .... `‘‘23445’’`
- `strip ‘‘1’’ ‘‘21314415’’`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                             else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

- `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
- `strip '1' ‘‘21314415’’` .... `‘‘23445’’`
- `strip ‘‘1’’ ‘‘21314415’’` .... error
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                          else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

- `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
- `strip '1' ‘‘21314415’’` .... `‘‘23445’’`
- `strip ‘‘1’’ ‘‘21314415’’` .... error

  `strip :: Eq a => a -> [a] -> [a]`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                              else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

- `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
- `strip '1' ''21314415''` .... `''23445''`
- `strip ''1'' ''21314415''` .... error
  `strip :: Eq a => a -> [a] -> [a]`

e.g.: `g xs = sum (map (^2) (filter even xs))`
  ```

Variations

- `map f l` has the same length as `l`
- What if we want to filter out some elements `p :: a -> Bool`
- ```
  filter p [] = []
  filter p (h:t) = if p h then h:filter p t
                             else filter p t
  ```
- Better in Haskell: `filter p xs = [x | x <- xs, p x]`
- `filter (\x -> x>0) [1,-1,0,2,-2]` .... `[1,2]`

e.g.: `strip a l = filter (/= a) l`

- `strip 1 [2,1,3,1,4,4,1,5]` .... `[2,3,4,4,5]`
- `strip '1' ‘‘21314415’’` .... `‘‘23445’’`
- `strip ‘‘1’’ ‘‘21314415’’` .... error

  `strip :: Eq a => a -> [a] -> [a]`

e.g.: `g xs = sum (map (^2) (filter even xs))`

`g :: Num a => [a] -> [a]`
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!

`foldr` and `foldl`

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  f_all f v [] = v
  f_all f v (h:t) = f h (f_all f v t)
  ```
  ```

`foldr` and `foldl`

- `sum_all [] = 0`
  `sum_all (h:t) = h + sum_all t`
- `mul_all [] = 1`
  `mul_all (h:t) = h * mul_all t`
- Better in Haskell: `sum`, `product`, . . . !
- `f_all f v [] = v`
  `f_all f v (h:t) = f h (f_all f v t)`
- `sum_all = f_all (+) 0`

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  f_all f v [] = v
  f_all f v (h:t) = f h (f_all f v t)
  ```
- `sum_all = f_all (+) 0`
- `mul_all = f_all (*) 1`
  ```

`foldr` and `foldl`

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = f_all (+) 0`
- `mul_all = f_all (*) 1`
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = foldr (+) 0`
- `mul_all = f_all (*) 1`
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
  ```

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\text{foldr} \circ v\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$
  ```

foldr and foldl

- `sum_all [] = 0`
  `sum_all (h:t) = h + sum_all t`
- `mul_all [] = 1`
  `mul_all (h:t) = h * mul_all t`
- Better in Haskell: `sum`, `product`, ...!
- `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\text{foldr} \circ v\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

foldr and foldl

- `sum_all [] = 0`
  `sum_all (h:t) = h + sum_all t`
- `mul_all [] = 1`
  `mul_all (h:t) = h * mul_all t`
- Better in Haskell: `sum`, `product`, …!
- `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\text{foldr} \circ v \, [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

- foldr $(+)$ 0 [1,2,3]

foldr and foldl

- `sum_all [] = 0`
  `sum_all (h:t) = h + sum_all t`
- `mul_all [] = 1`
  `mul_all (h:t) = h * mul_all t`
- Better in Haskell: `sum`, `product`, ...!
- `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\text{foldr} \circ v \; [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

- foldr $(+)$ 0 [1,2,3] $=$ 1$+$(foldr $(+)$ 0 [2,3])

`foldr` and `foldl`

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\mathrm{foldr} \circ v\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

- foldr (+) 0 [1,2,3] = 1+(foldr (+) 0 [2,3])
  = 1+(2+(foldr (+) 0 [3]))
  ```

foldr and foldl

- `sum_all [] = 0`
  `sum_all (h:t) = h + sum_all t`
- `mul_all [] = 1`
  `mul_all (h:t) = h * mul_all t`
- Better in Haskell: `sum`, `product`, ...!
- `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  $\text{foldr} \circ v\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

- foldr (+) 0 [1,2,3] = 1+(foldr (+) 0 [2,3])
  = 1+(2+(foldr (+) 0 [3])) = 1+(2+(3+(foldr (+) 0 [])))

foldr and foldl

- ```
  sum_all [] = 0
  sum_all (h:t) = h + sum_all t
  ```
- ```
  mul_all [] = 1
  mul_all (h:t) = h * mul_all t
  ```
- Better in Haskell: `sum`, `product`, ...!
- ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```
- `sum_all = foldr (+) 0`
- `mul_all = foldr (*) 1`
- `foldr` is right-associative:
  foldr $\circ$ v $[x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ v)...)))$

i.e., is not tail-recursive...

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

Use accumulator

- `sum_all l = sum_tail 0 l`

Use accumulator

- ```
  sum_all l = sum_tail 0 l
  ```
  ```
  sum_tail v [] = v
  sum_tail v (h:t) = sum_tail v+h t
  ```
  ```

Use accumulator

- ```
  sum_all l = sum_tail 0 l
  sum_tail v [] = v
  sum_tail v (h:t) = sum_tail v+h t
  ```
- ```
  f_tail f v [] = v
  f_tail f v (h:t) = f_tail f (f v h) t
  ```

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `f_tail f v [] = v`

  `f_tail f v (h:t) = f_tail f (f v h) t`
- `f_tail` is left-associative, tail-recursive

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `f_tail f v [] = v`

  `f_tail f v (h:t) = f_tail f (f v h) t`
- `f_tail` is left-associative, tail-recursive
- `sum_all = f_tail (+) 0`

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `f_tail f v [] = v`

  `f_tail f v (h:t) = f_tail f (f v h) t`
- `f_tail` is left-associative, tail-recursive
- `sum_all = f_tail (+) 0`
- `mul_all = f_tail (*) 1`

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = f_tail (+) 0`
- `mul_all = f_tail (*) 1`

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = f_tail (*) 1`

Use accumulator

- ```
  sum_all l = sum_tail 0 l
  sum_tail v [] = v
  sum_tail v (h:t) = sum_tail v+h t
  ```
- ```
  foldl f v [] = v
  foldl f v (h:t) = foldl f (f v h) t
  ```
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

Use accumulator

- `sum_all l = sum_tail 0 l`
  ```
  sum_tail v [] = v
  sum_tail v (h:t) = sum_tail v+h t
  ```
- ```
  foldl f v [] = v
  foldl f v (h:t) = foldl f (f v h) t
  ```
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:
  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

- foldl $(+)$ 0 [1,2,3]
  ```

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

- foldl $(+)$ 0 [1,2,3] $=$ foldl $(+)$ 0+1 [2,3]

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

- foldl $(+)$ 0 $[1,2,3]$ $=$ foldl $(+)$ $0+1$ $[2,3]$

  $=$ foldl $(+)$ $(0+1)+2$ $[3]$

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

- foldl $(+)$ 0 [1,2,3] $=$ foldl $(+)$ $0+1$ [2,3]

  $=$ foldl $(+)$ $(0+1)+2$ [3] $=$ foldl $(+)$ $((0+1)+2)+3$ []

Use accumulator

- `sum_all l = sum_tail 0 l`

  `sum_tail v [] = v`

  `sum_tail v (h:t) = sum_tail v+h t`
- `foldl f v [] = v`

  `foldl f v (h:t) = foldl f (f v h) t`
- `foldl` is left-associative, tail-recursive
- `sum_all = foldl (+) 0`
- `mul_all = foldl (*) 1`
- `foldl` is left-associative:

  foldl $\circ$ b $[x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

and is tail-recursive.

- foldl $(+)$ 0 $[1,2,3] =$ foldl $(+)$ $0+1$ $[2,3]$

  $=$ foldl $(+)$ $(0+1)+2$ $[3] =$ foldl $(+)$ $((0+1)+2)+3$ $[]$

  $= ((0+1)+2)+3 = 6$

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call

- `foldr`: right recursion

  ```
  []--->v
  (h:t)--->(f h ...)
  ```
  where `...` stands for a recursive call

  ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```

List-recursive – summary

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

:t `foldr ::`

List-recursive – summary

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

  ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```

:t `foldr :: (.............) -> tv -> [..] -> tv`

List-recursive – summary

- `foldr`: right recursion

  ```
  []--->v
  (h:t)--->(f h ...)
  ```
  where `...` stands for a recursive call

  ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```

```
:t foldr :: (e -> tv -> tv) -> tv -> [..] -> tv
```

- foldr: right recursion

```
[]--->v
(h:t)--->(f h ...)
```

where ... stands for a recursive call

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

```
:t foldr :: (e -> tv -> tv) -> tv -> [e] -> tv
```

List-recursive – summary

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion

  `[]--->v` accumulator

  `(h:t)--->... (f v h) ...` tail-recursion

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion
  `[]--->v` accumulator
  `(h:t)--->... (f v h) ...` tail-recursion

```
foldl f v [] = v
foldl f v (h:t) = foldl f (f v h) t
```

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call
  `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`

:t `foldr ::` `(e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion
  `[]--->v` accumulator
  `(h:t)--->... (f v h) ...` tail-recursion
  `foldl f v [] = v`
  `foldl f v (h:t) = foldl f (f v h) t`

:t `foldl ::`

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call
  `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion
  `[]--->v` accumulator
  `(h:t)--->... (f v h) ...` tail-recursion
  `foldl f v [] = v`
  `foldl f v (h:t) = foldl f (f v h) t`

:t `foldl :: (............) -> tv -> [..] -> tv`

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

`:t foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion

  `[]--->v` accumulator

  `(h:t)--->... (f v h) ...` tail-recursion

```
foldl f v [] = v
foldl f v (h:t) = foldl f (f v h) t
```

`:t foldl :: (............) -> tv -> [e] -> tv`

List-recursive – summary

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

  ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion

  `[]--->v` accumulator

  `(h:t)--->... (f v h) ...` tail-recursion

  ```
  foldl f v [] = v
  foldl f v (h:t) = foldl f (f v h) t
  ```

:t `foldl :: (tv -> e -> tv) -> tv -> [e] -> tv`

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call
  `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion
  `[]--->v` accumulator
  `(h:t)--->... (f v h) ...` tail-recursion
  `foldl f v [] = v`
  `foldl f v (h:t) = foldl f (f v h) t`

:t `foldl :: (tv -> e -> tv) -> tv -> [e] -> tv`

- `foldr` and `foldl` "mirror" each other:
  $\text{foldr} \circ \text{b}\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ b)...)))$

List-recursive – summary

- `foldr`: right recursion
  `[]--->v`
  `(h:t)--->(f h ...)` where `...` stands for a recursive call
  `foldr f v [] = v`
  `foldr f v (h:t) = f h (foldr f v t)`

```
:t foldr :: (e -> tv -> tv) -> tv -> [e] -> tv
```

- `foldl`: left recursion
  `[]--->v` accumulator
  `(h:t)--->... (f v h) ...` tail-recursion
  `foldl f v [] = v`
  `foldl f v (h:t) = foldl f (f v h) t`

```
:t foldl :: (tv -> e -> tv) -> tv -> [e] -> tv
```

- `foldr` and `foldl` "mirror" each other:
  $\text{foldr} \circ \text{b}\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ b)...)))$
  $\text{foldl} \circ \text{b}\ [x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

List-recursive – summary

- `foldr`: right recursion

  `[]--->v`

  `(h:t)--->(f h ...)` where `...` stands for a recursive call

  ```
  foldr f v [] = v
  foldr f v (h:t) = f h (foldr f v t)
  ```

:t `foldr :: (e -> tv -> tv) -> tv -> [e] -> tv`

- `foldl`: left recursion

  `[]--->v` accumulator

  `(h:t)--->... (f v h) ...` tail-recursion

  ```
  foldl f v [] = v
  foldl f v (h:t) = foldl f (f v h) t
  ```

:t `foldl :: (tv -> e -> tv) -> tv -> [e] -> tv`

- `foldr` and `foldl` "mirror" each other:

  $\text{foldr} \circ \text{b}\ [x_1, x_2, x_3, ..., x_k] = x_1 \circ (x_2 \circ (x_3 \circ (...(x_k \circ b)...)))$

  $\text{foldl} \circ \text{b}\ [x_1, x_2, x_3, ..., x_k] = (...(((b \circ x_1) \circ x_2) \circ x_3) \circ ...) \circ x_k$

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` $= 1 - (2 - (3 - (4 - 0)))$

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, \ldots, x_k] = x_1 \circ (x_2 \circ (\ldots(x_k \circ b)\ldots))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, \ldots, x_k] = (\ldots((a \circ x_1) \circ x_2) \circ \ldots) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl ∘ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```
foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```
foldl ∘ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl ∘ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3])

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] [])))

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3])

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2]

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1]

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```
foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```
foldl ∘ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] . . . . = reverse

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] . . . . = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] .... = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] . . . . = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3] = [1,2,3]`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] . . . . = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3] = [1,2,3]`
- `foldl (\xs -> \x -> x:xs) [] [1,2,3] =`

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] . . . . = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3] = [1,2,3]`
- `foldl (\xs -> \x -> x:xs) [] [1,2,3]` =

What do these do?

```
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldl f a [] = a
foldl f a (h:t) = foldl f (f a h) t
```

foldl $\circ$ a $[x_1, x_2, ..., x_k] = (...((a \circ x_1) \circ x_2) \circ ...) \circ x_k$

- `foldr (-) 0 [1,2,3,4]` = 1 - (2 - (3 - (4 - 0))) = -2
- `foldl (-) 0 [1,2,3,4]` = (((0 - 1) - 2) - 3) - 4 = -10
- `foldl (*) 1 [1,2,3,4]` = 24 = `foldr (*) 1 [1,2,3,4]`
- `foldr (\_ -> (1+)) 0 [1,2,3,4]` = length
- `foldr (\x -> \xs -> xs++[x]) [] [1,2,3]` =
  g 1 (foldr g [] [2,3]) = g 1 (g 2 (foldr g [] [3]))
  = g 1 (g 2 (g 3 (foldr g [] []))) = g 1 (g 2 (g 3 []))
  = g 1 (g 2 [3]) = g 1 [3,2] = [3,2,1] .... = reverse
- `foldl (\xs -> \x -> xs++[x]) [] [1,2,3] = [1,2,3]`
- `foldl (\xs -> \x -> x:xs) [] [1,2,3]` = reverse (tail-recur.)

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr :: (e -> tv -> tv) -> tv -> [e] -> tv
```

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs $==$

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

  map1 f li =

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

  map1 f li = foldr (\x -> \xs -> (f x):xs) [] li

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

  map1 f li = foldr (\x -> \xs -> (f x):xs) [] li
  map2 f = foldr (\x -> (f x:)) []

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
```

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

map1	f li	=	foldr (\x -> \xs -> (f x):xs) [] li
map2	f	=	foldr (\x -> (f x:)) []
map3	f	=	foldr ((:) . f) []
map4		=	\f -> foldr ((:) . f) []

- and `filter`:

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

- and `filter`:

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

$\text{foldr} \circ \text{b}\ [x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

map1	f li	=	foldr (\x -> \xs -> (f x):xs) [] li
map2	f	=	foldr (\x -> (f x:)) []
map3	f	=	foldr ((:) . f) []
map4		=	\f -> foldr ((:) . f) []

- and `filter`:

filter p l =

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

- and `filter`:

filter p l = foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) [] l

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
```

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1   f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2   f     =  foldr (\x -> (f x:) ) []
map3   f     =  foldr ((:) . f) []
map4         =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2  p    =  foldr (\x -> if p x then (x:) . id else id) []
```

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2  p    =  foldr (\x -> if p x then (x:) . id else id) []
filter2  p    =  foldr (\x -> if p x then (x:) else id) []
```

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

$\text{foldr} \circ \text{b} \; [x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

map1	f li	=	foldr (\x -> \xs -> (f x):xs) [] li
map2	f	=	foldr (\x -> (f x:)) []
map3	f	=	foldr ((:) . f) []
map4		=	\f -> foldr ((:) . f) []

- and `filter`:

filter	p l	=	foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) [] l
filter1	p	=	foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) []
filter2	p	=	foldr (\x -> **if** p x **then** (x:) . id **else** id) []
filter2	p	=	foldr (\x -> **if** p x **then** (x:) **else** id) []

- Why not `foldl`::

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1   f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2   f     =  foldr (\x -> (f x:) ) []
map3   f     =  foldr ((:) . f) []
map4         =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2  p    =  foldr (\x -> if p x then (x:) . id else id) []
filter2  p    =  foldr (\x -> if p x then (x:) else id) []
```

- Why not `foldl`::

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1   f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2   f     =  foldr (\x -> (f x:) ) []
map3   f     =  foldr ((:) . f) []
map4         =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter    p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1   p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2   p    =  foldr (\x -> if p x then (x:) . id else id) []
filter2   p    =  foldr (\x -> if p x then (x:) else id) []
```

- Why not `foldl::(v->e->v) -> v->[e]->v`

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2  f     =  foldr (\x -> (f x:) ) []
map3  f     =  foldr ((:) . f) []
map4        =  \f -> foldr ((:) . f) []
```

- and `filter`:

filter p l = foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) [] l
filter1 p = foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) []
filter2 p = foldr (\x -> **if** p x **then** (x:) . id **else** id) []
filter2 p = foldr (\x -> **if** p x **then** (x:) **else** id) []

- Why not `foldl`:: `([e]->e->[e]) -> [e]->[e]->[e]`

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

  map1 f li = foldr (\x -> \xs -> (f x):xs) [] li
  map2 f = foldr (\x -> (f x:)) []
  map3 f = foldr ((:) . f) []
  map4 = \f -> foldr ((:) . f) []

- and `filter`:

  filter p l = foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) [] l
  filter1 p = foldr (\x -> \xs -> **if** p x **then** x:xs **else** xs) []
  filter2 p = foldr (\x -> **if** p x **then** (x:) . id **else** id) []
  filter2 p = foldr (\x -> **if** p x **then** (x:) **else** id) []

- Why not `foldl`:: `([e]->e->[e]) -> [e]->[e]->[e]`
  foldl (:) [] $[x_1, x_2, x_3, ..., x_k]$ ==

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

  ```
  map1  f li  =  foldr (\x -> \xs -> (f x):xs) [] li
  map2  f     =  foldr (\x -> (f x:) ) []
  map3  f     =  foldr ((:) . f) []
  map4        =  \f -> foldr ((:) . f) []
  ```

- and `filter`:

  ```
  filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
  filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
  filter2  p    =  foldr (\x -> if p x then (x:) . id else id) []
  filter2  p    =  foldr (\x -> if p x then (x:) else id) []
  ```

- Why not `foldl`:: `([e]->e->[e]) -> [e]->[e]->[e]`

  foldl (:) [] $[x_1, x_2, x_3, ..., x_k] == (...(([] : x_1) : x_2)...) : x_k$

more `foldr`

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr ∘ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1  f li = foldr (\x -> \xs -> (f x):xs) [] li
map2  f    = foldr (\x -> (f x:) ) []
map3  f    = foldr ((:) . f) []
map4       = \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l = foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p   = foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2  p   = foldr (\x -> if p x then (x:) . id else id) []
filter2  p   = foldr (\x -> if p x then (x:) else id) []
```

- Why not `foldl`:: `([e]->e->[e]) -> [e]->[e]->[e]`

  foldl (:) [] $[x_1, x_2, x_3, ..., x_k] == (...(([] : x_1) : x_2)...) : x_k$

more foldr

```
foldr f v [] = v
foldr f v (h:t) = f h (foldr f v t)
```

foldr $\circ$ b $[x_1, x_2, ..., x_k] = x_1 \circ (x_2 \circ (...(x_k \circ b)...))$

```
foldr ::  (e -> [e] -> [e]) -> [e] -> [e] -> [e]
```

foldr (:) [] xs == xs

- define `map` by using `foldr`:

```
map1   f li  =  foldr (\x -> \xs -> (f x):xs) [] li
map2   f     =  foldr (\x -> (f x:) ) []
map3   f     =  foldr ((:) . f) []
map4         =  \f -> foldr ((:) . f) []
```

- and `filter`:

```
filter   p l  =  foldr (\x -> \xs -> if p x then x:xs else xs) [] l
filter1  p    =  foldr (\x -> \xs -> if p x then x:xs else xs) []
filter2  p    =  foldr (\x -> if p x then (x:) . id else id) []
filter2  p    =  foldr (\x -> if p x then (x:) else id) []
```

- Why not `foldl`:: `([e]->e->[e]) -> [e]->[e]->[e]`

  foldl (:) [] $[x_1, x_2, x_3, ..., x_k] == (...(([] : x_1) : x_2)...) : x_k$

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id))

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h)

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))
- :t compose

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))
- :t compose :: `[t -> t] -> t -> t`

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))
- :t compose :: `[t -> t] -> t -> t`
- function composition associates to the right

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))
- :t compose :: `[t -> t] -> t -> t`
- function composition associates to the right

i.e., (f . g . h) x = f(g(h(x)))

Function composition

```
foldr fu v [] = v
foldr fu v (h:t) = fu h (foldr fu v t)
```

- compose = foldr (.) id
- foldr . id $[f_1, f_2, f_3, ..., f_k] = f_1.(f_2.(f_3.(...(f_k.id)...)))$
- compose [f,g,h] = foldr (.) [f,g,h]
  = (.) f (foldr (.) id [g,h])
  = (.) f ((.) g (foldr (.) id [h]))
  = (.) f ((.) g ((.) h (foldr (.) id [])))
  = (.) f ((.) g ((.) h id)) = (.) f ((.) g h)
  = f . (g . h) = \x -> f (g (h x))
- :t compose :: `[t -> t] -> t -> t`
- function composition associates to the right

i.e., (f . g . h) x = f(g(h(x)))

but since it is associative, this is not important to the result, i.e.,.
$\forall x : (f_1.(f_2.f_3))x = ((f_1.f_2).f_3)x$

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

f1 = map (\ x -> negate (abs x))

f2 = map (negate . abs)

- convert all number into positive and calculate the square root of the successor of the result

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Note! . associates to right: (f . g . h) x = f (g (h (x)))

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Note! . associates to right: (f . g . h) x = f (g (h (x)))

```
f x = ceiling (negate (tan (cos  (max 30 x))))
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Note! . associates to right: (f . g . h) x = f (g (h (x)))

```
f x = ceiling (negate (tan (cos  (max 30 x))))
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Note! . associates to right: (f . g . h) x = f (g (h (x)))

```
f x = ceiling . negate . tan . cos (max 30 x)
```

Function composition . – for example

- given a list of numbers, covert of them into negative
  [1, 3, -6, -7, 8] -> [-1, -3, -6, -7, 8]

```
f1 = map (\ x -> negate (abs x))
f2 = map (negate . abs)
```

- convert all number into positive and calculate the square root of the successor of the result

```
g1 = map (\ x -> sqrt ((abs x)+1))
g2 = map (sqrt . (+1) . abs)
```

Note! . associates to right: (f . g . h) x = f (g (h (x)))

```
f x = ceiling . negate . tan . cos (max 30 x)
  f = ceiling . negate . tan . cos . max 30
```

Note! Function application associate to the left!

- a b c x = ((a b) c) x

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

! ERROR

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument

- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 $\rightarrow$ type error

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument

- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 $\rightarrow$ type error

but :t (+) (+2) 1 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument

- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 → type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 → type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

- (+) ((+2) 1) 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 $\rightarrow$ type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

- (+) ((+2) 1) 7

= ((+) ((+2) 1)) 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 $\rightarrow$ type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

- (+) ((+2) 1) 7

= ((+) ((+2) 1)) 7

= ((+) (3)) 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument
- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 → type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

- (+) ((+2) 1) 7

= ((+) ((+2) 1)) 7

= ((+) (3)) 7

= (+3) 7

Note! Function application associate to the left!

- a b c x = ((a b) c) x

  a :: tb -> tc -> tx

  'curried' functions takes only one argument

- (+) (+2) 1 7

! ERROR

why? ((+) (+2)) 1 7 $\rightarrow$ type error

but :t (+) (+2) 1 7

gives (Num (a -> a), Num a) => a .....?????

- (+) ((+2) 1) 7

= ((+) ((+2) 1)) 7

= ((+) (3)) 7

= (+3) 7

= 10

“Strict” function application – $

- let us write even more parentheses

"Strict" function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

“Strict” function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

“Strict” function application – $

- let us write even more parentheses

```
f = sum ( filter (>10) (map (*2) [1..10]) )
```

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

```
f = sum $ filter (>10) (map (*2) [1..10])
```

“Strict” function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

"Strict" function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

f = sum $ filter (>10) $ map (*2) [1..10]

“Strict” function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

f = sum $ filter (>10) $ map (*2) [1..10]

$ allows to use a value as if it was a function

“Strict” function application – $

- let us write even more parentheses

```
f = sum ( filter (>10) (map (*2) [1..10]) )
```

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

```
f = sum $ filter (>10) (map (*2) [1..10])
```

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

```
f = sum $ filter (>10) $ map (*2) [1..10]
```

$ allows to use a value as if it was a function

section ($ 9) can be sent as function argument

"Strict" function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

f = sum $ filter (>10) $ map (*2) [1..10]

$ allows to use a value as if it was a function

section ($ 9) can be sent as function argument

map ($ 9) [(+2), sqrt] = [11.0, 3.0]

"Strict" function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

f = sum $ filter (>10) $ map (*2) [1..10]

$ allows to use a value as if it was a function

section ($ 9) can be sent as function argument

map ($ 9) [(+2), sqrt] = [11.0, 3.0]

($) :: (a -> b) -> a -> b, so

“Strict” function application – $

- let us write even more parentheses

f = sum (filter (>10) (map (*2) [1..10]))

- exp1 $ exp2 – force to use exp2 as a parameter of exp1

f = sum $ filter (>10) (map (*2) [1..10])

- $ associates to the right, i.e., f $ g $ h x = f $ (g $ (h x))

f = sum $ filter (>10) $ map (*2) [1..10]

$ allows to use a value as if it was a function

section ($ 9) can be sent as function argument

map ($ 9) [(+2), sqrt] = [11.0, 3.0]

($) :: (a -> b) -> a -> b, so

($ 9) :: (a -> b) -> b, is a function

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x,

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x,

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x,

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x),

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)),

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ?

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ?

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
  can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as “lazy” evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:

iterate :: (a -> a) -> a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt 'mod' 2) : debi(dt 'div' 2)
  dec2bin dt = reverse(debi dt)

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt 'mod' 2) : debi(dt 'div' 2)
  dec2bin dt = reverse(debi dt)
- repeat x = [x, x, x, x, x, ...]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt 'mod' 2) : debi(dt 'div' 2)
  dec2bin dt = reverse(debi dt)
- repeat x = [x, x, x, x, x, ...]

e.g. fill in a list l with a value x(=0) inclusive n(=8) position

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt 'mod' 2) : debi(dt 'div' 2)
  dec2bin dt = reverse(debi dt)
- repeat x = [x, x, x, x, x, ...]

e.g. fill in a list l with a value x(=0) inclusive n(=8) position

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt ‘mod‘ 2) : debi(dt ‘div‘ 2)
  dec2bin dt = reverse(debi dt)
- repeat x = [x, x, x, x, x, ...]

e.g. fill in a list l with a value x(=0) inclusive n(=8) position
fill l x n = take n (l ++ repeat x)

iterate :: (a -> a) -> a -> [a]   repeat :: a -> [a]

- iterate f x = [x, f x, f (f x), f (f (f x)), ...] – infinite list?
  iterate (1+) 1 ? can be used the number as "lazy" evaluation

e.g. binary number written as a list: [1,0,1,1]
can converted to decimal number $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

- iterate (2*) 1 = [1, 2, 2^2, 2^3, 2^4, ...]
- bin2dec bit = sum [w*b | (w,b) <- zip weight (reverse bit)]
  where weight = iterate (2*) 1

foldl b2d bit = foldl (\s -> \h -> (2*s)+h) 0 bit

- decimal to binary:
  debi 0 = []
  debi dt = (dt ‘mod‘ 2) : debi(dt ‘div‘ 2)
  dec2bin dt = reverse(debi dt)
- repeat x = [x, x, x, x, x, ...]

e.g. fill in a list l with a value x(=0) inclusive n(=8) position
fill l x n = take n (l ++ repeat x)
fill [1,2,3,4] 0 8 = [1,2,3,4,0,0,0,0]

Types and Type-operators

- Syntax for types:
  `T ::= name | [T] | ({T,} T, T) | T->T | (T->T)`
- Lists: `[typeE]`
- Product (Cartesian): `(typeV,typeH)`
- Functions: `typeK -> typeM`

Type classes

- class Eq t where   (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)
instance Eq Tree where

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

```
instance                Eq  Tree    where
              Leaf a == Leaf b   =   a == b
  (Node l1 r1) == (Node l2 r2)   =   (l1==l2) && (r1==r2)
                        _ == _   =   False
```

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

```
instance (Eq t) => Eq (Tree t) where
            Leaf a == Leaf b    =   a == b
  (Node l1 r1) == (Node l2 r2)  =   (l1==l2) && (r1==r2)
                     _ == _     =   False
```

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

```
instance (Eq t) => Eq (Tree t) where
              Leaf a == Leaf b    =   a == b
  (Node l1 r1) == (Node l2 r2)    =   (l1==l2) && (r1==r2)
                        _ == _    =   False
```

- data Maybe t = Nothing | Just t deriving (Eq, Ord)

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

```
instance (Eq t) => Eq (Tree t) where
              Leaf a == Leaf b    =   a == b
  (Node l1 r1) == (Node l2 r2)    =   (l1==l2) && (r1==r2)
                        _ == _    =   False
```

- data Maybe t = Nothing | Just t deriving (Eq, Ord)

```
instance          Eq (Maybe t) where
      Just a == Just b    =   a == b
  Nothing == Nothing      =   True
              _ == _      =   False
```

Type classes

- class Eq t where (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- subclasses: class Eq t => Ord t where
  (>), (<), (>=), (<=) :: t -> t -> Bool
  x > y = y < x ...
- types declared with `data` can be instances of classes

e.g.: instance Ord Bool where False < True = True

e.g.: data Tree t = Leaf t | Node (Tree t) (Tree t)

instance (Eq t) => Eq (Tree t) where

```
            Leaf a == Leaf b     =  a == b
 (Node l1 r1) == (Node l2 r2)    =  (l1==l2) && (r1==r2)
                       _ == _    =  False
```

- data Maybe t = Nothing | Just t deriving (Eq, Ord)

instance (Eq t) => Eq (Maybe t) where

```
      Just a == Just b    =  a == b
 Nothing == Nothing       =  True
             _ == _       =  False
```

class Functor f where

```
fmap :: (a -> b) -> f a -> f b
```

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor ≃ a traversable collection with an application of a function to all the elements

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor $\simeq$ a traversable collection with an application of a function to all the elements

- type constructor = a data(type) declaration with a single type parameter

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor $\simeq$ a traversable collection with an application of a function to all the elements

- type constructor = a data(type) declaration with a single type parameter
- data Maybe t = Nothing | Just t

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor $\simeq$ a traversable collection with an application of a function to all the elements

- type constructor = a data(type) declaration with a single type parameter
- data Maybe t = Nothing | Just t
  instance Functor Maybe where

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor $\simeq$ a traversable collection with an application of a function to all the elements

- type constructor = a data(type) declaration with a single type parameter
- data Maybe t = Nothing | Just t
  instance Functor Maybe where
  fmap g (Just x) = Just (g x)

class Functor f where

fmap :: (a -> b) -> f a -> f b

- a,b :: type
  f :: type -> type
  f is not a type but is a type constructor
- map :: (a -> b) -> [a] -> [b]
  instance Functor [ ] where fmap = map

not instance Functor [t] where fmap = map

Functor $\simeq$ a traversable collection with an application of a function to all the elements

- type constructor = a data(type) declaration with a single type parameter
- data Maybe t = Nothing | Just t
  instance Functor Maybe where
    fmap g (Just x) = Just (g x)
    fmap g Nothing = Nothing

class Functor f where

```
fmap :: (a -> b) -> f a -> f b
```

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where –fmap:: (t->r) -> [t] -> [r]

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where
  fmap f [] = []

–fmap:: (t->r) -> [t] -> [r]

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where –fmap:: (t->r) -> [t] -> [r]
    fmap f [] = []
    fmap f (x:xs) = (f x) (fmap f xs)

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where —fmap:: (t->r) -> [t] -> [r]
    fmap f [] = []
    fmap f (x:xs) = (f x) (fmap f xs)
- data Tree t = Leaf t | Node (Tree t) (Tree t)

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where –fmap:: (t->r) -> [t] -> [r]
  fmap f [] = []
  fmap f (x:xs) = (f x) (fmap f xs)

- data Tree t = Leaf t | Node (Tree t) (Tree t)
  instance Functor Tree where –fmap:: (t->r) -> Tree t -> Tree r

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where –fmap:: (t->r) -> [t] -> [r]
    fmap f [] = []
    fmap f (x:xs) = (f x) (fmap f xs)

- data Tree t = Leaf t | Node (Tree t) (Tree t)
  instance Functor Tree where –fmap:: (t->r) -> Tree t -> Tree r
    fmap f (Leaf x) = Leaf (f x)

class Functor f where

fmap :: (a -> b) -> f a -> f b

- data List t = [] | (:) t (List t)
  instance Functor List where   –fmap:: (t->r) -> [t] -> [r]
    fmap f [] = []
    fmap f (x:xs) = (f x) (fmap f xs)

- data Tree t = Leaf t | Node (Tree t) (Tree t)
  instance Functor Tree where   –fmap:: (t->r) -> Tree t -> Tree r
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node l r) = Node (fmap f l) (fmap f r)

Kinds $\simeq$ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b
  a,b :: type
  f :: type -> type

Kinds $\simeq$ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b
  a,b :: type
  f :: type -> type

i.e. f is a type constructor

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

>:k Int

Kinds $\simeq$ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

>:k Int
Int :: *

Kinds $\simeq$ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

>:k Int
  Int :: *

>:k Maybe

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b　　　　i GHCi
  a,b :: type　　　　a,b :: *
  f :: type -> type　　　　f :: * -> *

i.e. f is a type constructor

>:k Int
Int :: *

>:k Maybe
Maybe :: * -> *

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

>:k Int
Int :: *
>:k Maybe
Maybe :: * -> *
>:k Maybe Int

Kinds $\simeq$ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

```
>:k Int
    Int :: *
>:k Maybe
    Maybe :: * -> *
>:k Maybe Int
    Maybe Int :: *
```

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

```
>:k Int
    Int :: *
>:k Maybe
    Maybe :: * -> *
>:k Maybe Int
    Maybe Int :: *
>:k Tree
    Tree :: * -> *
```

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

>:k Int
Int :: *

>:k Maybe
Maybe :: * -> *

>:k Maybe Int
Maybe Int :: *

>:k Tree
Tree :: * -> *

>:k Tree Char
Tree Char :: *

Kinds ≃ "types of types"

- class Functor f where
  fmap :: (a -> b) -> f a -> f b — i GHCi
  a,b :: type — a,b :: *
  f :: type -> type — f :: * -> *

i.e. f is a type constructor

```
>:k Int
    Int :: *
>:k Maybe
    Maybe :: * -> *
>:k Maybe Int
    Maybe Int :: *
>:k Tree
    Tree :: * -> *
>:k Tree Char
    Tree Char :: *
```

- Kinds is not "visible" in Haskell programs