Programming Paradigms

Lecture 7. Lazy evaluation in Haskell

Test N°7

See Moodle

Outline

- Lazy evaluation
- Logical operators
- Data constructors and laziness
- Lazy lists
- Functions for working with lists
- Left and right folds

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y



f (3 > 2) (length [1, 2, 3]) 0   ====>   ???
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y



f (3 > 2) (length [1, 2, 3]) 0   ====>   0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

```
f (3 > 2) (length [1, 2, 3]) 0   ====>   0
```

How does the evaluation happen?

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Applying function f,
so we go to its definition

```
f (3 > 2) (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Check the first equation

```
f (3 > 2) (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Attempt pattern matching the first argument

```
f (3 > 2) (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Attempt pattern matching the first argument

```
f (3 > 2) (length [1, 2, 3]) 0
```

At this moment we have to evaluate `(3 > 2)` to see if it can be pattern matched with `True`

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Attempt pattern matching the first argument

```
f False (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Pattern matching fails

```
f  False  (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Try second equation

```
f  False  (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```haskell
f :: Bool -> Int -> Int -> Int
f True   x y = x
f False  x y = y
```

Attempt pattern matching the first argument

```haskell
f  False  (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Success,
go to the next argument

```
f  False  (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True   x y = x
f False  x y = y
```

Attempt pattern matching the second argument

```
f False (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```haskell
f :: Bool -> Int -> Int -> Int
f True   x y = x
f False  x y = y
```

Success,
go to the next argument

```haskell
f False (length [1, 2, 3]) 0
```

Matched variables

```haskell
x = (length [1, 2, 3])
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Attempt pattern matching the third argument

```
f False (length [1, 2, 3]) 0
```

Matched variables

```
x = (length [1, 2, 3])
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

Success

```
f False (length [1, 2, 3]) 0
```

Matched variables

```
x = (length [1, 2, 3])
y = 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True   x y = x
f False  x y = y
```

Use the body, substituting arguments for bound variables

```
f  False  (length [1, 2, 3]) 0
```

Matched variables

```
x = (length [1, 2, 3])
y = 0
```

Lazy evaluation in Haskell

```haskell
f :: Bool -> Int -> Int -> Int
f True   x y = x
f False  x y = y
```

Use the body, substituting arguments for bound variables

```haskell
f  False  (length [1, 2, 3]) 0

====> 0
```

Matched variables

```haskell
x = (length [1, 2, 3])

y = 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

This is **not** evaluated, if the first argument is `False`

```
f (3 > 2) (length [1, 2, 3]) 0
```

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

This is **not** evaluated, if the first argument is `False`

```
f (3 > 2) (length [1, 2, 3]) 0
```

This is **always** evaluated, since we need it to determine which equation to use

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

This is **not** evaluated, if the first argument is `False`

```
f (3 > 2) (length [1, 2, 3]) 0
```

`f` is **strict** in its first argument

Lazy evaluation in Haskell

```
f :: Bool -> Int -> Int -> Int
f True  x y = x
f False x y = y
```

This is **not** evaluated!

```
f  False  (length [1, 2, 3]) 0

====> 0
```

Matched variables

```
x = (length [1, 2, 3])

y = 0
```

Lazy evaluation in Haskell

```
f True  x y = x
f False x y = y
```

What is the type of `f`?

Lazy evaluation in Haskell

```
f :: Bool -> a -> a -> a
f True  x y = x
f False x y = y
```

Lazy evaluation in Haskell

```haskell
ifThenElse :: Bool -> a -> a -> a
ifThenElse True  x y = x
ifThenElse False x y = y
```

Lazy evaluation in Haskell

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True  x _ = x
ifThenElse _     _ y = y
```

Lazy evaluation in Haskell

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True   x _ = x
ifThenElse _      _ y = y
```

We do not care about these values!

Lazy evaluation in Haskell: logical operators

```
(&&) :: Bool -> Bool -> Bool
```

Lazy evaluation in Haskell: logical operators

```haskell
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
```

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
False &&& True  = False
True  &&& False = False
True  &&& True  = True
```

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
False &&& True  = False
True  &&& False = False
True  &&& True  = True
```

In which arguments is (&&&) strict?

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
False &&& True  = False
True  &&& False = False
True  &&& True  = True
```

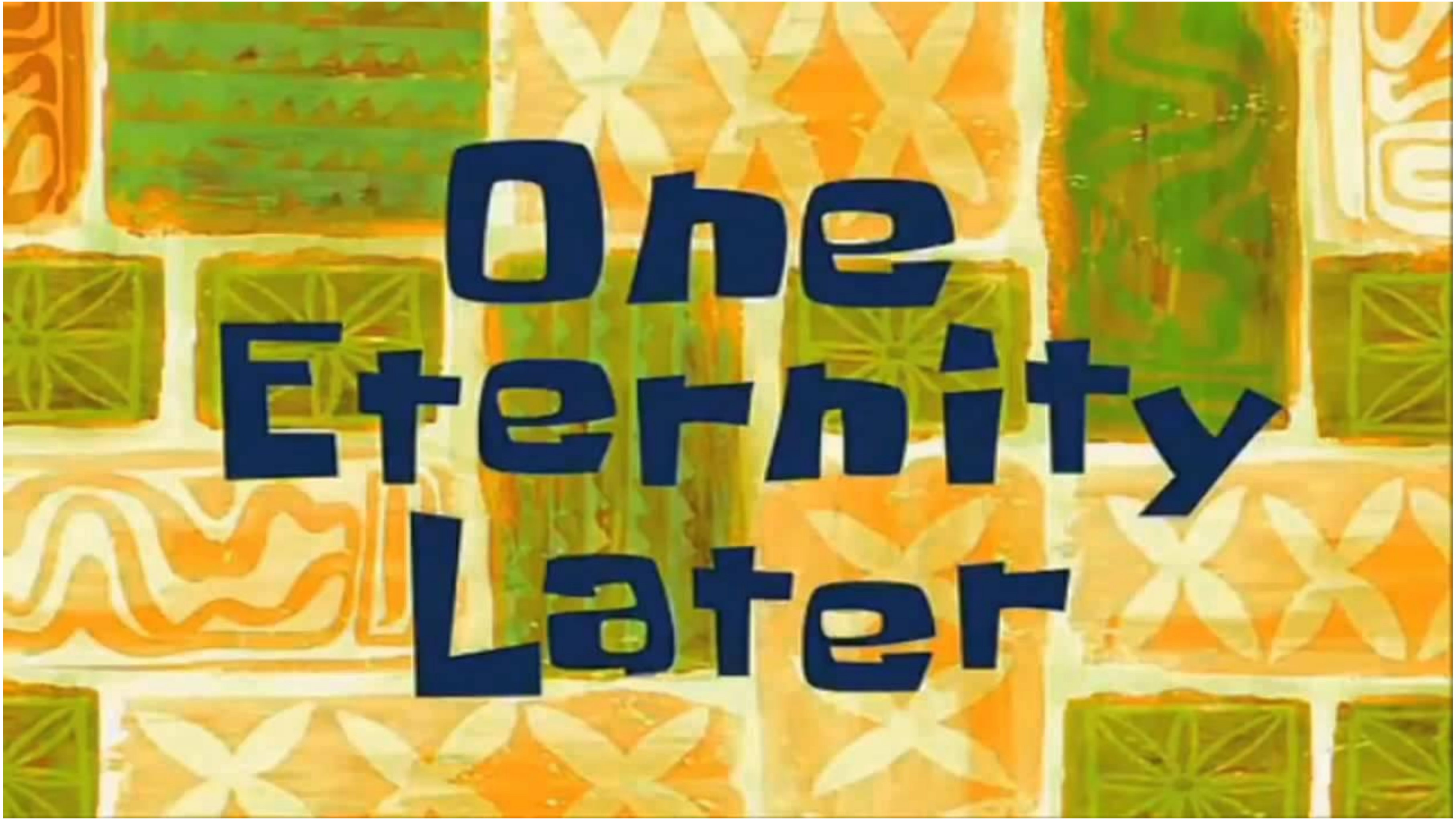
```
(3 > 2) &&& (digitOfPi (10^100) == 3) ====> ???
```

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
False &&& True  = False
True  &&& False = False
True  &&& True  = True
```

```
(3 > 2) &&& (digitOfPi (10^100) == 3) ====> ???
```

An expensive computation!


One
Eternity
Later

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& False = False
False &&& True  = False
True  &&& False = False
True  &&& True  = True


(3 > 2) &&& (digitOfPi (10^100) == 3) ====>
False
```

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& x = False
_     &&& y = y
```

In which arguments is (&&&) strict?

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& x = False
_     &&& y = y
```

```
(3 > 2) &&& (digitOfPi (10^100) == 3) ====> ???
```

Lazy evaluation in Haskell: logical operators

```
(&&&) :: Bool -> Bool -> Bool
False &&& x = False
_     &&& y = y
```

```
(3 > 2) &&& (digitOfPi (10^100) == 3) ====> False
```

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade
```

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade


Student :: Name -> Grade -> Student
```

Data (value) constructors are functions!

Data constructors and laziness

```haskell
type Name = String

type Grade = Int

data Student = Student Name Grade

Student :: Name -> Grade -> Student
```

Data (value) constructors are functions!
Data constructors are **lazy in all arguments**
(unless specified otherwise).

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade


nameOf :: Student -> Name
nameOf (Student name grade) = name
```

Pattern matching against a data constructor
only forces evaluation until the constructor is revealed.
Fields (arguments) of a data constructor
are **not** evaluated (unless there is a nested pattern).

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade


nameOf :: Student -> Name
nameOf (Student name grade) = name


nameOf (Student "Yoko" (digitOfPi (10^100)))
```

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade


nameOf :: Student -> Name
nameOf (Student name grade) = name


nameOf (Student "Yoko" (digitOfPi (10^100)))

====> "Yoko"
```

Data constructors and laziness

```
type Name = String

type Grade = Int

data Student = Student Name Grade

nameOf :: Student -> Name
nameOf (Student name grade) = name
```

This is not evaluated!

```
nameOf (Student "Yoko" (digitOfPi (10^100)))

====> "Yoko"
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
```

This is not evaluated!

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
= 1 + (1 + (1 + length []))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
= 1 + 2
```

Lazy lists: length

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [1, digitOfPi (10^100), 4]
= length (1 : (digitOfPi (10^100) : (4 : [])))
= 1 + length (digitOfPi (10^100) : (4 : []))
= 1 + (1 + length (4 : []))
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
= 1 + 2
= 3
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
= 1 : take 2 [2, 3, 4, 5]
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0     = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
= 1 : take 2 [2, 3, 4, 5]
= 1 : (2 : take 1 [3, 4, 5])
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
= 1 : take 2 [2, 3, 4, 5]
= 1 : (2 : take 1 [3, 4, 5])
= 1 : (2 : (3 : take 0 [3, 4, 5]))
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
= 1 : take 2 [2, 3, 4, 5]
= 1 : (2 : take 1 [3, 4, 5])
= 1 : (2 : (3 : take 0 [3, 4, 5]))
= 1 : (2 : (3 : []))
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 3 [1, 2, 3, 4, 5]
= 1 : take 2 [2, 3, 4, 5]
= 1 : (2 : take 1 [3, 4, 5])
= 1 : (2 : (3 : take 0 [3, 4, 5]))
= 1 : (2 : (3 : []))
= [1, 2, 3]
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 1 (take 3 [1, 2, 3, 4, 5])
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 1 (take 3 [1, 2, 3, 4, 5])
= take 1 (1 : take 2 [2, 3, 4, 5])
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 1 (take 3 [1, 2, 3, 4, 5])
= take 1 (1 : take 2 [2, 3, 4, 5])
= 1 : (take 0 (take 2 [2, 3, 4, 5]))
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 1 (take 3 [1, 2, 3, 4, 5])
= take 1 (1 : take 2 [2, 3, 4, 5])
= 1 : (take 0 (take 2 [2, 3, 4, 5]))
= 1 : []
```

Lazy lists: take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n - 1) xs

take 1 (take 3 [1, 2, 3, 4, 5])
= take 1 (1 : take 2 [2, 3, 4, 5])
= 1 : (take 0 (take 2 [2, 3, 4, 5]))
= 1 : []
= [1]
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
= 1 : 2 : fromTo 3 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
= 1 : 2 : fromTo 3 5
= 1 : 2 : 3 : fromTo 4 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
= 1 : 2 : fromTo 3 5
= 1 : 2 : 3 : fromTo 4 5
= 1 : 2 : 3 : 4 : fromTo 5 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
= 1 : 2 : fromTo 3 5
= 1 : 2 : 3 : fromTo 4 5
= 1 : 2 : 3 : 4 : fromTo 5 5
= 1 : 2 : 3 : 4 : 5 : fromTo 6 5
```

Generating a range

```
fromTo :: Int -> Int -> [Int]
fromTo from to
  | from > to = []
  | otherwise = from : fromTo (from + 1) to

fromTo 1 5
= 1 : fromTo 2 5
= 1 : 2 : fromTo 3 5
= 1 : 2 : 3 : fromTo 4 5
= 1 : 2 : 3 : 4 : fromTo 5 5
= 1 : 2 : 3 : 4 : 5 : fromTo 6 5
= 1 : 2 : 3 : 4 : 5 : []
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
= 1 : 2 : take 1 (from 3)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
= 1 : 2 : take 1 (from 3)
= 1 : 2 : take 1 (3 : from 4)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
= 1 : 2 : take 1 (from 3)
= 1 : 2 : take 1 (3 : from 4)
= 1 : 2 : 3 : take 0 (from 4)
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
= 1 : 2 : take 1 (from 3)
= 1 : 2 : take 1 (3 : from 4)
= 1 : 2 : 3 : take 0 (from 4)
= 1 : 2 : 3 : []
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

take 3 (from 1)
= take 3 (1 : from 2)
= 1 : take 2 (from 2)
= 1 : take 2 (2 : from 3)
= 1 : 2 : take 1 (from 3)
= 1 : 2 : take 1 (3 : from 4)
= 1 : 2 : 3 : take 0 (from 4)
= 1 : 2 : 3 : []
= [1, 2, 3]
```

Infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)

fromTo :: Int -> Int -> [Int]
fromTo start end = take n (from start)
  where n = end - start + 1
```

Range list notation

```
[start..end]
```

```
[1..5] = [1, 2, 3, 4, 5]
```

Range list notation

```
[start..end]                [1..5] = [1, 2, 3, 4, 5]
[start..]                   [1..]  = [1, 2, 3, …]
```

Range list notation

```
[start..end]              [1..5] = [1, 2, 3, 4, 5]
[start..]                 [1..]  = [1, 2, 3, …]

[start, next .. end]      [1, 3 .. 10] = [1, 3, 5, 7, 9]
```

Range list notation

```
[start..end]              [1..5] = [1, 2, 3, 4, 5]
[start..]                 [1..]  = [1, 2, 3, …]

[start, next .. end]      [1, 3 .. 10] = [1, 3, 5, 7, 9]
[start, next ..]          [1, 4] = [1, 4, 7, 10, …]
```

Functions on lists

```
f :: [a] -> a
```

What is “off” about this function?

Functions on lists

```
f :: [a] -> a
```

What is “off” about this function?

This function is guaranteed to be partial!

Functions on lists

```
head :: [a] -> a

tail :: [a] -> a

(!!) :: [a] -> Int -> a
```

What is “off” about this functions?

Functions on lists

```
head :: [a] -> a

tail :: [a] -> a

(!!) :: [a] -> Int -> a
```

You should **NEVER** use these!

Use pattern matching instead!

Functions on lists

```haskell
head :: [a] -> a

tail :: [a] -> a

(!!) :: [a] -> Int -> a
```

You should **NEVER** use these!

Use pattern matching instead!

```haskell
sum :: [Int] -> Int
sum xs
  | null xs   = 0
  | otherwise = head xs + sum (tail xs)
```

Bad, fragile code!

Functions on lists

```
head :: [a] -> a

tail :: [a] -> a

(!!) :: [a] -> Int -> a
```

You should **NEVER** use these!

Use pattern matching instead!

```
sum :: [Int] -> Int
sum xs = case xs of
  []    -> 0
  n:ns -> n + sum ns
```

Good and stable!

Functions on lists: useful functions

```haskell
import Data.List

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

Functions on lists: concatenation

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

concat :: [[a]] -> [a]

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

Functions on lists: cutting

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]

takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Functions on lists: reducing functions

```
length  :: [a] -> Int
sum     :: Num a => [a] -> a
product :: Num a => [a] -> a

or  :: [Bool] -> Bool
and :: [Bool] -> Bool

any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool

elem    :: Eq a => a -> [a] -> Bool
notElem :: Eq a => a -> [a] -> Bool
```

Functions on lists: zipping

```
zip :: [a] -> [b] -> [(a, b)]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Left fold

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

Left fold

```
foldl :: (state -> a -> state) -> state -> [a] -> state
```

Left fold

```
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state
```

Left fold

```
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldl (+) z [a, b, c]  =    ???
```

Left fold

```
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldl (+) z [a, b, c]  =    z
```

Left fold

```
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldl (+) z [a, b, c]  =    z + a
```

Left fold

```
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldl (+) z [a, b, c]  =   (z + a) + b
```

Left fold

```haskell
foldl
    :: (state -> a -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldl (+) z [a, b, c]  =  ((z + a) + b) + c
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (+) z [a, b, c]  = ???
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (+) z [a, b, c]  =                 z
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (+) z [a, b, c]  =            c + z
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (+) z [a, b, c]  =        b + (c + z)
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (+) z [a, b, c]  =  a + (b + (c + z))
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (&&) z [False, b, c]
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (&&) z [False, b, c]
= False && (foldr (&&) z [b, c])
```

Right fold

```
foldr
    :: (a -> state -> state)    -- stepping function
    -> state                    -- initial state
    -> [a]                      -- list of values
    -> state

foldr (&&) z [False, b, c]
= False && (foldr (&&) z [b, c])
= False
```

Left fold vs Right fold

	Left fold	Right fold
Lazy	`foldl` should **not** be used*	`foldr` should be used when stepping function is lazy in the accumulator parameter Examples: `or`, `and`, `concat`, `map`, `filter`
Strict	`Data.List.foldl'` should be used when stepping function is strict in accumulator parameter Examples: `sum`, `product`, `length`	—

What was the most unclear part of the lecture for you?

See Moodle