Functional Programming in Haskell

CSCI 3136

Principles of Programming Languages

Faculty of Computer Science

Dalhousie University

Winter 2013

Disclaimer

Everything we will talk about here ***can*** be done in C or even assembly language.

The question is not ***whether*** it can be done but ***how easily*** it can be done.

It's all about expressiveness of the language.

Functional vs. Imperative Programming

Imperative programming:

The program specifies what the computer should ***do***.

Functional programming:

The program specifies what the ***value*** of a function should be.

The exact sequence of steps to compute this value is left unspecified.

This is one form of ***declarative programming***.

Consequences:

- Need mechanisms to specify execution order when necessary
- Code correctness and memoization
- Lazy evaluation
- ...

Examples

C

```
int two() {
  return 2;
}
```

```
int timestwo(int x) {
  return 2*x;
}
```

???

???

Haskell

```
two :: Int
two = 2
```

```
timestwo :: Int -> Int
timestwo x = 2 * x
```

Polymorphism

```
timestwo' :: Num a => a -> a
timestwo' x = 2 * x
```

Currying

```
timestwo'' :: Num a => a -> a
timestwo'' = (*) 2
```

Control Constructs

if-then-else

```
abs :: Int -> Int
abs x = if x < 0 then (-x) else x
```

The else-branch is ***mandatory***. Why?

Control Constructs

if-then-else

```
abs :: Int -> Int
abs x = if x < 0 then (-x) else x
```

The else-branch is ***mandatory***. Why?

case

```
is-two-or-five :: Int -> Bool
is-two-or-five x = case x of
                     2 -> True
                     5 -> True
                     _ -> False
```

_ is a wildcard that matches any value.

Loops?

Loops make no sense in a functional language. Why?

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C

```
int factorial(int n) {
  int fac = 1;
  int i;
  for( i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

Recursive C

```
int factorial(int n) {
  if( n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C

```
int factorial(int n) {
  int fac = 1;
  int i;
  for( i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

Recursive C

```
int factorial(int n) {
  if( n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Haskell

```
factorial :: Int -> Int
factorial x = if x <= 1 then 1 else x * factorial (x - 1)
```

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C

Efficient

```
int factorial(int n) {
  int fac = 1;
  int i;
  for( i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

Recursive C

```
int factorial(int n) {
  if( n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Haskell

```
factorial :: Int -> Int
factorial x = if x <= 1 then 1 else x * factorial (x - 1)
```

Loops?

Loops make no sense in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C Efficient

```
int factorial(int n) {
  int fac = 1;
  int i;
  for( i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

Recursive C Inefficient

```
int factorial(int n) {
  if( n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Haskell Inefficient

```
factorial :: Int -> Int
factorial x = if x <= 1 then 1 else x * factorial (x - 1)
```

Making Recursion Efficient: Tail Recursion

When the last statement in a function is a recursive invocation of the same function, the compiler can convert these recursive calls into a loop.

Not tail recursive

```
factorial :: Int -> Int
factorial x = if x <= 1 then 1 else x * factorial (x - 1)
```

- Stack size = depth of recursion
- Overhead to maintain the stack

Making Recursion Efficient: Tail Recursion

When the last statement in a function is a recursive invocation of the same function, the compiler can convert these recursive calls into a loop.

Not tail recursive

```
factorial :: Int -> Int
factorial x = if x <= 1 then 1 else x * factorial (x - 1)
```

- Stack size = depth of recursion
- Overhead to maintain the stack

Tail recursive

```
factorial :: Int -> Int
factorial x = factorial' x 1

factorial' :: Int -> Int -> Int
factorial' 1 p = p
factorial' x p = factorial' (x-1) (x*p)
```

- Constant stack size
- No overhead to maintain the stack

Patterns

Haskell allows multiple definitions of the same function.

All must have the same type.

It uses the first one that matches the actual parameters.

The formal parameters are patterns that need to be matched by the actual parameters.

```
factorial’ :: Int -> Int -> Int
factorial’ 1 p = p
factorial’ x p = factorial’ (x-1) (x*p)
```

Patterns

Haskell allows multiple definitions of the same function.

All must have the same type.

It uses the first one that matches the actual parameters.

The formal parameters are patterns that need to be matched by the actual parameters.

```
factorial' :: Int -> Int -> Int
factorial' 1 p = p
factorial' x p = factorial' (x-1) (x*p)
```

This is identical to the following single function definition using a case statement.

```
factorial' :: Int -> Int -> Int
factorial' x p = case x of
                   1 -> p
                   _ -> factorial' (x-1) (x*p)
```

Arrays

Haskell does support arrays, but they're slow.

(One way to) create an array:

```
a = listArray (1, 10) [1 .. 10]
```

Access array element in constant time:

```
a ! 4
> 4
```

Arrays

Haskell does support arrays, but they're slow.

(One way to) create an array:

```
a = listArray (1, 10) [1 .. 10]
```

Access array element in constant time:

```
a ! 4
> 4
```

Array update in linear time (!!!):

```
b = a \\ [(4, 8), (6, 9)]
b ! 4
> 8
b ! 6
> 9
b ! 1
> 1
```

Array update creates a copy of the original array with the specified elements changed.

Why?

Lists

To Haskell (Scheme, Lisp, ...), lists are what arrays are to C.

Lists are defined recursively and, thus, match the recursive world view of functional programming:

A list

- Is empty or
- Consists of an element followed by a list

Lists

To Haskell (Scheme, Lisp, . . .), lists are what arrays are to C.

Lists are defined recursively and, thus, match the recursive world view of functional programming:

A list

- Is empty or
- Consists of an element followed by a list

In Haskell:

```
emptyList      = []
oneElementList = 1 : emptyList
twoElementList = 2 : oneElementList
```

Lists

To Haskell (Scheme, Lisp, ...), lists are what arrays are to C.

Lists are defined recursively and, thus, match the recursive world view of functional programming:

A list

- Is empty or
- Consists of an element followed by a list

In Haskell:

```
emptyList      = []
oneElementList = 1 : emptyList
twoElementList = 2 : oneElementList
```

List comprehensions

```
a = [1, 2, 3]
b = [1 .. 10]
c = [1, 3 .. 10]
d = [x | x <- [1 .. 10], odd x]
```

Working with Lists

Decomposing a list

```
a = [1 .. 10]
head a
> 1
tail a
> [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Adding elements

```
1 : [2 .. 10]
> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

List concatenation

```
[1, 2, 3] ++ [4, 5]
> [1, 2, 3, 4, 5]
```

“Iterating” over Lists

Many iterative processes can be expressed as a combination of a few common idioms.

Mapping: Apply a function to each element of a sequence independently

```
a = [1 .. 10]
map (* 2) [1 .. 10]
> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Folding: Accumulate the elements in a list

```
a = [1 .. 10]
foldr (+) 0 a
> 55
```

Filtering: Compute the sublist of all elements that satisfy a certain condition

```
a = [1 .. 10]
filter even a
> [2, 4, 6, 8, 10]
```

Implementing Iteration Constructs

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

Implementing Iteration Constructs

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ b []     = b
foldr f b (a:as) = f a (foldr b as)
```

Implementing Iteration Constructs

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ b []     = b
foldr f b (a:as) = f a (foldr b as)
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs) | p x       = x : (filter p xs)
                | otherwise = filter p xs
```

Functions Are First-Class Values

The name ***functional programming*** comes from the fact that ***functions are first-class values***, the entire focus is on functions:

- Functions can be passed as arguments to functions
- Functions can be returned as the results of function calls
- We can construct new functions from existing ones at runtime
- ...

Pairs and Tuples

Lists have a limitation: all elements must be of the same type.

```
l :: [Int]
l = [1 .. 10]
l' = 'a' : l
Lengthy error message about a type mismatch between Int and Char
```

Pairs and Tuples

Lists have a limitation: all elements must be of the same type.

```
l :: [Int]
l = [1 .. 10]
l' = 'a' : l
Lengthy error message about a type mismatch between Int and Char
```

This problem does not arise in Lisp or Scheme. Why?

Pairs and Tuples

Lists have a limitation: all elements must be of the same type.

```
l :: [Int]
l = [1 .. 10]
l' = 'a' : l
Lengthy error message about a type mismatch between Int and Char
```

This problem does not arise in Lisp or Scheme. Why?

Pairs and tuples allow us to group things of different types.

```
a :: (Int, Char)
a = (1, 'x')
```

```
b :: (Int, Char, String)
b = (2, 'y', "something")
```

Pairs and Tuples

Lists have a limitation: all elements must be of the same type.

```
l :: [Int]
l = [1 .. 10]
l' = 'a' : l
Lengthy error message about a type mismatch between Int and Char
```

This problem does not arise in Lisp or Scheme. Why?

Pairs and tuples allow us to group things of different types.

```
a :: (Int, Char)
a = (1, 'x')
```

```
b :: (Int, Char, String)
b = (2, 'y', "something")
```

Pairs and tuples in turn have a limitation lists do not have: the number of elements is fixed.

Functions for Pairs and Tuples

Pairs

```
fst :: (a, b) -> a
fst (x, _) = x
```

```
snd :: (a, b) -> a
snd (_, x) = x
```

Functions for Pairs and Tuples

Pairs

```
fst :: (a, b) -> a
fst (x, _) = x
```

```
snd :: (a, b) -> a
snd (_, x) = x
```

```
(,) :: a -> b -> (a, b)
(,) x y = (x, y)
```

Functions for Pairs and Tuples

Pairs

```
fst :: (a, b) -> a
fst (x, _) = x
```

```
snd :: (a, b) -> a
snd (_, x) = x
```

```
(,) :: a -> b -> (a, b)
(,) x y = (x, y)
```

Tuples

```
(,,,) :: a -> b -> c -> d -> (a, b, c, d)
(,,,) w x y z = (w, x, y, z)
```

Zipping and Unzipping

Zipping two lists together

```
zip [1, 2, 3] ['a', 'b']
> [(1, 'a'), (2, 'b')]
```

Zipping and Unzipping

Zipping two lists together

```
zip [1, 2, 3] ['a', 'b']
> [(1, 'a'), (2, 'b')]
```

Unzipping a list of pairs

```
unzip [(1, 'a'), (2, 'b')]
> ([1, 2], ['a', 'b'])
```

Zipping and Unzipping

Zipping two lists together

```
zip [1, 2, 3] ['a', 'b']
> [(1, 'a'), (2, 'b')]
```

Unzipping a list of pairs

```
unzip [(1, 'a'), (2, 'b')]
> ([1, 2], ['a', 'b'])
```

Variants

```
zipWith (*) [1, 2, 3] [4, 5, 6]
> [4, 10, 18]

zip3 [1, 2] ['a', 'b'] [[1, 2], [3, 4, 5]]
> [(1, 'a', [1, 2]), (2, 'b', [3, 4, 5])]

...
```

Anonymous Functions

When we write

```
f x y = x * y,
```

this is just syntactic sugar for

```
f = \ x y -> x * y
```

Anonymous Functions

When we write

`f x y = x * y`,

this is just syntactic sugar for

```
f = \ x y -> x * y
```

`\x y -> x * y` is an ***anonymous function***.

Anonymous Functions

When we write

`f x y = x * y`,

this is just syntactic sugar for

```
f = \ x y -> x * y
```

`\x y -> x * y` is an ***anonymous function***.

Mapping over a list

```
swapelems :: [(a, b)]
          -> [(b, a)]
swapelems xs = map swap xs
  where
    swap (x, y) = (y, x)
```

Anonymous Functions

When we write

`f x y = x * y`,

this is just syntactic sugar for

```
f = \ x y -> x * y
```

`\x y -> x * y` is an ***anonymous function***.

Mapping over a list

```
swapelems :: [(a, b)]
          -> [(b, a)]
swapelems xs = map swap xs
  where
    swap (x, y) = (y, x)
```

```
swapelems :: [(a, b)]
          -> [(b, a)]
swapelems xs =
  map (\(x, y) -> (y, x)) xs
```

Anonymous Functions

When we write

`f x y = x * y`,

this is just syntactic sugar for

```
f = \ x y -> x * y
```

`\x y -> x * y` is an ***anonymous function***.

Mapping over a list

```
swapelems :: [(a, b)]
          -> [(b, a)]
swapelems xs = map swap xs
  where
    swap (x, y) = (y, x)
```

```
swapelems :: [(a, b)]
          -> [(b, a)]
swapelems xs =
  map (\(x, y) -> (y, x)) xs
```

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

Huh?

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> b -> c -> d
```

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> b -> c -> d
```

Why not

```
f ::  (a, b, c) -> d?
```

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> b -> c -> d
```

Why not

`f ::  (a, b, c) -> d`?

They're different, but they have one thing in common: neither is **really** a multi-argument function.

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> b -> c -> d
```

Why not

`f ::  (a, b, c) -> d`?

They're different, but they have one thing in common: neither is **really** a multi-argument function.

`f ::  a -> b -> c -> d` is a function with one argument of type `a` and whose result is ...

... a function with one argument of type `b` and whose result is ...

... a function with one argument of type `c` and whose result is of type `d`.

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> (b -> (c -> d))
```

Why not

`f :: (a, b, c) -> d`?

They're different, but they have one thing in common: neither is **really** a multi-argument function.

`f :: a -> b -> c -> d` is a function with one argument of type `a` and whose result is ...

... a function with one argument of type `b` and whose result is ...

... a function with one argument of type `c` and whose result is of type `d`.

Partial Function Application and Currying

Here is how we write multi-argument functions.

`f :: a -> (b -> (c -> d))`

Why not

`f ::  (a, b, c) -> d`?

They're different, but they have one thing in common: neither is **really** a multi-argument function.

`f ::  a -> b -> c -> d` is a function with one argument of type `a` and whose result is …

… a function with one argument of type `b` and whose result is …

… a function with one argument of type `c` and whose result is of type `d`.

`f ::  (a, b, c) -> d` is a function with one argument of type `(a, b, c)` and whose result is a value of type `d`.

Partial Function Application and Currying

Here is how we write multi-argument functions.

```
f :: a -> (b -> (c -> d))
```

Why not

`f ::  (a, b, c) -> d`?

They're different, but they have one thing in common: neither is **really** a multi-argument function.

`f ::  a -> b -> c -> d` is a function with one argument of type `a` and whose result is ...

... a function with one argument of type `b` and whose result is ...

... a function with one argument of type `c` and whose result is of type `d`.

`f ::  (a, b, c) -> d` is a function with one argument of type `(a, b, c)` and whose result is a value of type `d`.

We call `f ::  a -> b -> c -> d` a *curried* function.

Applying Curried Functions

`f x y z` really means `((f x) y) z`, that is,

Apply `f` to `x`.

Apply the resulting function to `y`.

Apply the resulting function to `z`.

And that's the final result … which could happen to be itself a function!

Why Are Curried Functions Better?

Multiplying all elements in a list by two.

Why Are Curried Functions Better?

Multiplying all elements in a list by two.

Without currying:

```
timestwo :: [Int] -> [Int]
timestwo xs = map (\x -> 2 * x) xs
```

Why Are Curried Functions Better?

Multiplying all elements in a list by two.

Without currying:

```
timestwo :: [Int] -> [Int]
timestwo xs = map (\x -> 2 * x) xs
```

With currying (part 1):

`(*)` is itself a function of type

```
(*) :: Num a => a -> a -> a
```

It maps its first argument x to a function f that multiplies its argument y by x.

Why Are Curried Functions Better?

Multiplying all elements in a list by two.

Without currying:

```
timestwo :: [Int] -> [Int]
timestwo xs = map (\x -> 2 * x) xs
```

With currying (part 1):

`(*)` is itself a function of type

```
(*) :: Num a => a -> a -> a
```

It maps its first argument x to a function f that multiplies its argument y by x.

```
timestwo xs = map (* 2) xs
```

Why Are Curried Functions Better?

With currying (part 2):

`map` is a function of type

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

It maps its first argument (a function f) to a function m that applies f to each element in its argument list.

Why Are Curried Functions Better?

With currying (part 2):

`map` is a function of type

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

It maps its first argument (a function f) to a function m that applies f to each element in its argument list.

```
timestwo = map (* 2)
```

Why Are Curried Functions Better?

With currying (part 2):

`map` is a function of type

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

It maps its first argument (a function f) to a function m that applies f to each element in its argument list.

```
timestwo = map (* 2)
```

This is often called ***point-free*** programming. The focus is on building functions from functions rather than specifying the value a function produces on a particular argument.

Function Composition

Point-free programming cannot work without function composition:

```
multiplyevens :: [Int] -> [Int]
multiplyevens xs = map (* 2) (filter even xs)
```

Function Composition

Point-free programming cannot work without function composition:

```
multiplyevens :: [Int] -> [Int]
multiplyevens xs = map (* 2) (filter even xs)
```

Function composition:

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

Function Composition

Point-free programming cannot work without function composition:

```
multiplyevens :: [Int] -> [Int]
multiplyevens xs = map (* 2) (filter even xs)
```

Function composition:

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

```
multiplyevens = map (* 2) . filter even
```

A Few Useful Functions

```
($)     :: (a -> b) -> a -> b               -- f $ x applies f to x
flip    :: (a -> b -> c) -> b -> a -> c     -- Flips the function
                                            -- arguments
curry   :: ((a, b) -> c) -> a -> b -> c     -- Curries a function
                                            -- whose argument is a
                                            -- pair
uncurry :: (a -> b -> c) -> (a, b) -> c     -- Collapses the first
                                            -- two arguments of the
                                            -- given function into
                                            -- a single pair
```

A Few Useful Functions

```
($)     :: (a -> b) -> a -> b                -- f $ x applies f to x
flip    :: (a -> b -> c) -> b -> a -> c      -- Flips the function
                                             -- arguments
curry   :: ((a, b) -> c) -> a -> b -> c      -- Curries a function
                                             -- whose argument is a
                                             -- pair
uncurry :: (a -> b -> c) -> (a, b) -> c      -- Collapses the first
                                             -- two arguments of the
                                             -- given function into
                                             -- a single pair
```

Why the need for an application operator?

Function application binds more tightly than function composition, which binds more tightly than the `($)` operator.

```
f :: a -> b
g :: b -> c
x :: a
g . f $ x :: c
g . f x -- type error
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

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

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

```
flip                    :: (a -> b -> c) -> b -> a -> c
uncurry                 :: (b -> a -> c) -> (b, a) -> c
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

```
flip                    :: (a -> b -> c) -> b -> a -> c
uncurry                 :: (b -> a -> c) -> (b, a) -> c
uncurry . flip          :: (a -> b -> c) -> (b, a) -> c
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

```
flip                 :: (a -> b -> c) -> b -> a -> c
uncurry              :: (b -> a -> c) -> (b, a) -> c
uncurry . flip       :: (a -> b -> c) -> (b, a) -> c
(,)                  :: a -> b -> (a, b)
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

```
flip                 :: (a -> b -> c) -> b -> a -> c
uncurry              :: (b -> a -> c) -> (b, a) -> c
uncurry . flip       :: (a -> b -> c) -> (b, a) -> c
(,)                  :: a -> b -> (a, b)
uncurry . flip $ (,) :: (b, a) -> (a, b)
```

Back to `swapelems`

```
swapelems :: [(a, b)] -> [(b, a)]
swapelems = map (uncurry . flip $ (,))
```

```
flip                   :: (a -> b -> c) -> b -> a -> c
uncurry                :: (b -> a -> c) -> (b, a) -> c
uncurry . flip         :: (a -> b -> c) -> (b, a) -> c
(,)                    :: a -> b -> (a, b)
uncurry . flip $ (,) :: (b, a) -> (a, b)
```

Now try to do this in C, C++, Java, . . . !

Mergesort

The algorithm

```
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort = uncurry merge
          . both mergesort
          . divide
  where both f (x, y) = (f x, f y)

-- Merge two sorted lists
merge :: Ord a => [a] -> [a] -> [a]

-- Distribute n elements into two lists of length n/2
divide :: [a] -> ([a], [a])
```

Mergesort

Merging two lists is easy:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs         []          = xs
merge []         ys          = ys
merge xs@(x:xs') ys@(y:ys') | x < y     = x : merge xs' ys
                            | otherwise = y : merge xs ys'
```

Mergesort

Merging two lists is easy:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs          []          = xs
merge []          ys          = ys
merge xs@(x:xs’) ys@(y:ys’) | x < y      = x : merge xs’ ys
                            | otherwise = y : merge xs ys’
```

Evenly splitting a list without getting its length is a bit trickier:

```
divide :: [a] -> ([a], [a])
divide []       = ([], [])
divide [x]      = ([x], [])
divide (x:y:zs) = (x:xs, y:ys) where (xs, ys) = divide zs
```

Quicksort

Normally we'd use a random pivot, but generating random numbers involves side effects. Why?

So we use the simple strategy: pick the first element as pivot.

Quicksort

Normally we'd use a random pivot, but generating random numbers involves side effects. Why?

So we use the simple strategy: pick the first element as pivot.

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
  where
    (ys, zs) = partition (< x) xs
```

Quicksort

Normally we'd use a random pivot, but generating random numbers involves side effects. Why?

So we use the simple strategy: pick the first element as pivot.

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
  where
    (ys, zs) = partition (< x) xs
```

`partition` is part of the standard library. If it wasn't, we could implement it as follows:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p []     = ([], [])
partition p (x:xs) | p x       = (x:ys, ns)
                   | otherwise = (ys, x:ns)
  where (ys, ns) = partition xs
```

Polymorphism and Type Variables

A function to access the head of a list of integers:

```
head :: [Int] -> Int
head []     = undefined
head (x:_) = x
```

Polymorphism and Type Variables

A function to access the head of a list of integers:

```
head :: [Int] -> Int
head []     = undefined
head (x:_) = x
```

This function cannot be applied to a list of floating point numbers!

Polymorphism and Type Variables

A function to access the head of a list of integers:

```
head :: [Int] -> Int
head []     = undefined
head (x:_) = x
```

This function cannot be applied to a list of floating point numbers!

A function to access the head of a list of floating point numbers:

```
head :: [Double] -> Double
head []     = undefined
head (x:_) = x
```

Polymorphism and Type Variables

A function to access the head of a list of integers:

```
head :: [Int] -> Int
head []     = undefined
head (x:_) = x
```

This function cannot be applied to a list of floating point numbers!

A function to access the head of a list of floating point numbers:

```
head :: [Double] -> Double
head []     = undefined
head (x:_) = x
```

The behaviour is exactly the same, so why do we need two functions?

Polymorphism and Type Variables

A function to access the head of a list of integers:

```
head :: [Int] -> Int
head []     = undefined
head (x:_) = x
```

This function cannot be applied to a list of floating point numbers!

A function to access the head of a list of floating point numbers:

```
head :: [Double] -> Double
head []     = undefined
head (x:_) = x
```

The behaviour is exactly the same, so why do we need two functions?

A variant that works for any type of list elements:

```
head :: [a] -> a
head []     = undefined
head (x:_) = x
```

Type Classes

Quicksort for arbitrary element types — does not work:

```
quicksort :: [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
  where
    (ys, zs) = partition (< x) xs
```

Type Classes

Quicksort for element types that provide an ordering:

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
  where
    (ys, zs) = partition (< x) xs
```

The `Ord` type class:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<)     :: a -> a -> Bool
  (<=)    :: a -> a -> Bool
  (>)     :: a -> a -> Bool
  (>=)    :: a -> a -> Bool
  min     :: a -> a -> a
  max     :: a -> a -> a
```

The `Eq` type class:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Type Classes

Quicksort for element types that provide an ordering:

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
  where
    (ys, zs) = partition (< x) xs
```

The `Ord` type class:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<)     :: a -> a -> Bool
  (<=)    :: a -> a -> Bool
  (>)     :: a -> a -> Bool
  (>=)    :: a -> a -> Bool
  min     :: a -> a -> a
  max     :: a -> a -> a
```

The `Eq` type class:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Note how this is very similar to Java interfaces.

Lazy Evaluation

Assume we write a parser and want to provide line numbers in our error messages. We need to annotate each input line with its line number.

Lazy Evaluation

Assume we write a parser and want to provide line numbers in our error messages. We need to annotate each input line with its line number.

The hard way:

```
splitInput :: String -> [(Int, String)]
splitInput text = zip ns ls
  where
    ls = lines text
    ns = [1 .. length ls]
```

Lazy Evaluation

Assume we write a parser and want to provide line numbers in our error messages. We need to annotate each input line with its line number.

The hard way:

```
splitInput :: String -> [(Int, String)]
splitInput text = zip ns ls
  where
    ls = lines text
    ns = [1 .. length ls]
```

The easy way:

```
splitInput :: String -> [(Int, String)]
splitInput = zip [1 ..] . lines
```

Lazy Evaluation

Assume we write a parser and want to provide line numbers in our error messages. We need to annotate each input line with its line number.

The hard way:

```
splitInput :: String -> [(Int, String)]
splitInput text = zip ns ls
  where
    ls = lines text
    ns = [1 .. length ls]
```

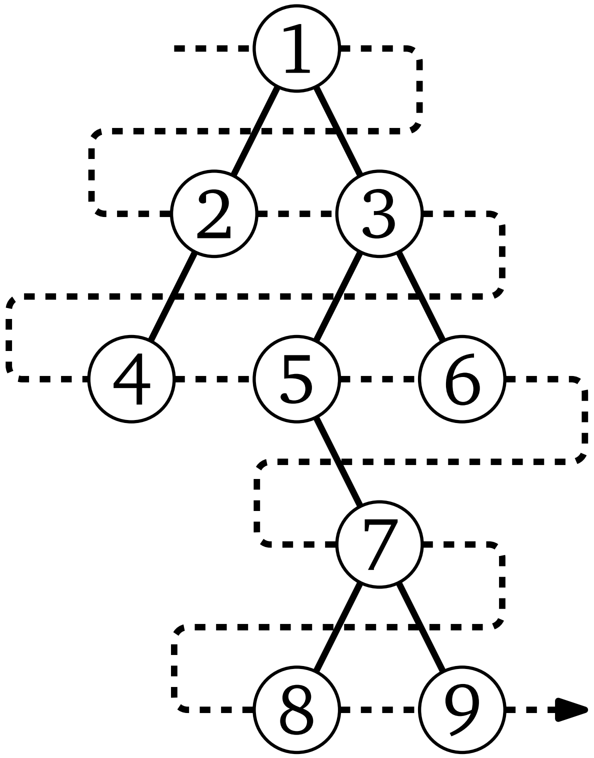
The easy way:

```
splitInput :: String -> [(Int, String)]
splitInput = zip [1 ..] . lines
```

The list of Fibonacci numbers:

```
fibonacci :: [Int]
fibonacci = 1 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

BFS Numbering

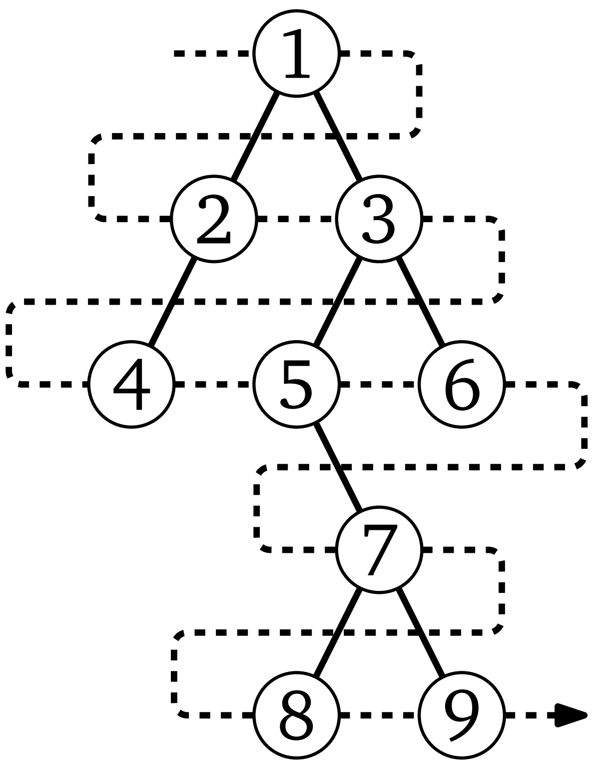


BFS Numbering

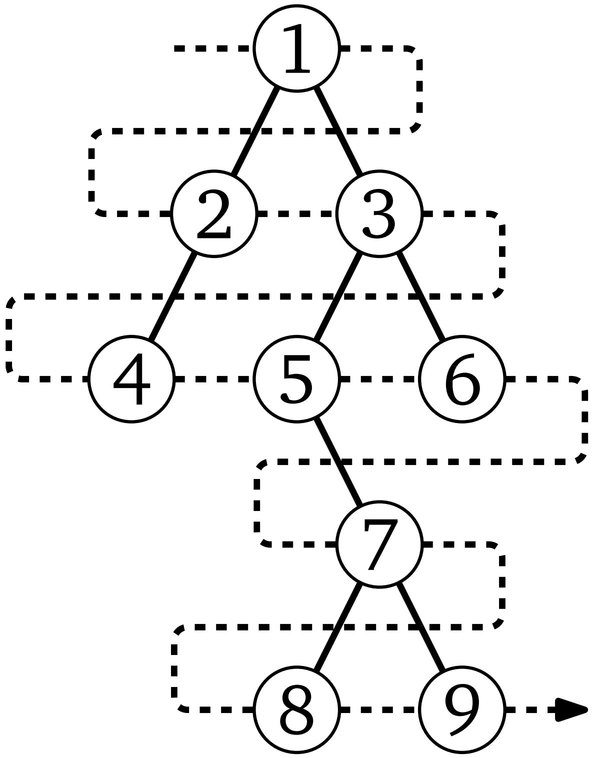
The naïve solution:

- Build a list of the nodes in level order
- Number them in order
- Reassemble the tree

I refuse to turn this into code; it's messy.



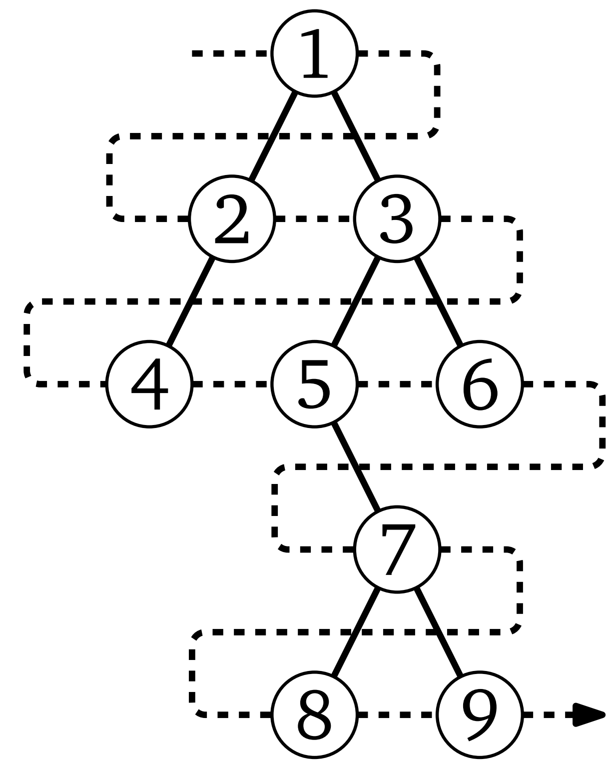
BFS Numbering



The tree type:

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

BFS Numbering



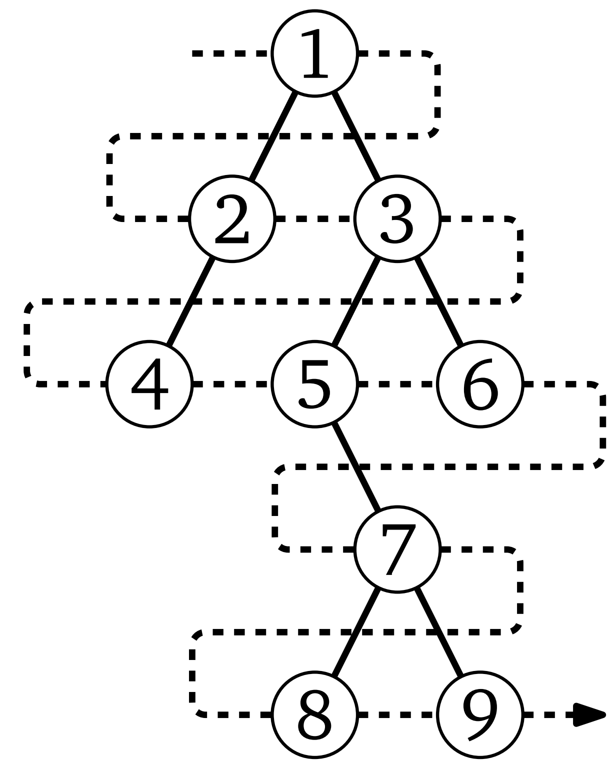
The tree type:

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

The main procedure:

```
bfs' :: ([Int], Tree a) -> ([Int], Tree Int)
bfs' (nums, Empty) = (nums, Empty)
bfs' (num : nums, Node _ l r) = (num+1 : nums'', Node num l' r')
  where (nums',  l') = bfs' (nums,  l)
        (nums'', r') = bfs' (nums', r)
```

BFS Numbering



The tree type:

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

The main procedure:

```
bfs' :: ([Int], Tree a) -> ([Int], Tree Int)
bfs' (nums, Empty) = (nums, Empty)
bfs' (num : nums, Node _ l r) = (num+1 : nums'', Node num l' r')
  where (nums',  l') = bfs' (nums,  l)
        (nums'', r') = bfs' (nums', r)
```

The magic wand: laziness

```
bfs :: Tree a -> Tree Int
bfs t = t'
  where (nums, t') = bfs' (1 : nums, t)
```

The Pitfalls of Laziness

Three kinds of folds:

Right-to-left

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x []     = x
foldr f x (y:ys) = f y (foldr x ys)
```

The Pitfalls of Laziness

Three kinds of folds:

Right-to-left

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x []     = x
foldr f x (y:ys) = f y (foldr x ys)
```

Left-to-right, lazy

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f x []     = x
foldl f x (y:ys) = foldl (f x y) ys
```

The Pitfalls of Laziness

Three kinds of folds:

Right-to-left

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x []     = x
foldr f x (y:ys) = f y (foldr x ys)
```

Left-to-right, lazy

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f x []     = x
foldl f x (y:ys) = foldl (f x y) ys
```

Left-to-right, strict

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f x []     = x
foldl' f x (y:ys) = let x' = f x y
                    in  x' `seq` foldl' f x' ys
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr  (+) 0 [1 .. n]` $O(n)$

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr (+) 0 [1 .. n]` $O(n)$

```
foldr (+) 0 [1 .. 5]
    ↓ Recursive call
foldr (+) 0 [2 .. 5]
    ↓ Recursive call
foldr (+) 0 [3 .. 5]
    ↓ Recursive call
foldr (+) 0 [4 .. 5]
    ↓ Recursive call
foldr (+) 0 [5]
    ↓ Recursive call
foldr (+) 0 []
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr (+) 0 [1 .. n]` $O(n)$

$$
\begin{array}{l}
\texttt{foldr (+) 0 [1 .. 5]} \\
\quad \downarrow \text{Recursive call} \\
\texttt{foldr (+) 0 [2 .. 5]} \\
\quad \downarrow \text{Recursive call} \\
\texttt{foldr (+) 0 [3 .. 5]} \\
\quad \downarrow \text{Recursive call} \\
\texttt{foldr (+) 0 [4 .. 5]} \\
\quad \downarrow \text{Recursive call} \\
\texttt{foldr (+) 0 [5]} \\
\quad \downarrow \text{Recursive call} \\
\texttt{foldr (+) 0 []} \qquad \boxed{0}
\end{array}
$$

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr  (+) 0 [1 .. n]` $O(n)$

```
foldr (+) 0 [1 .. 5]
   | Recursive call
   v
foldr (+) 0 [2 .. 5]
   | Recursive call
   v
foldr (+) 0 [3 .. 5]
   | Recursive call
   v
foldr (+) 0 [4 .. 5]
   | Recursive call
   v
foldr (+) 0 [5]                (+) 5
   | Recursive call
   v
foldr (+) 0 []                   0
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

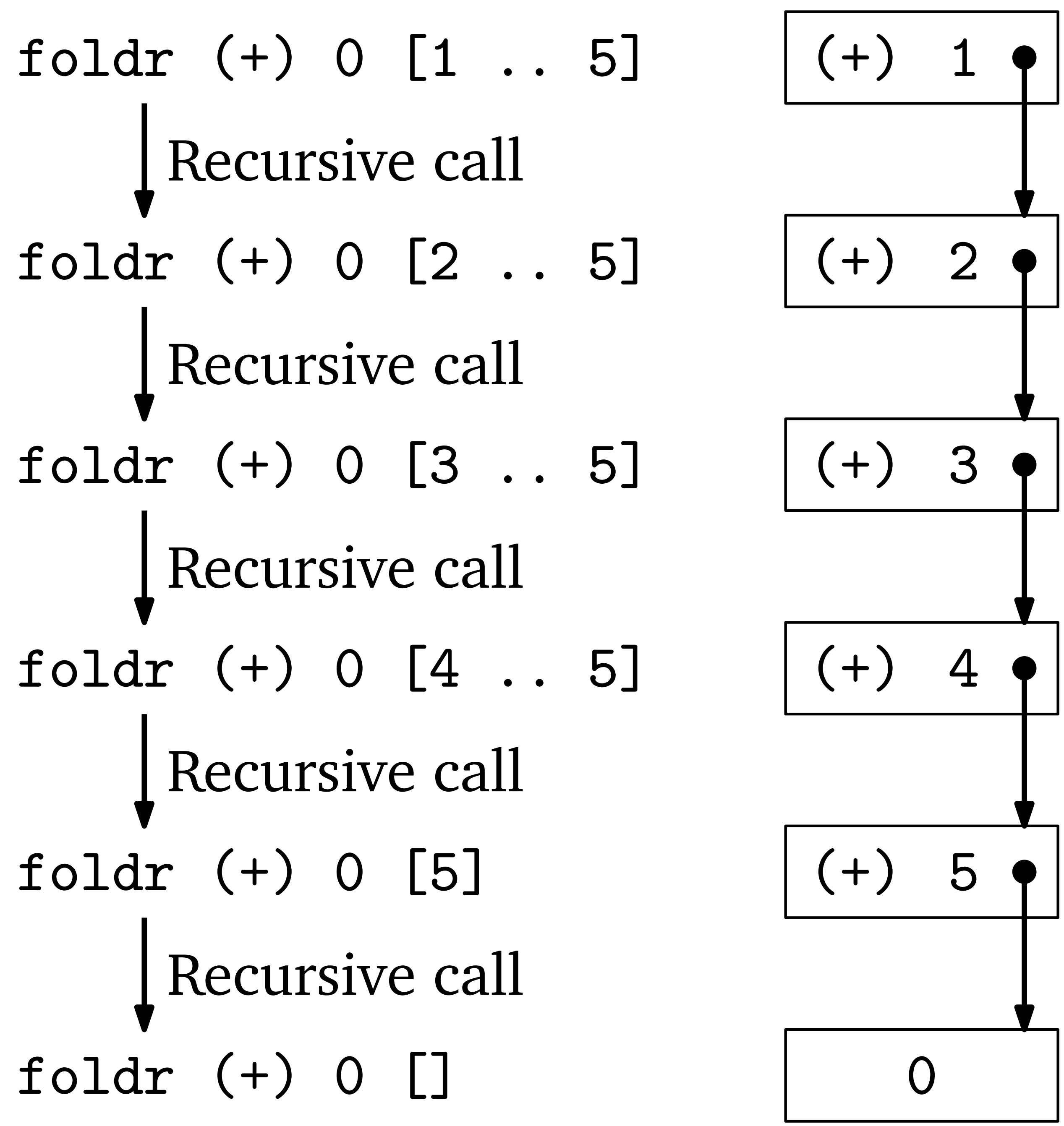
`foldr (+) 0 [1 .. n]` $O(n)$

```
foldr (+) 0 [1 .. 5]
    ↓ Recursive call
foldr (+) 0 [2 .. 5]
    ↓ Recursive call
foldr (+) 0 [3 .. 5]
    ↓ Recursive call
foldr (+) 0 [4 .. 5]        (+) 4 •
    ↓ Recursive call
foldr (+) 0 [5]             (+) 5 •
    ↓ Recursive call
foldr (+) 0 []              0
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr (+) 0 [1 .. n]` $O(n)$



The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
```
$O(n)$

```
foldl  (+) 0 [1 .. n]
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
foldl  (+) 0 [1 .. n]
```

`foldr (+) 0 [1 .. n]`	$O(n)$
`foldl (+) 0 [1 .. n]`	$O(n)$

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
foldl  (+) 0 [1 .. n]
```

$O(n)$

$O(n)$

```
foldl (+) 0 [1 .. 5]
```

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
```
$O(n)$

```
foldl  (+) 0 [1 .. n]
```
$O(n)$

```
foldl (+) 0 [1 .. 5]
```

$\longrightarrow$ `foldl (+) (0 + 1) [2 .. 5]`

The Pitfalls of Laziness

Space usage of summing a list of integers:

foldr (+) 0 [1 .. n] $O(n)$
foldl (+) 0 [1 .. n] $O(n)$

```
foldl (+) 0 [1 .. 5]
```

$\longrightarrow$ `foldl (+) (0 + 1) [2 .. 5]`

$\longrightarrow$ `foldl (+) ((0 + 1) + 2) [3 .. 5]`

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
foldl  (+) 0 [1 .. n]
```

$O(n)$
$O(n)$

```
foldl (+) 0 [1 .. 5]
```

$\longrightarrow$ `foldl (+) (0 + 1) [2 .. 5]`

$\longrightarrow$ `foldl (+) ((0 + 1) + 2) [3 .. 5]`

$\longrightarrow$ `foldl (+) (((0 + 1) + 2) + 3) [4 .. 5]`

$\longrightarrow$ `foldl (+) ((((0 + 1) + 2) + 3) + 4) [5]`

$\longrightarrow$ `foldl (+) (((((0 + 1) + 2) + 3) + 4) + 5) []`

The Pitfalls of Laziness

Space usage of summing a list of integers:

`foldr (+) 0 [1 .. n]` $O(n)$
`foldl (+) 0 [1 .. n]` $O(n)$

```
foldl (+) 0 [1 .. 5]
⟶ foldl (+) (0 + 1) [2 .. 5]
⟶ foldl (+) ((0 + 1) + 2) [3 .. 5]
⟶ foldl (+) (((0 + 1) + 2) + 3) [4 .. 5]
⟶ foldl (+) ((((0 + 1) + 2) + 3) + 4) [5]
⟶ foldl (+) (((((0 + 1) + 2) + 3) + 4) + 5) []
⟶ (((((0 + 1) + 2) + 3) + 4) + 5)
```

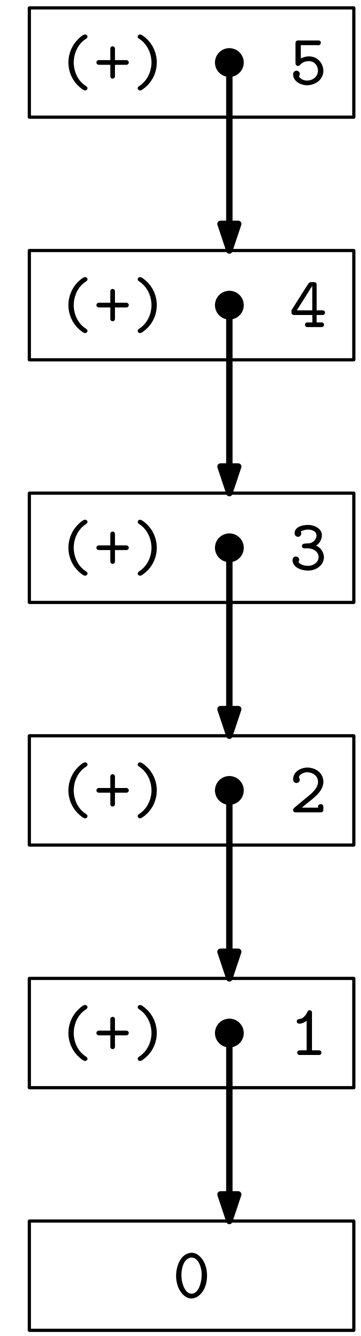
The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]
foldl  (+) 0 [1 .. n]
```

$O(n)$

$O(n)$




```
foldl (+) 0 [1 .. 5]
```

$\longrightarrow$ `foldl (+) (0 + 1) [2 .. 5]`

$\longrightarrow$ `foldl (+) ((0 + 1) + 2) [3 .. 5]`

$\longrightarrow$ `foldl (+) (((0 + 1) + 2) + 3) [4 .. 5]`

$\longrightarrow$ `foldl (+) ((((0 + 1) + 2) + 3) + 4) [5]`

$\longrightarrow$ `foldl (+) (((((0 + 1) + 2) + 3) + 4) + 5) []`

$\longrightarrow$ `(((((0 + 1) + 2) + 3) + 4) + 5)`

The Pitfalls of Laziness

Space usage of summing a list of integers:

```
foldr  (+) 0 [1 .. n]    O(n)
foldl  (+) 0 [1 .. n]    O(n)
foldl' (+) 0 [1 .. n]    O(1)
```

```
foldl' (+) 0 [1 .. 5]

⟶ foldl' (+) 1 [2 .. 5]

⟶ foldl' (+) 3 [3 .. 5]

⟶ foldl' (+) 6 [4 .. 5]

⟶ foldl' (+) 10 [5]

⟶ foldl' (+) 15 []

⟶ 15
```

Types

Built-in types:

`Int`, `Rational`, `Float`, `Char`, `String` (`[Char]`), lists, pairs, . . .

Types

Built-in types:

`Int`, `Rational`, `Float`, `Char`, `String` (`[Char]`), lists, pairs, ...

Enumeration types:

```
data Colour = Red | Blue | Green | Yellow
```

Types

Built-in types:

`Int`, `Rational`, `Float`, `Char`, `String` (`[Char]`), lists, pairs, ...

Enumeration types:

```
data Colour = Red | Blue | Green | Yellow
```

Parameterized types:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
data Either a b = Left a | Right b
data Maybe a = Just a | Nothing
```

Types

Built-in types:

`Int`, `Rational`, `Float`, `Char`, `String` (`[Char]`), lists, pairs, ...

Enumeration types:

```
data Colour = Red | Blue | Green | Yellow
```

Parameterized types:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
data Either a b = Left a | Right b
data Maybe a = Just a | Nothing
```

Types with accessor functions:

```
data Person = Person { name :: String, age :: Int
                     , address :: String }
```

Types

Built-in types:

`Int`, `Rational`, `Float`, `Char`, `String` (`[Char]`), lists, pairs, ...

Enumeration types:

```
data Colour = Red | Blue | Green | Yellow
```

Parameterized types:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
data Either a b = Left a | Right b
data Maybe a = Just a | Nothing
```

Types with accessor functions:

```
data Person = Person { name :: String, age :: Int
                     , address :: String }

p = Person { name = "Norbert Zeh", age = "100"
           , address = "Halifax" }
name p -- "Norbert Zeh"
q = p { age = "39" }
```

The Unrealistic Dream of No Side Effects

Advantage of disallowing side effects:

- The value of a function depends only on its arguments. Two invocations of the function with the same arguments are guaranteed to produce the same result.
- Theoreticians like this because it makes formal reasoning about code correctness easier.
- Practical benefit: Once you've tested a function and verified that it produces the correct result, it is guaranteed to produce the correct result at least on the inputs you tested.

The Unrealistic Dream of No Side Effects

Advantage of disallowing side effects:

- The value of a function depends only on its arguments. Two invocations of the function with the same arguments are guaranteed to produce the same result.
- Theoreticians like this because it makes formal reasoning about code correctness easier.
- Practical benefit: Once you've tested a function and verified that it produces the correct result, it is guaranteed to produce the correct result at least on the inputs you tested.

The need for side effects:

- Interactions with the real world require side effects. Without this interaction, programs are useless.
- Storing state in data structures and updating these data structures requires side effects.
- ...

The IO Monad

```
-- Read a character from stdin and return it
getChar :: IO Char
```

This is an ***action*** in the ***IO monad***. It is ***not*** a function.

A ***monad*** is a structure that allows us to sequence actions.

The ***IO monad*** is the monad that allows us to interact with the outside world.

The IO Monad

```
-- Read a character from stdin and return it
getChar :: IO Char
```

This is an ***action*** in the ***IO monad***. It is ***not*** a function.

A ***monad*** is a structure that allows us to sequence actions.

The ***IO monad*** is the monad that allows us to interact with the outside world.

Every Haskell program must have a `main` function of type

```
main :: IO ()
```

- When you start the program, this action is executed.
- It may be composed of smaller IO actions that are sequenced together.
- These actions call pure functions to carry out the part of the computation that is purely functional.
- The aim is to create a clear separation between the part of the computation that has side effects (which needs to be expressed as monadic actions) and the part that does not (which is expressed using pure functions).

IO Monad: Example

```
database :: [(String, Int)]
database = [("Norbert", 39), ("Luca", 9), ("Mateo", 1)]

lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup x []                  = Nothing
lookup x ((k, v):vs) | x == k    = Just v
                     | otherwise = lookup x vs

main :: IO ()
main = do
  name <- getLine
  if name == "quit"
    then return ()
    else do let age = lookup name database
            maybe (putStrLn $ "I don't know the age of " ++
                                   name ++ ".")
                  (\a -> putStrLn $ "The age of " ++ name ++
                                    " is " ++ show a ++ ".")
                  age
            main
```

Monads

```
class Monad m where
  (>>=)  :: forall a b . m a -> (a -> m b) -> m b
  (>>)   :: forall a b . m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Monads

```
class Monad m where
  (>>=)  :: forall a b . m a -> (a -> m b) -> m b
  (>>)   :: forall a b . m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Examples:

```
readAndEcho :: IO ()
readAndEcho = getLine >>= putStrLn
```

```
getLine  :: IO String
putStrLn :: String -> IO ()
```

Monads

```
class Monad m where
  (>>=)  :: forall a b . m a -> (a -> m b) -> m b
  (>>)   :: forall a b . m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Examples:

```
readAndEcho :: IO ()
readAndEcho = getLine >>= putStrLn
```

```
getLine  :: IO String
putStrLn :: String -> IO ()
```

```
sillyPrint :: IO ()
sillyPrint = return "This is printed" >>= putStrLn
```

Monads

```
class Monad m where
  (>>=)  :: forall a b . m a -> (a -> m b) -> m b
  (>>)   :: forall a b . m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Examples:

```
readAndEcho :: IO ()
readAndEcho = getLine >>= putStrLn
```

```
getLine  :: IO String
putStrLn :: String -> IO ()
```

```
sillyPrint :: IO ()
sillyPrint = return "This is printed" >>= putStrLn
```

```
printTwoLines :: String -> String -> IO ()
printTwoLines a b = putStrLn a >> putStrLn b
```

Monads

```
class Monad m where
  (>>=)  :: forall a b . m a -> (a -> m b) -> m b
  (>>)   :: forall a b . m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Examples:

```
readAndEcho :: IO ()
readAndEcho = getLine >>= putStrLn
```

```
getLine  :: IO String
putStrLn :: String -> IO ()
```

```
sillyPrint :: IO ()
sillyPrint = return "This is printed" >>= putStrLn
```

```
printTwoLines :: String -> String -> IO ()
printTwoLines a b = putStrLn a >> putStrLn b
```

```
failIfOdd :: Int -> IO ()
failIfOdd x = if even x then return () else fail "x is odd"
```

Do Notation

Standard monadic composition of actions sure isn't pretty:

```
getAndPrintTwoStrings :: IO ()
getAndPrintTwoStrings = getString                >>= \s1 ->
                        getString                >>= \s2 ->
                        putStrLn $ "S1 = " ++ s1 >>
                        putStrLn $ "S2 = " ++ s2
```

do-notation makes this much easier to write:

```
getAndPrintTwoStrings :: IO ()
getAndPrintTwoStrings = do
  s1 <- getString
  s2 <- getString
  putStrLn $ "S1 = " ++ s1
  putStrLn $ "S2 = " ++ s2
```

A preprocessing step translates this into the form above and then compiles the above code.

Lazy I/O

Assume we want to copy a file "input" into a file "output".

This one works:

```
main :: IO ()
main = do
  infile  <- openFile "input"
                      ReadMode
  outfile <- openFile "output"
                      WriteMode
  txt    <- hGetContents infile
  hPutStr outfile txt
  hClose infile
  hClose outfile
```

Lazy I/O

Assume we want to copy a file "input" into a file "output".

This one works:

```
main :: IO ()
main = do
  infile  <- openFile "input"
                      ReadMode
  outfile <- openFile "output"
                      WriteMode
  txt    <- hGetContents infile
  hPutStr outfile txt
  hClose infile
  hClose outfile
```

This one doesn't:

```
main :: IO ()
main = do
  infile  <- openFile "input"
                      ReadMode
  outfile <- openFile "output"
                      WriteMode
  txt    <- hGetContents infile
  hClose infile
  hPutStr outfile txt
  hClose outfile
```

Lazy I/O

Assume we want to copy a file “input” into a file “output”.

This one works:

```
main :: IO ()
main = do
  infile  <- openFile "input"
                      ReadMode
  outfile <- openFile "output"
                      WriteMode
  txt     <- hGetContents infile
  hPutStr outfile txt
  hClose infile
  hClose outfile
```

This one doesn’t:

```
main :: IO ()
main = do
  infile  <- openFile "input"
                      ReadMode
  outfile <- openFile "output"
                      WriteMode
  txt     <- hGetContents infile
  hClose infile
  hPutStr outfile txt
  hClose outfile
```

`hGetContents` has no reason to actually read the input file before it is closed.

Once we try to write the file contents to the output file, the input file is closed already, and trying to populate `txt` from the input file we encounter an EOF.

Pure Computations with State

It is common to have a computation that is pure in the sense that its result depends only on the inputs, but it needs to maintain state during its execution.

```
seededRandomSequence :: Int -> Int -> [Int]
seededRandomSequence seed n = fst (genseq seed n)

genseq :: Int -> Int -> ([Int], Int)
genseq seed 0 = ([], seed)
genseq seed n = (x:xs, seed'')
  where
    (x,  seed' ) = generateRandomNumberAndSeed seed
    (xs, seed'') = genseq seed' (n - 1)

main :: IO ()
main = do
  let xs = seededRandomSequence 15321 100
  ...
```

A Non-Solution: Lift the Computation into the IO Monad

```
seededRandomSequence :: Int -> Int -> IO [Int]
seededRandomSequence seed n = do
  st <- newIORef seed
  mapM (const $ gennum st) [1 .. n]

gennum :: (IORef Int) -> IO Int
gennum st = do
  seed <- readIORef st
  let (x, seed') = generateRandomNumberAndSeed seed
  writeIORef st seed'
  return x

main :: IO ()
main = do
  xs <- seededRandomSequence 15321 100
  ...
```

Solution: Use the State Monad

```
import Control.Monad.State

type St = State Int

seededRandomSequence :: Int -> Int -> [Int]
seededRandomSequence seed n = evalState (genseq n) seed

genseq :: Int -> St [Int]
genseq n = mapM (const gennum) [1 .. n]

gennum :: St Int
gennum = do
  seed <- get
  let (x, seed') = generateRandomNumberAndSeed seed
  put seed'
  return x

main :: IO ()
main = do
  let xs = seededRandomSequence 15321 100
```

The State Monad

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
  a >>= b  = State $\s -> let (x, s') = runState a s
                          in  runState (b x) s'
  a >>  b  = a >>= const b
  return x = State $ \s -> (x, s)
  fail     = error
```

The State Monad

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
  a >>= b  = State $\s -> let (x, s') = runState a s
                          in  runState (b x) s'
  a >>  b  = a >>= const b
  return x = State $ \s -> (x, s)
  fail     = error


get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> ((), s)

modify :: (s -> s) -> State s ()
modify f = State $ \s -> ((), f s)

evalState :: State s a -> s -> a
evalState st = fst . runState st
```

Error Handling with Maybe and Either

A type for computations that may fail to produce a result:

```
data Maybe a = Just a | Nothing
```

Error Handling with Maybe and Either

A type for computations that may fail to produce a result:

```
data Maybe a = Just a | Nothing
```

Default values for failed computations:

```
maybe :: a -> (b -> a) -> Maybe b -> a
```

Example:

```
maybe 2 (* 2) Nothing  -- 2
maybe 2 (* 2) (Just 3) -- 6
```

Error Handling with Maybe and Either

A type for computations that may fail to produce a result:

```
data Maybe a = Just a | Nothing
```

Default values for failed computations:

```
maybe :: a -> (b -> a) -> Maybe b -> a
```

Example:

```
maybe 2 (* 2) Nothing  -- 2
maybe 2 (* 2) (Just 3) -- 6
```

Sequencing computations that may fail:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

a :: [(String, Int)]
b :: [(Int, Bool)]

let x = lookup "John Doe" a
    y = maybe Nothing (flip lookup b) x
    z = maybe False id y
```

Error Handling with Maybe and Either

Maybe is a monad:

```
instance Monad Maybe where
  (Just x) >>= a = a x      -- (>>=)  :: m a -> (a -> m b) -> m b
  Nothing  >>= _ = Nothing
  (Just _) >>  a = a        -- (>>)   :: m a -> m b -> m b
  Nothing  >>  _ = Nothing
  return         = Just     -- return :: a -> m a
  fail _         = Nothing -- fail   :: String -> m a
```

Error Handling with Maybe and Either

Maybe is a monad:

```
instance Monad Maybe where
  (Just x) >>= a = a x      -- (>>=)  :: m a -> (a -> m b) -> m b
  Nothing  >>= _ = Nothing
  (Just _) >>  a = a        -- (>>)   :: m a -> m b -> m b
  Nothing  >>  _ = Nothing
  return         = Just     -- return :: a -> m a
  fail _         = Nothing  -- fail   :: String -> m a
```

Sequencing computations that may fail:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

a :: [(String, Int)]
b :: [(Int, Bool)]

let y = lookup "John Doe" a >>= flip lookup b
    z = maybe False id y
```

Error Handling with Maybe and Either

Maybe is a monad:

```
instance Monad Maybe where
  (Just x) >>= a = a x      -- (>>=)  :: m a -> (a -> m b) -> m b
  Nothing  >>= _ = Nothing
  (Just _) >>  a = a        -- (>>)   :: m a -> m b -> m b
  Nothing  >>  _ = Nothing
  return         = Just     -- return :: a -> m a
  fail _         = Nothing -- fail   :: String -> m a
```

Sequencing computations that may fail:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

a :: [(String, Int)]
b :: [(Int, Bool)]

let y = do x <- lookup "John Doe" a
           lookup x b
    z = maybe False id y
```

Error Handling with Maybe and Either

A type for computations with two kinds of outcomes:

```
data Either a b = Left a | Right b
```

Error Handling with Maybe and Either

A type for computations with two kinds of outcomes:

```
data Either a b = Left a | Right b
```

Unifying the two result types:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Example:

```
either (== 'a') (== 1) (Left 'b') -- False
either (== 'a') (== 1) (Right 1)  -- True
```

Error Handling with Maybe and Either

A type for computations with two kinds of outcomes:

```
data Either a b = Left a | Right b
```

Unifying the two result types:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Example:

```
either (== 'a') (== 1) (Left 'b') -- False
either (== 'a') (== 1) (Right 1)  -- True
```

Sequencing computations that may fail:

```
scan  :: String  -> Either String [Token]
parse :: [Token] -> Either String ParseTree

let toks = scan text
    tree = either Left parse toks
either putStrLn doSomethingWithParseTree tree
```

Error Handling with Maybe and Either

(Either String) is a monad:

```
instance Monad (Either String) where
  (Right x) >>= a = a x      -- (>>=)  :: m a -> (a -> m b) -> m b
  (Left  e) >>= _ = Left e
  (Right _) >>  a = a        -- (>>)   :: m a -> m b -> m b
  (Left  e) >>  _ = Left e
  return          = Right    -- return :: a -> m a
  fail            = Left     -- fail   :: String -> m a
```

Error Handling with Maybe and Either

(Either String) is a monad:

```
instance Monad (Either String) where
  (Right x) >>= a = a x       -- (>>=)  :: m a -> (a -> m b) -> m b
  (Left  e) >>= _ = Left e
  (Right _) >>  a = a         -- (>>)   :: m a -> m b -> m b
  (Left  e) >>  _ = Left e
  return          = Right     -- return :: a -> m a
  fail            = Left      -- fail   :: String -> m a
```

Sequencing computations that may fail:

```
scan  :: String  -> Either String [Token]
parse :: [Token] -> Either String ParseTree
```

```
let tree = scan text >>= parse
either putStrLn doSomethingWithParseTree tree
```

Error Handling with Maybe and Either

(Either String) is a monad:

```
instance Monad (Either String) where
  (Right x) >>= a = a x       -- (>>=)  :: m a -> (a -> m b) -> m b
  (Left  e) >>= _ = Left e
  (Right _) >>  a = a         -- (>>)   :: m a -> m b -> m b
  (Left  e) >>  _ = Left e
  return          = Right     -- return :: a -> m a
  fail            = Left      -- fail   :: String -> m a
```

Sequencing computations that may fail:

```
scan  :: String  -> Either String [Token]
parse :: [Token] -> Either String ParseTree

let tree = do toks <- scan text
              parse toks
either putStrLn doSomethingWithParseTree tree
```

Lists as a Monad

[] is a monad:

```
instance Monad [] where
  xs >>= a = concatMap a xs
  xs >>  a = concatMap (const a) xs
  return x = [x]
  fail _   = []
```

Lists as a Monad

[] is a monad:

```
instance Monad [] where
  xs >>= a = concatMap a xs
  xs >>  a = concatMap (const a) xs
  return x = [x]
  fail _   = []
```

```
transFunc  :: [((State, Symbol), [State])]
accStates  :: [State]
startState :: State
```

```
runNFA :: [Symbol] -> [State]
runNFA = foldM go startState
  where go s x      = epsClose s >>= flip goChar x >>= epsClose
        epsClose s = s : (goEps s >>= epsClose)
        goEps  s   = maybe [] id $ lookup (s, E) transFunc
        goChar s x = maybe [] id $ lookup (s, x) transFunc
```

```
isInLanguage :: [Symbol] -> Bool
isInLanguage = any (`elem` accStates) . runNFA
```

Code Structuring using Modules

```haskell
module A (Transparent(..), Opaque, toOpaque, fromOpaque) where

data Transparent = T { x, y :: Int }
data Opaque      = O { ox, oy :: Int }

toOpaque :: Transparent -> Opaque
toOpaque (Transparent a b) = Opaque a b

fromOpaque :: Opaque -> Transparent
fromOpaque (Opaque a b) = Transparent a b
```

Code Structuring using Modules

```
module A (Transparent(..), Opaque, toOpaque, fromOpaque) where

data Transparent = T { x, y :: Int }
data Opaque      = O { ox, oy :: Int }

toOpaque :: Transparent -> Opaque
toOpaque (Transparent a b) = Opaque a b

fromOpaque :: Opaque -> Transparent
fromOpaque (Opaque a b) = Transparent a b
```

```
module B where

import A

t = T 1 2
o = toOpaque t
t' = fromOpaque o
(a, b) = (x t', y t')
(c, d) = (ox o, oy o) -- Error
```

Code Structuring using Modules

```
module A (a, b) where
...
module B (c, d) where
...

module E where

import A (a)
import A as C hiding (a)
import qualified B as D

b   -- refers to A's b
C.b -- also refers to A's b
a   -- refers to A's a
C.a -- error, hidden
D.c -- refers to B's c
c   -- error, B must be used qualified
```

Why Do I Like Haskell?

- Think a lot, type little
- Large standard library
- hackage.haskell.org
  Central repository of lots of add-on modules I can use
- Superb documentation