INTRODUCTION TO HASKELL

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

HASKELL: A PURELY FUNCTIONAL PROGRAMMING LANGUAGE

- Functions are first-class values: Can be passed as function arguments, returned from functions.

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are (normally) immutable.

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are (normally) immutable.
- Deeply grounded in the mathematics of computing.

HASKELL: A PURELY FUNCTIONAL PROGRAMMING LANGUAGE

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are (normally) immutable.
- Deeply grounded in the mathematics of computing.
- Effectful computations are modelled in a functional manner.

- Functions are first-class values: Can be passed as function arguments, returned from functions.
- Variables are (normally) immutable.
- Deeply grounded in the mathematics of computing.
- Effectful computations are modelled in a functional manner.
- Elegant and concise.

VALUES AND FUNCTIONS

In Haskell, functions are values and values are (nullary) functions.

In Haskell, functions are values and values are (nullary) functions.

C++:

```
int x = 2;
```

Haskell:

```
x :: Int
x = 2
```

In Haskell, functions are values and values are (nullary) functions.

C++:

```
int x = 2;
```

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

Haskell:

```
x :: Int
x = 2
```

```
x :: Int
x = 2
```

In Haskell, functions are values and values are (nullary) functions.

C++:

```
int x = 2;
```

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

```
int add(int x, int y) {
  return x + y;
}
```

Haskell:

```
x :: Int
x = 2
```

```
x :: Int
x = 2
```

```
add :: Int -> Int -> Int
add x y = x + y
```

Local variables are useful in many programming languages to store intermediate results.

Haskell is no different.

The following two pieces of code behave identically:

```
veclen :: (Float, Float) -> Float
veclen (x, y) = sqrt(xx + yy)
  where xx = x * x
        yy = y * y
```

```
veclen :: (Float, Float) -> Float
veclen (x, y) = let xx = x * x
                    yy = y * y
                in  sqrt(xx + yy)
```

VARIABLES ARE IMMUTABLE

C++:

```
int four() {
  int x = 2;
  x = x + 2;
  return x;
}
```

... returns 4.

Haskell:

```
four :: Int
four = x
  where x = 2
        x = x + 2
```

... gives a compile-time error.

VARIABLES ARE IMMUTABLE

C++:

```
int four() {
  int x = 2;
  x = x + 2;
  return x;
}
```

... returns 4.

Haskell:

```
four :: Int
four = x2
  where x1 = 2
        x2 = x1 + 2
```

... works.

VARIABLES ARE IMMUTABLE

C++:

```
int four() {
  int x = 2;
  x = x + 2;
  return x;
}
```

... returns 4.

Haskell:

```
four :: Int
four = x2
  where x2 = x1 + 2
        x1 = 2
```

... also works.

if-then-else:

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

if-then-else:

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

The else-branch is mandatory! Why?

if-then-else:

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

case:

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

The else-branch is mandatory! Why?

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.

PATTERNS

```
fibonacci :: Int -> Int
fibonacci n = case n of
                0 -> 1
                1 -> 1
                _ -> fibonacci (n-1) + fibonacci (n-2)
```

```
fibonacci :: Int -> Int
fibonacci n = case n of
                0 -> 1
                1 -> 1
                _ -> fibonacci (n-1) + fibonacci (n-2)
```

Idiomatic Haskell uses multiple function definitions for this:

```
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

PATTERNS

```
fibonacci :: Int -> Int
fibonacci n = case n of
                0 -> 1
                1 -> 1
                _ -> fibonacci (n-1) + fibonacci (n-2)
```

Idiomatic Haskell uses multiple function definitions for this:

```
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Pattern matching: The first equation whose formal arguments match the arguments of the invocation is used.

```haskell
fibonacci :: Int -> Int
fibonacci n = case n of
                0 -> 1
                1 -> 1
                _ -> fibonacci (n-1) + fibonacci (n-2)
```

Idiomatic Haskell uses multiple function definitions for this:

```haskell
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
fibonacci 0 = 1
fibonacci 1 = 1
```

This gives an infinite loop!

Pattern matching: The first equation whose formal arguments match the arguments of the invocation is used.

Pattern guards: Patterns can be combined with conditions on when they match.

```
abs :: Int -> Int
abs x | x < 0       = -x
      | otherwise = x
```

Pattern guards: Patterns can be combined with conditions on when they match.

```
abs :: Int -> Int
abs x | x < 0     = -x
      | otherwise = x

sign :: Int -> Int
sign 0 = 0
sign x | x < 0     = -1
       | otherwise = 1
```

Pattern guards: Patterns can be combined with conditions on when they match.

```
abs :: Int -> Int
abs x | x < 0     = -x
      | otherwise = x

sign :: Int -> Int
sign 0 = 0
sign x | x < 0     = -1
       | otherwise = 1
```

Pattern guards can also be applied to branches of a case-statement.

LOOPS?

Loops are impossible in a functional language. Why?

LOOPS?

Loops are impossible in a functional language. Why?

What about iteration?

LOOPS?

Loops are impossible in a functional language. Why?

What about iteration?

Iteration becomes recursion.

LOOPS?

Loops are impossible in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C++:

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

LOOPS?

Loops are impossible in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C++:

```
int factorial(int n) {
  int fac = 1;
  for (int 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 are impossible in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C++:

```
int factorial(int n) {
  int fac = 1;
  for (int 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 0 = 1
factorial n = n * factorial (n-1)
```

LOOPS?

Loops are impossible in a functional language. Why?

What about iteration?

Iteration becomes recursion.

Iterative C++: Efficient

```
int factorial(int n) {
  int fac = 1;
  for (int 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 0 = 1
factorial n = n * factorial (n-1)
```

Tail recursion: When the last statement in a function is a recursive invocation of the same function, the compiler converts these recursive calls into a loop.

Tail recursion: When the last statement in a function is a recursive invocation of the same function, the compiler converts these recursive calls into a loop.

Not tail-recursive:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Tail recursion: When the last statement in a function is a recursive invocation of the same function, the compiler converts these recursive calls into a loop.

Not tail-recursive:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Tail-recursive:

```
factorial n = factorial' n 1

factorial' 0 f = f
factorial' n f = factorial' (n-1) (n*f)
```

Tail recursion: When the last statement in a function is a recursive invocation of the same function, the compiler converts these recursive calls into a loop.

Not tail-recursive:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

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

Tail-recursive:

```
factorial n = factorial' n 1

factorial' 0 f = f
factorial' n f = factorial' (n-1) (n*f)
```

Tail recursion: When the last statement in a function is a recursive invocation of the same function, the compiler converts these recursive calls into a loop.

Not tail-recursive:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

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

Tail-recursive:

```
factorial n = factorial' n 1

factorial' 0 f = f
factorial' n f = factorial' (n-1) (n*f)
```

- Constant stack size
- No overhead to maintain the stack

Primitive types:

- `Int`, `Rational`, `Float`, `Char`

Collection types:

- Lists, tuples, arrays, `String` (list of `Char`)

Custom types:

- Algebraic types (similar to `struct` in C)
- Type aliases (similar to `typedef` in C)

LISTS

Lists are ubiquitous in Haskell because they match the recursive world view of functional languages:

A list

- Is empty or
- Consists of an element, its head, followed by a list, its tail.

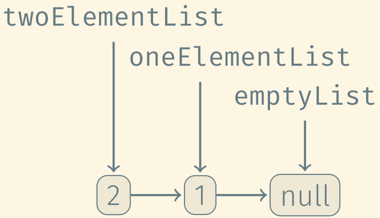
Lists are ubiquitous in Haskell because they match the recursive world view of functional languages:

A list

- Is empty or
- Consists of an element, its head, followed by a list, its tail.

In Haskell:

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



```
[1, 2, 3]
```

LIST COMPREHENSIONS

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

```
[1, 2, 3]
[1 .. 10]         -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1 ..]            -- the infinite list [1, 2, 3, ...]
```

LIST COMPREHENSIONS

```
[1, 2, 3]
[1 .. 10]        -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1 ..]           -- the infinite list [1, 2, 3, ...]
[2, 4 .. 10]     -- [2, 4, 6, 8, 10]
```

LIST COMPREHENSIONS

```
[1, 2, 3]
[1 .. 10]        -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1 ..]           -- the infinite list [1, 2, 3, ...]
[2, 4 .. 10]     -- [2, 4, 6, 8, 10]

[(x, y) | x <- [0..8], y <- [0..8], even x || even y]
-- The list of coordinates
-- .........
-- . . . . .
-- .........
-- . . . . .
-- .........
-- . . . . .
-- .........
-- . . . . .
-- .........
```

POLYMORPHISM (1)

Many functions and data types in Haskell are polymorphic (can be applied to arbitrary types, in a type-safe manner).

POLYMORPHISM (1)

Many functions and data types in Haskell are polymorphic (can be applied to arbitrary types, in a type-safe manner).

The idea is the same as generics/templates, but the use is more light-weight:

Many functions and data types in Haskell are polymorphic (can be applied to arbitrary types, in a type-safe manner).

The idea is the same as generics/templates, but the use is more light-weight:

C++/Java/Scala: Do I have a good enough reason to implement this function or class as a generic/template?

Many functions and data types in Haskell are polymorphic (can be applied to arbitrary types, in a type-safe manner).

The idea is the same as generics/templates, but the use is more light-weight:

C++/Java/Scala: Do I have a good enough reason to implement this function or class as a generic/template?

Haskell: Do I have a good reason not to make this function or type polymorphic?

POLYMORPHISM (2)

C++:

```
template <typename T>
std::vector<T> concat(const std::vector<T> &xs,
                      const std::vector<T> &ys) {
  std::vector<T> result(xs);
  for (auto &y : ys)
    result.push_back(y);
  return result;
}
```

POLYMORPHISM (2)

C++:

```
template <typename T>
std::vector<T> concat(const std::vector<T> &xs,
                      const std::vector<T> &ys) {
  std::vector<T> result(xs);
  for (auto &y : ys)
    result.push_back(y);
  return result;
}
```

Haskell:

```
concat :: [t] -> [t] -> [t]
concat []     ys = ys
concat (x:xs) ys = x : concat xs ys
```

C++:

```
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

C++:

```
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

- (Relatively) obscure message when using the template with a type `T` that does not support addition.

C++:

```
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

- (Relatively) obscure message when using the template with a type `T` that does not support addition.

Haskell:

```
sum :: [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

C++:

```
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

- (Relatively) obscure message when using the template with a type T that does not support addition.

Haskell:

```
sum :: [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

- Function must specify what "interface" it expects from its argument types.

C++:

```cpp
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

- (Relatively) obscure message when using the template with a type T that does not support addition.

Haskell:

```haskell
sum :: [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs

sum :: Num t => [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

- Function must specify what "interface" it expects from its argument types.

MAKING POLYMORPHISM TYPE-SAFE: TYPE CLASSES

C++:

```
template <typename T>
T sum(const std::vector<T> &xs) {
  T total = 0;
  for (auto x : xs)
    total += x;
  return total;
}
```

- (Relatively) obscure message when using the template with a type T that does not support addition.

Haskell:

```
sum :: [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs

sum :: Num t => [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

- Function must specify what "interface" it expects from its argument types.
- Use of function not satisfying type constraints reported upon use.

COMMON TYPE CLASSES

`Eq`:

- Supports equality testing using `==` and `/=`

`Ord`: (Requires `Eq`)

- Supports ordering using `<`, `>`, `<=`, and `>=`

`Num`:

- Supports `+`, `-`, `*`, `abs`, ..., not `/`!

`Show`:

- Supports conversion to a string using `show`

`Read`:

- Supports conversion from a string using `read`

Inspecting the contents of lists is often done using patterns, but we can also explicitly ask for the head or tail of a list:

```
head :: [t] -> t
head (x:_) = x
head _     = error "Cannot take head of empty list"

tail :: [t] -> t
tail (_:xs) = xs
tail _      = error "Cannot take tail of empty list"
```

```
-- Concatenate two lists
[1, 2] ++ [3, 4, 5] == [1 .. 5]
```

MORE LIST FUNCTIONS

```
-- Concatenate two lists
[1, 2] ++ [3, 4, 5] == [1 .. 5]

-- Concatenate a list of lists
concat [[1, 2], [3], [4, 5]] == [1 .. 5]
```

MORE LIST FUNCTIONS

```
-- Concatenate two lists
[1, 2] ++ [3, 4, 5] == [1 .. 5]

-- Concatenate a list of lists
concat [[1, 2], [3], [4, 5]] == [1 .. 5]

-- Take the first 5 elements of the list
take 5 [1 .. 10] == [1 .. 5]
```

MORE LIST FUNCTIONS

```
-- Concatenate two lists
[1, 2] ++ [3, 4, 5] == [1 .. 5]

-- Concatenate a list of lists
concat [[1, 2], [3], [4, 5]] == [1 .. 5]

-- Take the first 5 elements of the list
take 5 [1 .. 10] == [1 .. 5]

-- Drop the first 5 elements of the list
drop 5 [1 .. 10] == [6 .. 10]
```

MORE LIST FUNCTIONS

```
-- Concatenate two lists
[1, 2] ++ [3, 4, 5] == [1 .. 5]

-- Concatenate a list of lists
concat [[1, 2], [3], [4, 5]] == [1 .. 5]

-- Take the first 5 elements of the list
take 5 [1 .. 10] == [1 .. 5]

-- Drop the first 5 elements of the list
drop 5 [1 .. 10] == [6 .. 10]

-- Split the list after the 5th element
splitAt 5 [1 .. 10] = ([1 .. 5], [6 .. 10])
```

Lists can hold an arbitrary number of elements of the same type:

```
l  =  [1 .. 10]    -- l :: [Int]
l' =  'a' : l      -- error!
```

Tuples can hold a fixed number of elements of potentially different types:

```
t = ('a', 1, [2, 3]) -- t :: (Char, Int, [Int])
```

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

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

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

OPERATIONS FOR PAIRS AND TUPLES

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

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

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

Zipping and unzipping: From a pair of lists to a list of pairs and back.

```
zip ['a', 'b', 'c'] [1 .. 10] == [('a',1), ('b',2), ('c',3)]
-- The result has the length of the shorter of the two lists
```

Zipping and unzipping: From a pair of lists to a list of pairs and back.

```
zip ['a', 'b', 'c'] [1 .. 10] == [('a',1), ('b',2), ('c',3)]
-- The result has the length of the shorter of the two lists

unzip [('a',1), ('b',2), ('c',3)] = (['a', 'b', 'c'], [1, 2, 3])
```

Zipping and unzipping: From a pair of lists to a list of pairs and back.

```
zip ['a', 'b', 'c'] [1 .. 10] == [('a',1), ('b',2), ('c',3)]
-- The result has the length of the shorter of the two lists
```

```
unzip [('a',1), ('b',2), ('c',3)] = (['a', 'b', 'c'], [1, 2, 3])
```

Zipping with a function:

```
zipWith (\x y -> x + y) [1, 2, 3] [4, 5, 6] == [5, 7, 9]
```

ARRAYS (1)

Arrays do exist in Haskell and do have their uses because they support constant-time access.

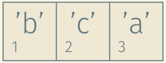
However, arrays are (normally) immutable, so updates are expensive.

Arrays do exist in Haskell and do have their uses because they support constant-time access.

However, arrays are (normally) immutable, so updates are expensive.

Creating arrays:

```
array (1,3) [(3,'a'), (1,'b'), (2,'c')]
```



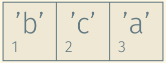
ARRAYS (1)

Arrays do exist in Haskell and do have their uses because they support constant-time access.

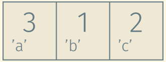
However, arrays are (normally) immutable, so updates are expensive.

Creating arrays:

```
array (1,3) [(3,'a'), (1,'b'), (2,'c')]
```




```
listArray ('a','c') [3,1,2]
```



ARRAYS (2)

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']
```

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']
```

```
a ! 1 == 'a'
a ! 3 == 'c'
```

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']

a ! 1 == 'a'
a ! 3 == 'c'

elems a == ['a', 'b', 'c']
```

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']

a ! 1 == 'a'
a ! 3 == 'c'

elems a == ['a', 'b', 'c']

assocs a == [(1,'a'), (2,'b'), (3,'c')]
```

ARRAYS (2)

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']
```

```
a ! 1 == 'a'
a ! 3 == 'c'
```

```
elems a == ['a', 'b', 'c']
```

```
assocs a == [(1,'a'), (2,'b'), (3,'c')]
```

“Updating” arrays:

```
a // [(2,'a'), (1,'d')] == listArray (1,3) ['d', 'a', 'c']
```

ARRAYS (2)

Accessing array elements:

```
let a = listArray (1,3) ['a', 'b', 'c']
```

```
a ! 1 == 'a'
a ! 3 == 'c'
```

```
elems a == ['a', 'b', 'c']
```

```
assocs a == [(1,'a'), (2,'b'), (3,'c')]
```

"Updating" arrays:

```
a // [(2,'a'), (1,'d')] == listArray (1,3) ['d', 'a', 'c']
```

`(//)` does not update the original array but creates a new array with the specified elements changed. Why?

Counting characters in a text:

```
countChars :: String -> [(Char, Int)]
countChars txt = filter nonZero (assocs counts)
  where counts         = accumArray (+) 0 ('a','z')
                                        (zip txt (repeat 1))
        nonZero (_, c) = c > 0

countChars "mississippi" == [('i',4), ('m',1), ('p',2), ('s',4)]
```

Custom algebraic data types (similar to classes/structs) are defined using `data`.

Custom algebraic data types (similar to classes/structs) are defined using `data`.

A simple enum type:

```
data Colors = Red | Green | Blue
```

Custom algebraic data types (similar to classes/structs) are defined using `data`.

A simple enum type:

```
data Colors = Red | Green | Blue
              deriving (Eq, Ord, Ix)
```

CUSTOM ALGEBRAIC DATA TYPES

Custom algebraic data types (similar to classes/structs) are defined using `data`.

A simple enum type:

```
data Colors = Red | Green | Blue
              deriving (Eq, Ord, Ix)
```

A binary tree:

```
data Tree t = Leaf
            | Node { item        :: t
                   , left, right :: Tree t
                   }
```

Custom algebraic data types (similar to classes/structs) are defined using `data`.

A simple enum type:

```
data Colors = Red | Green | Blue
              deriving (Eq, Ord, Ix)
```

A binary tree:

```
data Tree t = Leaf
            | Node { item        :: t
                   , left, right :: Tree t
                   }

fun1 (Tree x l r) = ...    -- work with x, l, and r
fun2 tree         = ...    -- work with (item tree), (left tree),
                           -- and (right tree)
updItem tree x    = tree { item = x }
```

WHEN "DATA" IS TOO COSTLY

Type aliases similar to `typedef` or `using` in C/C++ are defined using `type`:

```
type Point     = (Float, Float)
type PointList = [Point]
```

`Point` and `(Float, Float)` can be used 100% interchangeably.

WHEN “DATA” IS TOO COSTLY

Type aliases similar to `typedef` or `using` in C/C++ are defined using `type`:

```
type Point     = (Float, Float)
type PointList = [Point]
```

`Point` and `(Float, Float)` can be used 100% interchangeably.

IDs are integers but adding or multiplying them makes no sense:

```
newtype ID = ID Int
             deriving (Eq, Ord)
```

WHEN "DATA" IS TOO COSTLY

Type aliases similar to `typedef` or `using` in C/C++ are defined using `type`:

```
type Point     = (Float, Float)
type PointList = [Point]
```

`Point` and `(Float, Float)` can be used 100% interchangeably.

IDs are integers but adding or multiplying them makes no sense:

```
newtype ID = ID Int
             deriving (Eq, Ord)
```

- Internally, an `ID` is represented as an integer. With `data`, there would have been some space overhead.

WHEN "DATA" IS TOO COSTLY

Type aliases similar to `typedef` or `using` in C/C++ are defined using `type`:

```
type Point     = (Float, Float)
type PointList = [Point]
```

`Point` and `(Float, Float)` can be used 100% interchangeably.

IDs are integers but adding or multiplying them makes no sense:

```
newtype ID = ID Int
             deriving (Eq, Ord)
```

- Internally, an `ID` is represented as an integer. With `data`, there would have been some space overhead.
- Without the `deriving` clause, `ID` does not support any operations.

WHEN "DATA" IS TOO COSTLY

Type aliases similar to `typedef` or `using` in C/C++ are defined using `type`:

```
type Point     = (Float, Float)
type PointList = [Point]
```

`Point` and `(Float, Float)` can be used 100% interchangeably.

IDs are integers but adding or multiplying them makes no sense:

```
newtype ID = ID Int
             deriving (Eq, Ord)
```

- Internally, an `ID` is represented as an integer. With `data`, there would have been some space overhead.
- Without the `deriving` clause, `ID` does not support any operations.
- The `deriving` clause says that `ID`s should inherit equality and ordering from its underlying type.

@-PATTERNS

```
fun1 (Tree x l r) = ... -- work with x, l, and r
fun2 tree         = ... -- work with (item tree), (left tree),
```

- `fun1` can refer to the parts of the tree but not to the whole tree.
- `fun2` has access to the whole tree but needs to take extra steps to access its parts.
- Sometimes, we'd like to have both.

@-PATTERNS

```
fun1 (Tree x l r) = ... -- work with x, l, and r
fun2 tree         = ... -- work with (item tree), (left tree),
```

- `fun1` can refer to the parts of the tree but not to the whole tree.
- `fun2` has access to the whole tree but needs to take extra steps to access its parts.
- Sometimes, we'd like to have both.

Merging two sorted lists:

```
merge :: Ord t => [t] -> [t] -> [t]
merge []          ys          = ys
merge xs          []          = xs
merge xs@(x:xs') ys@(y:ys') | y < x     = y : merge xs  ys'
                            | otherwise = x : merge xs' ys
```

ANONYMOUS FUNCTIONS

Anonymous functions are often called λ-expressions.
Haskell people think that \ looks close enough to λ.

So an anonymous function for additing two elements together would be `\x y -> x + y`.

Anonymous functions are often called λ-expressions.
Haskell people think that \ looks close enough to λ.

So an anonymous function for additing two elements together would be `\x y -> x + y`.

The normal function definition

```
add x y = x + y
```

is just syntactic sugar for

```
add = \x y -> x + y
```

ANONYMOUS FUNCTIONS

Anonymous functions are often called λ-expressions.
Haskell people think that \ looks close enough to λ.

So an anonymous function for additing two elements together would be `\x y -> x + y`.

The normal function definition

```
add x y = x + y
```

is just syntactic sugar for

```
add = \x y -> x + y
```

or, as we will see soon, for

```
add = \x -> \y -> x + y
```

COMMON ITERATION PATTERNS (1)

Many things we do using loops in imperative languages are instances of some common patterns.

Expressing these patterns explicitly instead of hand-crafting them using loops makes our code more readable.

Mapping: Transform a list into a new list by applying a function to every element:

```
map (\x -> 2*x) [1 .. 10] == [2, 4 .. 20]
```

COMMON ITERATION PATTERNS (2)

Mapping: Transform a list into a new list by applying a function to every element:

```
map (\x -> 2*x) [1 .. 10] == [2, 4 .. 20]
```

Folding: Accumulate the elements of a list into a single value:

```
foldr (\x y -> x + y) 0 [1 .. 10] == 55
-- the sum of the list elements
```

Mapping: Transform a list into a new list by applying a function to every element:

```
map (\x -> 2*x) [1 .. 10] == [2, 4 .. 20]
```

Folding: Accumulate the elements of a list into a single value:

```
foldr (\x y -> x + y) 0 [1 .. 10] == 55
-- the sum of the list elements
```

Filtering: Extract the list elements that meet a given condition:

```
filter odd [1 .. 10] == [1, 3, 5, 7, 9]
```

IMPLEMENTING ITERATION CONSTRUCTS

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

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b as = go b as
  where go b []       = b
        go b (a:as') = f a (go b as')
```

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b as = go b as
  where go b []       = b
        go b (a:as') = f a (go b as')
```

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

CURRIED FUNCTIONS AND PARTIAL APPLICATION (1)

"Flipping" all pairs in a list:

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

“Flipping” all pairs in a list:

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

A little less verbose:

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

“Flipping” all pairs in a list:

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

A little less verbose:

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

Highly compressed:

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

"Flipping" all pairs in a list:

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

A little less verbose:

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

Highly compressed:

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

???

"Flipping" all pairs in a list:

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

A little less verbose:

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

This is (almost) what you'd do in practice.

Highly compressed:

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

???

We write a multi-argument function as

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

Why not

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

CURRIED FUNCTIONS AND PARTIAL APPLICATION (2)

We write a multi-argument function as

`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!

We write a multi-argument function as

`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` has one argument of type `a` and its result is …

CURRIED FUNCTIONS AND PARTIAL APPLICATION (2)

We write a multi-argument function as

`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` has one argument of type `a` and its result is …

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

CURRIED FUNCTIONS AND PARTIAL APPLICATION (2)

We write a multi-argument function as

`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` has one argument of type `a` and its 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`.

We write a multi-argument function as

`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` has one argument of type `a` and its 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`.

We write a multi-argument function as

`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` has one argument of type `a` and its 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` has one argument of type `(a, b, c)` and its result is of type `d`.

CURRIED FUNCTIONS AND PARTIAL APPLICATION (2)

We write a multi-argument function as

`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` has one argument of type `a` and its 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` has one argument of type `(a, b, c)` and its result is of type `d`.

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

CURRIED FUNCTIONS AND PARTIAL APPLICATION (3)

`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 itself be a function!

WHY ARE CURRIED FUNCTIONS BETTER? (1)

Multiplying all elements in a list by two.

WHY ARE CURRIED FUNCTIONS BETTER? (1)

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? (1)

Multiplying all elements in a list by two.

Without currying:

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

With currying (part 1):

WHY ARE CURRIED FUNCTIONS BETTER? (1)

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 a function of type `(*) :: Num t => t -> t -> t`.

WHY ARE CURRIED FUNCTIONS BETTER? (1)

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 a function of type `(*) :: Num t => t -> t -> t`.
- It maps its first argument `x` to a function that multiplies its second argument `y` by `x`.

WHY ARE CURRIED FUNCTIONS BETTER? (1)

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 a function of type `(*) :: Num t => t -> t -> t`.
- It maps its first argument `x` to a function that multiplies its second argument `y` by `x`.

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

With currying (part 2):

WHY ARE CURRIED FUNCTIONS BETTER? (2)

With currying (part 2):

- `map` is a function of type `map :: (a -> b) -> [a] -> [b]`.

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 every element in its argument list.

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 every element in its argument list.

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

WHY ARE CURRIED FUNCTIONS BETTER? (2)

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 every element in its argument list.

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

This is called point-free programming. The focus is on building functions from functions instead of specifying the value of a function for a particular argument.

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 every element in its argument list.

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

This is called point-free programming. The focus is on building functions from functions instead of specifying the value of a function for a particular argument.

Revisiting `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = go
  where go b []       = b
        go b (a:as') = f a (go b as')
```

Point-free programming cannot work without function composition:

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

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)
```

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 -> b) -> a -> b                     -- f $ x == f x
```

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)   -- Exchange the first two
                                         -- function arguments
```

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                 -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)    -- Exchange the first two
                                          -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)    -- Curry a function whose
                                          -- argument is a pair
```

```
($) :: (a -> b) -> a -> b                 -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)    -- Exchange the first two
                                          -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)    -- Curry a function whose
                                          -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c) -- Collapse two function
                                          -- arguments into a pair
```

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                 -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)    -- Exchange the first two
                                          -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)    -- Curry a function whose
                                          -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c)  -- Collapse two function
                                          -- arguments into a pair
```

Why the need for a function application operator?

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)   -- Exchange the first two
                                         -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)   -- Curry a function whose
                                         -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c) -- Collapse two function
                                         -- arguments into a pair
```

Why the need for a function application operator?

Function application binds more tightly than function composition, which binds more tightly than `($)`:

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                    -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)       -- Exchange the first two
                                             -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)       -- Curry a function whose
                                             -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c)     -- Collapse two function
                                             -- arguments into a pair
```

Why the need for a function application operator?

Function application binds more tightly than function composition, which binds more tightly than `($)`:

```
f :: a -> b
g :: b -> c
x :: a
```

```haskell
($) :: (a -> b) -> a -> b                 -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)    -- Exchange the first two
                                          -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)    -- Curry a function whose
                                          -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c) -- Collapse two function
                                          -- arguments into a pair
```

Why the need for a function application operator?

Function application binds more tightly than function composition, which binds more tightly than `($)`:

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

A FEW USEFUL FUNCTIONS

```
($) :: (a -> b) -> a -> b                  -- f $ x == f x
flip :: (a -> b -> c) -> (b -> a -> c)     -- Exchange the first two
                                           -- function arguments
curry :: ((a,b) -> c) -> (a -> b -> c)     -- Curry a function whose
                                           -- argument is a pair
uncurry :: (a -> b -> c) -> ((a,b) -> c) -- Collapse two function
                                           -- arguments into a pair
```

Why the need for a function application operator?

Function application binds more tightly than function composition, which binds more tightly than `($)`:

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

BACK TO SWAPELEMS

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

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

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

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

BACK TO SWAPELEMS

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

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

BACK TO SWAPELEMS

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

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

Sequences: Containers that can be "flattened" to a list:

```
class Sequence s where
  flatten :: s t -> [t]
  flatMap :: (a -> b) -> s a -> [b]
  flatMap f = map f . flatten
```

Sequences: Containers that can be "flattened" to a list:

```
class Sequence s where
  flatten :: s t -> [t]
  flatMap :: (a -> b) -> s a -> [b]
  flatMap f = map f . flatten
```

```
generalizedFilter :: Sequence s => (t -> Bool) -> s t -> [t]
generalizedFilter p = filter p . flatten
```

Lists are sequences:

```
instance Sequence [] where
  flatten = id
  flatMap = map
```

Lists are sequences:

```
instance Sequence [] where
  flatten = id
  flatMap = map
```

So are arrays:

```
instance Sequence (Array ix) where
  flatten = elems
```

Lists are sequences:

```
instance Sequence [] where
  flatten = id
  flatMap = map
```

So are arrays:

```
instance Sequence (Array ix) where
  flatten = elems
```

... and binary trees:

```
instance Sequence Tree where
  flatten Leaf         = []
  flatten (Tree x l r) = flatten l ++ [x] ++ flatten r
```

`Maybe t` can be used as the result of functions that may fail:

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

`Maybe t` can be used as the result of functions that may fail:

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

data Maybe t = Just t
             | Nothing
```

Using patterns:

```
formattedLookup :: (Eq a, Show a, Show b) => a -> [(a,b)] -> String
formattedLookup x ys = format (lookup x ys)
  where format Nothing  = "Key " ++ show x ++ " not found"
        format (Just y) = "Key " ++ show x ++ " stores value "
                                 ++ show y
```

Using patterns:

```
formattedLookup :: (Eq a, Show a, Show b) => a -> [(a,b)] -> String
formattedLookup x ys = format (lookup x ys)
  where format Nothing  = "Key " ++ show x ++ " not found"
        format (Just y) = "Key " ++ show x ++ " stores value "
                          ++ show y
```

Using `maybe`:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe def _ Nothing  = def
maybe _   f (Just x) = f x
```

Using patterns:

```
formattedLookup :: (Eq a, Show a, Show b) => a -> [(a,b)] -> String
formattedLookup x ys = format (lookup x ys)
  where format Nothing  = "Key " ++ show x ++ " not found"
        format (Just y) = "Key " ++ show x ++ " stores value "
                          ++ show y
```

Using `maybe`:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe def _ Nothing  = def
maybe _   f (Just x) = f x

lookupWithDefault :: Eq a => a -> b -> [(a,b)] -> b
lookupWithDefault x y ys = maybe y id (lookup x ys)
```

EITHER: CAPTURING DIFFERENT OUTCOMES

`Either a b` can be used as the result of computations that may produce two different outcomes:

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

tagEvensAndOdds :: [Int] -> [Either Int Int]
tagEvensAndOdds = map tag
  where tag x | even x    = Left  x
              | otherwise = Right x
```

Using patterns:

```
addOrMultiply :: [Int] -> [Int]
addOrMultiply = map aom . tagEvensAndOdds
  where aom (Left even) = even + 2
        aom (Right odd) = 2 * odd
```

Using patterns:

```
addOrMultiply :: [Int] -> [Int]
addOrMultiply = map aom . tagEvensAndOdds
  where aom (Left even) = even + 2
        aom (Right odd) = 2 * odd
```

Using `either`:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left  x) = f x
either _ g (Right y) = g y

addOrMultiply = map (either (+ 2) (* 2)) . tagEvensAndOdds
```

`map` allows us to apply a function to every list element, but we cannot `map` over the elements of a binary tree.

`map` allows us to apply a function to every list element, but we cannot `map` over the elements of a binary tree.

What if I want to apply a function to `Maybe` some value?

`map` allows us to apply a function to every list element, but we cannot `map` over the elements of a binary tree.

What if I want to apply a function to `Maybe` some value?

The `Functor` type class captures containers:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The list type is a functor:

```
instance Functor [] where
  fmap = map
```

EXAMPLES OF FUNCTORS

The list type is a functor:

```
instance Functor [] where
  fmap = map
```

So is `Maybe`:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

EXAMPLES OF FUNCTORS

The list type is a functor:

```
instance Functor [] where
  fmap = map
```

So is `Maybe`:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

... and the binary tree type:

```
instance Functor Tree where
  fmap _ Leaf         = Leaf
  fmap f (Tree x l r) = Tree (f x) (fmap f l) (fmap f r)
```

What takes longer?

- `let l1 = [1 .. 10]`
- `let l2 = [1 .. 10000000]`
- `let l3 = [1 ..]`

What takes longer?

- `let l1 = [1 .. 10]`
- `let l2 = [1 .. 10000000]`
- `let l3 = [1 ..]`

They all take constant time!

What takes longer?

- `let l1 = [1 .. 10]`
- `let l2 = [1 .. 10000000]`
- `let l3 = [1 ..]`

They all take constant time!

Haskell evaluates expressions lazily:

- Expressions are evaluated only when their value is needed.
- The evaluated value is cached, in case it's needed again.

LAZY EVALUATION

What takes longer?

- `let l1 = [1 .. 10]`
- `let l2 = [1 .. 10000000]`
- `let l3 = [1 ..]`

They all take constant time!

Haskell evaluates expressions lazily:

- Expressions are evaluated only when their value is needed.
- The evaluated value is cached, in case it's needed again.

`head l1` produces `1` and changes the representation of `l1` to `1 : [2 .. 10]`.

What takes longer?

- `let l1 = [1 .. 10]`
- `let l2 = [1 .. 10000000]`
- `let l3 = [1 ..]`

They all take constant time!

Haskell evaluates expressions lazily:

- Expressions are evaluated only when their value is needed.
- The evaluated value is cached, in case it's needed again.

`head l1` produces `1` and changes the representation of `l1` to `1 : [2 .. 10]`.

Useful consequence: We can define infinite data structures as long as we only work with finite portions of them.

WHY ARE INFINITE DATA STRUCTURES USEFUL? (1)

WHY ARE INFINITE DATA STRUCTURES USEFUL? (1)

Elegance!

WHY ARE INFINITE DATA STRUCTURES USEFUL? (1)

Elegance!

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

Elegance!

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

The hard way:

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

WHY ARE INFINITE DATA STRUCTURES USEFUL? (1)

Elegance!

Assume we write a parser and want to provide line numbers in our error messages. We need to annotate each input line with its 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 inifinite sequence of Fibonacci numbers:

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

WHY ARE INFINITE DATA STRUCTURES USEFUL? (2)

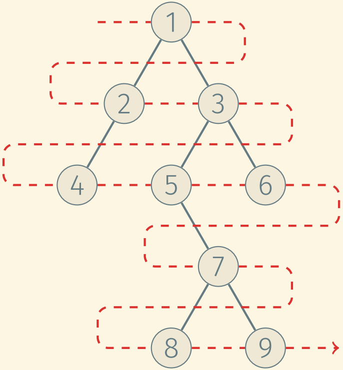
The inifinite sequence of Fibonacci numbers:

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

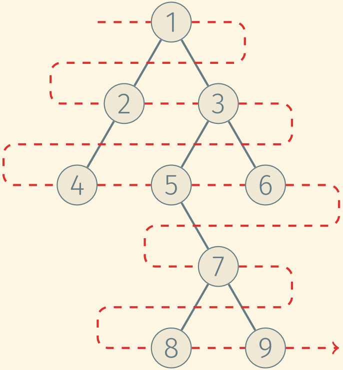
The first 10 Fibonacci numbers:

```
take 10 fibonacci == [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

BFS numbering of a binary tree



BFS numbering of a binary tree



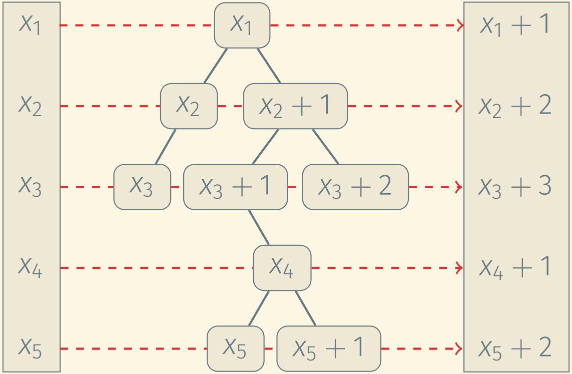
The naive solution:

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

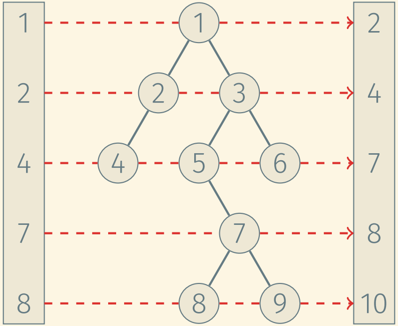
I refuse to turn this into code; it's messy.

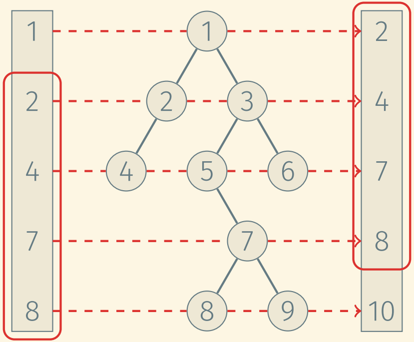
MORE LAZINESS: USING VALUES BEFORE THEY'RE COMPUTED (2)

```haskell
bfs' :: ([Int], Tree t) -> ([Int], Tree Int)
bfs' (nums, Leaf)               = (nums, Leaf)
bfs' (num:nums, Tree _ l r) = (num+1 : nums'', Tree num l' r')
  where (nums',  l') = bfs' (nums,  l)
        (nums'', r') = bfs' (nums', r)
```



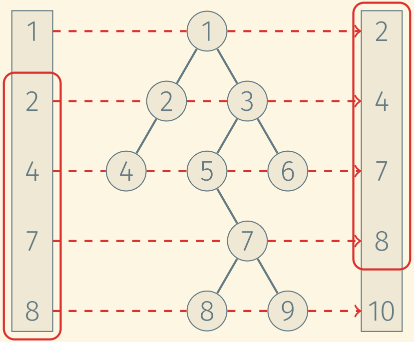
MORE LAZINESS: USING VALUES BEFORE THEY'RE COMPUTED (3)




1
2
4
7
8
1
2
3
4
5
6
7
8
9
2
4
7
8
10

MORE LAZINESS: USING VALUES BEFORE THEY'RE COMPUTED (3)

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



Many computations are about transforming collections of items.

LISTS AS CONTROL STRUCTURES (1)

Many computations are about transforming collections of items.

It would be clearest to express such sequences of transformations explicitly, but explictly building up these collections (vectors, lists, ...) is often costly.

LISTS AS CONTROL STRUCTURES (1)

Many computations are about transforming collections of items.

It would be clearest to express such sequences of transformations explicitly, but explictly building up these collections (vectors, lists, ...) is often costly.

$\Rightarrow$ We often build up complicated loops to avoid materializing intermediate collections.

LISTS AS CONTROL STRUCTURES (1)

Many computations are about transforming collections of items.

It would be clearest to express such sequences of transformations explicitly, but explictly building up these collections (vectors, lists, ...) is often costly.

$\Rightarrow$ We often build up complicated loops to avoid materializing intermediate collections.

Laziness allows us to express computations as list transformations while still not materializing any intermediate lists.

LISTS AS CONTROL STRUCTURES (2)

```
filterAndMultiply :: [Bool] -> [Int] -> [Int] -> [Int]
filterAndMultiply keep items factors = map (*) kept factors
  where kept      = map    snd  keptPairs
        keptPairs = filter fst  pairs
        pairs     = zip    keep items
```

LISTS AS CONTROL STRUCTURES (2)

```
filterAndMultiply :: [Bool] -> [Int] -> [Int] -> [Int]
filterAndMultiply keep items factors = map (*) kept factors
  where kept      = map    snd  keptPairs
        keptPairs = filter fst  pairs
        pairs     = zip    keep items
```

LISTS AS CONTROL STRUCTURES (2)

```
filterAndMultiply :: [Bool] -> [Int] -> [Int] -> [Int]
filterAndMultiply keep items factors = map (*) kept factors
  where kept      = map    snd  keptPairs
        keptPairs = filter fst  pairs
        pairs     = zip    keep items
```

- Only one node of each list needed at any point in time.
- A good compiler will optimize the lists away.

SOME PITFALLS OF LAZINESS (1)

Three kinds of folds:

SOME PITFALLS OF LAZINESS (1)

Three kinds of folds:

Right to left:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = go
  where go b []     = b
        go b (x:xs) = f x (go b xs)
```

Three kinds of folds:

Right to left:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = go
  where go b []     = b
        go b (x:xs) = f x (go b xs)
```

Left to right, lazy:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f = go
  where go a []     = a
        go a (x:xs) = go (f a x) xs
```

Three kinds of folds:

Right to left:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = go
  where go b []     = b
        go b (x:xs) = f x (go b xs)
```

Left to right, lazy:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f = go
  where go a []     = a
        go a (x:xs) = go (f a x) xs
```

Left to right, strict:

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

Space usage of summing a list of integers:

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

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

```
foldr (+) 0 [1..n]          foldr (+) 0 [1..5]
```

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

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

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

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

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

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

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

Space usage of summing a list of integers:

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

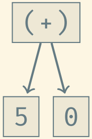
```
0
```

SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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

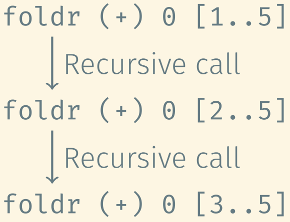
```
foldr (+) 0 [1..5]
   ↓ Recursive call
foldr (+) 0 [2..5]
   ↓ Recursive call
foldr (+) 0 [3..5]
   ↓ Recursive call
foldr (+) 0 [4..5]
```

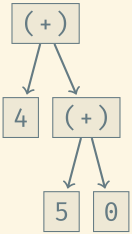


SOME PITFALLS OF LAZINESS (2)

Space usage of summing a list of integers:

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





SOME PITFALLS OF LAZINESS (2)

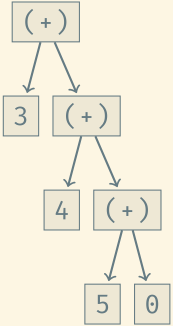
Space usage of summing a list of integers:

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

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

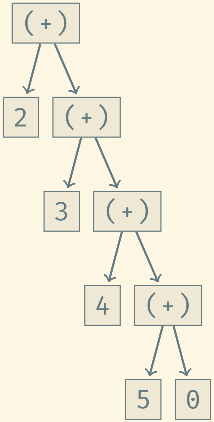
↓ Recursive call

```
foldr (+) 0 [2..5]
```



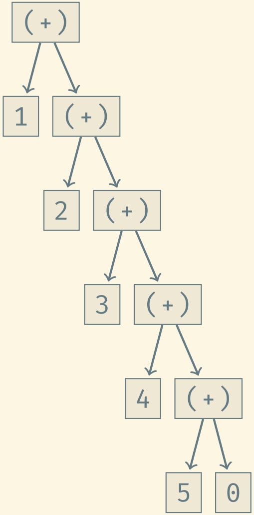
Space usage of summing a list of integers:

```
foldr (+) 0 [1..n]        foldr (+) 0 [1..5]
```



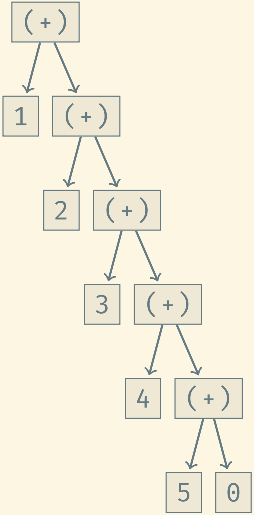
Space usage of summing a list of integers:

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



Space usage of summing a list of integers:

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



SOME PITFALLS OF LAZINESS (3)

Space usage of summing a list of integers:

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

SOME PITFALLS OF LAZINESS (3)

Space usage of summing a list of integers:

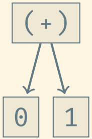
```
foldl (+) 0 [1..n]
```

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

Space usage of summing a list of integers:

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

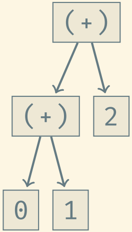
```
  foldl (+)     0          [1..5]
→ foldl (+)   (0+1)        [2..5]
```



Space usage of summing a list of integers:

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

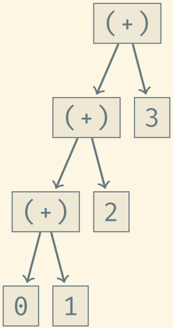
```
  foldl (+)      0                    [1..5]
→ foldl (+)     (0+1)                 [2..5]
→ foldl (+)   ((0+1) + 2)             [3..5]
```



Space usage of summing a list of integers:

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

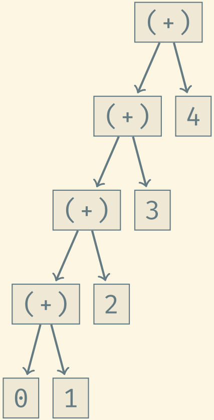
```
  foldl (+)       0                         [1..5]
→ foldl (+)      (0+1)                      [2..5]
→ foldl (+)     ((0+1) + 2)                 [3..5]
→ foldl (+)    (((0+1) + 2) + 3)            [4..5]
```



Space usage of summing a list of integers:

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

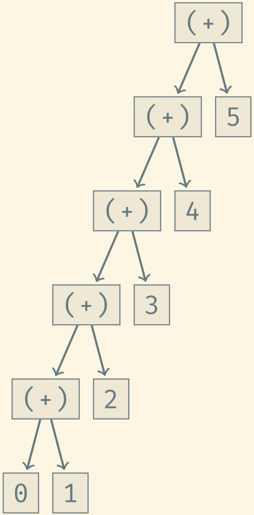


SOME PITFALLS OF LAZINESS (3)

Space usage of summing a list of integers:

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

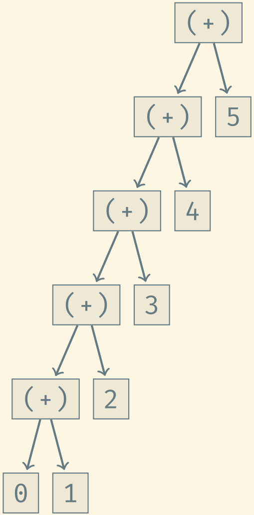
```
  foldl (+)      0                          [1..5]
→ foldl (+)     (0+1)                       [2..5]
→ foldl (+)    ((0+1) + 2)                  [3..5]
→ foldl (+)   (((0+1) + 2) + 3)             [4..5]
→ foldl (+)  ((((0+1) + 2) + 3) + 4)        [5]
→ foldl (+) (((((0+1) + 2) + 3) + 4) + 5) []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



Space usage of summing a list of integers:

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

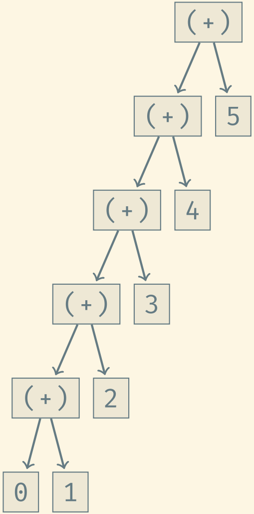
```
  foldl (+)      0                               [1..5]
→ foldl (+)     (0+1)                            [2..5]
→ foldl (+)    ((0+1) + 2)                       [3..5]
→ foldl (+)   (((0+1) + 2) + 3)                  [4..5]
→ foldl (+)  ((((0+1) + 2) + 3) + 4)             [5]
→ foldl (+) (((((0+1)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



Space usage of summing a list of integers:

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

```
  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)
```



Space usage of summing a list of integers:

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

SOME PITFALLS OF LAZINESS (4)

Space usage of summing a list of integers:

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

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

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
```

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
```

SOME PITFALLS OF LAZINESS (4)

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
→ foldl' (+)  6 [4..5]
```

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
→ foldl' (+)  6 [4..5]
→ foldl' (+) 10 [5]
```

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
→ foldl' (+)  6 [4..5]
→ foldl' (+) 10 [5]
→ foldl' (+) 15 []
```

SOME PITFALLS OF LAZINESS (4)

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
→ foldl' (+)  6 [4..5]
→ foldl' (+) 10 [5]
→ foldl' (+) 15 []
→            15
```

Space usage of summing a list of integers:

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

```
  foldl' (+)  0 [1..5]
→ foldl' (+)  1 [2..5]
→ foldl' (+)  3 [3..5]
→ foldl' (+)  6 [4..5]
→ foldl' (+) 10 [5]
→ foldl' (+) 15 []
→            15
```

Advantages 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.
- This makes understanding the code and formal reasoning about code correctness easier.

Advantages 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.
- This makes understanding the code and formal reasoning about code correctness easier.

The need fo side effects:

- Interactions with the real world require side effects. Without these interactions, why do we compute anything at all?

THE UNREALISTIC DREAM OF NO SIDE EFFECTS

Advantages 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.
- This makes understanding the code and formal reasoning about code correctness easier.

The need fo side effects:

- Interactions with the real world require side effects. Without these interactions, why do we compute anything at all?
- Storing state in data structures and updating these data structures destructively requires side effects. These updates can be emulated non-destructively with a logarithmic slow-down, but that may be unacceptable in some applications.

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

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

This is an action in the `IO` monad.

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

This is an action in the `IO` monad.

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

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

This is an action in the `IO` monad.

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.

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 ()`.

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

This is an action in the `IO` monad.

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.

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

This is an action in the `IO` monad.

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.

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

This is an action in the `IO` monad.

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 purely functional steps.

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

This is an action in the `IO` monad.

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 purely functional steps.
- The aim is to create a clear separation between steps that have side effects (and thus need to be expressed in some monad) and the steps that do not (and thus can be expressed using pure functions).

IO MONAD: EXAMPLE

```
database :: [(String, Int)]
database = [("Norbert", 44), ("Luca", 14), ("Mateo", 6)]

main :: IO ()
main = do name <- getLine
          if name == "quit"
            then return ()
            else putStrLn (msg name $ lookup name database)

  where msg name Nothing =
            "I don't know the age of " ++ name ++ "."
        msg name (Just age) =
            "The age of " ++ name ++ " is " ++ show age ++ "."
```

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

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

  fail   = error
  f >> g = f >>= const g

const :: a -> b -> a
const x _ = x
```

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

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

MONADS: EXAMPLES

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

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

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

MONADS: 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: 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"
```

Standard monadic composition of actions sure isn't pretty:

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

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 = do s1 <- getString
                           s2 <- getString
                           putStrLn $ "S1 = " ++ s1
                           putStrLn $ "S2 = " ++ s2
```

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 = do s1 <- getString
                           s2 <- getString
                           putStrLn $ "S1 = " ++ s1
                           putStrLn $ "S2 = " ++ s2
```

A preprocessing step translates this into the above form.

The second use of the `IO` monad is to provide mutable variables and arrays for when we can't do without them:

```
-- Create and initialize a mutable variable of type t
newIORef :: t -> IO (IORef t)

-- Read content of IORef
readIORef :: IORef t -> IO t

-- Update content of IORef
writeIORef :: IORef t -> t -> IO ()

-- Modify content of IORef by applying pure function
modifyIORef :: IORef t -> (t -> t) -> IO ()
```

```
-- Equivalents to array/listArray
newArray     :: Ix i => (i, i) -> e   -> IO (IOArray i e)
newArray_    :: Ix i => (i, i) ->        IO (IOArray i e)
newListArray :: Ix i => (i, i) -> [e] -> IO (IOArray i e)
```

MUTABLE ARRAYS

```
-- Equivalents to array/listArray
newArray     :: Ix i => (i, i) -> e   -> IO (IOArray i e)
newArray_    :: Ix i => (i, i) ->        IO (IOArray i e)
newListArray :: Ix i => (i, i) -> [e] -> IO (IOArray i e)

-- Reading (!) and writing (no pure equivalent)
readArray  :: Ix i => IOArray i e -> i      -> IO e
writeArray :: Ix i => IOArray i e -> i -> e -> IO ()
```

MUTABLE ARRAYS

```
-- Equivalents to array/listArray
newArray     :: Ix i => (i, i) -> e   -> IO (IOArray i e)
newArray_    :: Ix i => (i, i) ->        IO (IOArray i e)
newListArray :: Ix i => (i, i) -> [e] -> IO (IOArray i e)

-- Reading (!) and writing (no pure equivalent)
readArray  :: Ix i => IOArray i e -> i      -> IO e
writeArray :: Ix i => IOArray i e -> i -> e -> IO ()

-- Equivalents of elems/assocs
getElems  :: Ix i => IOArray i e -> IO [e]
getAssocs :: Ix i => IOArray i e -> IO [(i, e)]
```

```haskell
-- Equivalents to array/listArray
newArray     :: Ix i => (i, i) -> e   -> IO (IOArray i e)
newArray_    :: Ix i => (i, i) ->        IO (IOArray i e)
newListArray :: Ix i => (i, i) -> [e] -> IO (IOArray i e)

-- Reading (!) and writing (no pure equivalent)
readArray  :: Ix i => IOArray i e -> i      -> IO e
writeArray :: Ix i => IOArray i e -> i -> e -> IO ()

-- Equivalents of elems/assocs
getElems  :: Ix i => IOArray i e -> IO [e]
getAssocs :: Ix i => IOArray i e -> IO [(i, e)]

-- Turn immutable array into mutable one and vice versa
freeze :: Ix i => IOArray i e -> IO (  Array i e)
thaw   :: Ix i =>   Array i e -> IO (IOArray i e)
```

MUTABLE MEMORY IN PURE COMPUTATIONS?

The problem with `IORef`s and `IOArray`s is that any algorithm that uses them must live entirely in the `IO` monad.

What if we have a function without side effects whose efficient implementation needs mutable variables? We don't want to lift it into the `IO` monad.

MUTABLE MEMORY IN PURE COMPUTATIONS?

The problem with `IORefs` and `IOArrays` is that any algorithm that uses them must live entirely in the `IO` monad.

What if we have a function without side effects whose efficient implementation needs mutable variables? We don't want to lift it into the `IO` monad.

An illustrative (but bad) example:

```
sum :: [Int] -> IO Int
sum xs = do s <- newIORef 0
            mapM_ (add s) xs
            readIORef s
  where add s x = modifyIORef s (+ x)
```

THE STRICT STATE MONAD ST

The strict state monad `ST s` offers `STRef`s and `STArray`s.

`STArray`s have the same (overloaded) interface as `IOArray`s.

The equivalents of `newIORef`, `readIORef`, `writeIORef`, and `modifyIORef` are `newSTRef`, `readSTRef`, `writeSTRef`, and `modifySTRef`.

The strict state monad `ST s` offers `STRef`s and `STArray`s.

`STArray`s have the same (overloaded) interface as `IOArray`s.

The equivalents of `newIORef`, `readIORef`, `writeIORef`, and `modifyIORef` are `newSTRef`, `readSTRef`, `writeSTRef`, and `modifySTRef`.

Imperative summation in the `ST s` monad:

```
sum :: [Int] -> ST s Int
sum xs = do s <- newSTRef 0
            mapM_ (add s) xs
            readSTRef s
  where add s x = modifySTRef s (+ x)
```

The strict state monad `ST s` offers `STRef`s and `STArray`s.

`STArray`s have the same (overloaded) interface as `IOArray`s.

The equivalents of `newIORef`, `readIORef`, `writeIORef`, and `modifyIORef` are `newSTRef`, `readSTRef`, `writeSTRef`, and `modifySTRef`.

Imperative summation in the `ST s` monad:

```
sumM xs = do s <- newSTRef 0
             mapM_ (addM s) xs
             readSTRef s
addM s x = modifySTRef s (+ x)
```

The strict state monad `ST s` offers `STRef`s and `STArray`s.

`STArray`s have the same (overloaded) interface as `IOArray`s.

The equivalents of `newIORef`, `readIORef`, `writeIORef`, and `modifyIORef` are `newSTRef`, `readSTRef`, `writeSTRef`, and `modifySTRef`.

Imperative summation in the `ST s` monad:

```
sum :: [Int] -> Int
sum xs = runST (sumM xs)
  where sumM xs = do s <- newSTRef 0
                     mapM_ (addM s) xs
                     readSTRef s
        addM s x = modifySTRef s (+ x)
```

The strict state monad `ST s` offers `STRef`s and `STArray`s.

`STArray`s have the same (overloaded) interface as `IOArray`s.

The equivalents of `newIORef`, `readIORef`, `writeIORef`, and `modifyIORef` are `newSTRef`, `readSTRef`, `writeSTRef`, and `modifySTRef`.

Imperative summation in the `ST s` monad:

```
sum :: [Int] -> Int
sum xs = runST (sumM xs)
  where sumM xs = do s <- newSTRef 0
                     mapM_ (addM s) xs
                     readSTRef s
        addM s x = modifySTRef s (+ x)
```

```
runST :: (forall s . ST s t) -> t
```

MONADS FOR ELEGANT CONTROL FLOW (1)

Monads can be used in pure computations to express control flow more elegantly.

MONADS FOR ELEGANT CONTROL FLOW (1)

Monads can be used in pure computations to express control flow more elegantly.

Warm-up: Pure functions

- Pure functions with function composition form a monad!

Monads can be used in pure computations to express control flow more elegantly.

Warm-up: Pure functions

- Pure functions with function composition form a monad!

Drop the luggage: Computations with state

- Often, a set of functions share a common state that they manipulate.
- In an object-oriented language, we'd wrap them in an object.
- In Haskell, we can either explicitly pass the state around or use the `State` monad.

Computations that can fail:

- `Maybe` can be used to express success using `Just` and failure using `Nothing`.
- `Maybe` is also a monad that captures the logic: If any step in this function fails, the function fails.

MONADS FOR ELEGANT CONTROL FLOW (2)

Computations that can fail:

- `Maybe` can be used to express success using `Just` and failure using `Nothing`.
- `Maybe` is also a monad that captures the logic: If any step in this function fails, the function fails.

Searching for a solution:

- The list type is a monad.
- Intuition: A list of values represents all possible outcomes of a computation. The next step should try to continue with each of them.

MONADS FOR ELEGANT CONTROL FLOW (2)

Computations that can fail:

- `Maybe` can be used to express success using `Just` and failure using `Nothing`.
- `Maybe` is also a monad that captures the logic: If any step in this function fails, the function fails.

Searching for a solution:

- The list type is a monad.
- Intuition: A list of values represents all possible outcomes of a computation. The next step should try to continue with each of them.

Many more:

- `Reader`, `Writer`, ...
- Monad transformers allow us to stack monads on top of each other, e.g., computations with state that may fail.

PURE FUNCTIONS AS A MONAD

```
instance Monad Identity where
  return            = Identity
  Identity x >>= f = f x          -- f :: a -> Identity b
  _ >> g            = g
```

PURE FUNCTIONS AS A MONAD

```
instance Monad Identity where
  return           = Identity
  Identity x >>= f = f x          -- f :: a -> Identity b
  _ >> g           = g
```

- We need `Identity` as a container type to refer to in the instance definition. The logic, however, is that of pure function composition.

PURE FUNCTIONS AS A MONAD

```
instance Monad Identity where
  return              = Identity
  Identity x >>= f = f x          -- f :: a -> Identity b
  _ >> g              = g
```

- We need `Identity` as a container type to refer to in the instance definition. The logic, however, is that of pure function composition.
- We provide a custom implementation of `(>>)` to improve efficiency: In the expression `f >> g`, we discard `f`'s result and `f` has no side effects, so why run `f` at all.

PURE FUNCTIONS AS A MONAD

```
instance Monad Identity where
  return               = Identity
  Identity x >>= f = f x          -- f :: a -> Identity b
  _ >> g           = g
```

- We need `Identity` as a container type to refer to in the instance definition. The logic, however, is that of pure function composition.
- We provide a custom implementation of `(>>)` to improve efficiency: In the expression `f >> g`, we discard `f`'s result and `f` has no side effects, so why run `f` at all.
- The `Identity` monad may not seem very useful, but it can be used as the basis for constructing stacks of monads using monad transformers.

COMPUTATIONS WITH STATE (1)

Compute a random sequence from a seed:

```
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)

generateRandomNumberAndSeed :: Int -> (Int, Int)
generateRandomNumberAndSeed seed = ... -- Details unimportant for us
```

```
data State s t = State { runState :: s -> (t,s) }
```

```
data State s t = State { runState :: s -> (t,s) }

evalState :: State s t -> s -> t


execState :: State s t -> s -> t
```

```
data State s t = State { runState :: s -> (t,s) }

evalState :: State s t -> s -> t
evalState f s = fst (runState f s)

execState :: State s t -> s -> t
execState f s = snd (runState f s)
```

THE STATE MONAD

```
data State s t = State { runState :: s -> (t,s) }

evalState :: State s t -> s -> t
evalState f s = fst (runState f s)

execState :: State s t -> s -> t
execState f s = snd (runState f s)

instance Monad (State s) where
  return x = State $ \s -> (x,s)
```

THE STATE MONAD

```haskell
data State s t = State { runState :: s -> (t,s) }

evalState :: State s t -> s -> t
evalState f s = fst (runState f s)

execState :: State s t -> s -> t
execState f s = snd (runState f s)

instance Monad (State s) where
  return x = State $ \s -> (x,s)
  fail     = error
```

```
data State s t = State { runState :: s -> (t,s) }

evalState :: State s t -> s -> t
evalState f s = fst (runState f s)

execState :: State s t -> s -> t
execState f s = snd (runState f s)

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

ACCESSING THE CURRENT STATE

```
get :: State s s

put :: s -> State s ()

modify :: (s -> s) -> State s ()
```

ACCESSING THE CURRENT STATE

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

put :: s -> State s ()
put s = State $ const ((), s)

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

COMPUTATIONS WITH STATE (2)

```haskell
type Gen = State Int

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

genseq :: Int -> Gen [Int]
```

```
type Gen = State Int

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

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

gennum :: Gen Int
```

COMPUTATIONS WITH STATE (2)

```haskell
type Gen = State Int

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

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

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

COMPUTATIONS THAT CAN FAIL

```
step1 :: a -> Maybe b
step2 :: b -> Maybe c
step3 :: c -> Maybe d

-- Sequence steps 1-3
threeSteps :: a -> Maybe d
```

COMPUTATIONS THAT CAN FAIL

```
step1 :: a -> Maybe b
step2 :: b -> Maybe c
step3 :: c -> Maybe d

-- Sequence steps 1-3
threeSteps :: a -> Maybe d
threeSteps x = result3
  where result1 = step1 x
        result2 = maybe Nothing step2 result1
        result3 = maybe Nothing step3 result2
```

COMPUTATIONS THAT CAN FAIL

```
step1 :: a -> Maybe b
step2 :: b -> Maybe c
step3 :: c -> Maybe d

-- Sequence steps 1-3
threeSteps :: a -> Maybe d
threeSteps x = step1 x >>= step2 >>= step3
```

COMPUTATIONS THAT CAN FAIL

```
step1 :: a -> Maybe b
step2 :: b -> Maybe c
step3 :: c -> Maybe d

-- Sequence steps 1-3
threeSteps :: a -> Maybe d
threeSteps = step1 >=> step2 >=> step3

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

COMPUTATIONS THAT CAN FAIL

```
step1 :: a -> Maybe b
step2 :: b -> Maybe c
step3 :: c -> Maybe d

-- Sequence steps 1-3
threeSteps :: a -> Maybe d
threeSteps = step1 >=> step2 >=> step3

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

instance Monad Maybe where
  return  = Just
  fail    = const Nothing
  x >>= f = maybe Nothing f x
```

SEARCHING FOR A SOLUTION

Remember: A list ist interpreted as a collection of possible results of a computation.

Remember: A list ist interpreted as a collection of possible results of a computation.

Writing a number as a sum of non-decreasing positive numbers:

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

Remember: A list ist interpreted as a collection of possible results of a computation.

Writing a number as a sum of non-decreasing positive numbers:

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

```
nonDecreasingSplit :: Int -> [[Int]]
nonDecreasingSplit = split >=> splitRest
  where split x          = ... -- split x into two values y and z
        splitRest (y, z) = ... -- split z into zs so that
                               -- y:zs is non-decreasing
```

Remember: A list ist interpreted as a collection of possible results of a computation.

Writing a number as a sum of non-decreasing positive numbers:

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

```
nonDecreasingSplit :: Int -> [[Int]]
nonDecreasingSplit = split >=> splitRest
  where split x          = [(y, x-y) | y <- [1..x]]
        splitRest (y, z) = ... -- split z into zs so that
                               -- y:zs is non-decreasing
```

Remember: A list ist interpreted as a collection of possible results of a computation.

Writing a number as a sum of non-decreasing positive numbers:

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

```
nonDecreasingSplit :: Int -> [[Int]]
nonDecreasingSplit = split >=> splitRest
  where split x          = [(y, x-y) | y <- [1..x]]
        splitRest (y, 0) = return [y]
        splitRest (y, z) = nonDecreasingSplit z >>= extendWith y
        extendWith y zs = ... -- Prepend y to zs if
                              -- the result is non-decreasing
```

Remember: A list ist interpreted as a collection of possible results of a computation.

Writing a number as a sum of non-decreasing positive numbers:

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

```
nonDecreasingSplit :: Int -> [[Int]]
nonDecreasingSplit = split >=> splitRest
  where split x          = [(y, x-y) | y <- [1..x]]
        splitRest (y, 0) = return [y]
        splitRest (y, z) = nonDecreasingSplit z >>= extendWith y
        extendWith y zs@(z:_) | y <= z    = return (y:zs)
                              | otherwise = fail "Decreasing"
```

```
instance Monad [] where
  return x = [x]
  fail     = const []
  (>>=)    = flip concatMap

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs = concat (map f xs)
```

RESOURCES

- Lots of packages at `hackage.haskell.org`
- GHC documentation at `https://downloads.haskell.org/~ghc/latest/docs/html/`
- Hoogle at `www.haskell.org/hoogle`
- Books, tutorials, ... at `www.haskell.org/documentation`