Solve.hs

Module 1 Lecture Slides

Lecture 1 - Introduction and Pattern Matching

What You Will Learn

- Goals and Overview
- Defining a List
- Pattern Matching

- Course is divided into 4 modules
  - (2 for early release)
- **Module 1: Lists and Loop Patterns**
  - How does the List type work?
  - How do we implement core loop patterns?

What You Will Learn

- **Goals and Overview**
- Defining a List
- Pattern Matching

- Course is divided into 4 modules
  - (2 for early release)
- Module 1: Lists and Loop Patterns
  - How does the List type work?
  - How do we implement core loop patterns?
- **Module 2: Data Structures**
  - How are structures different in Haskell?
  - How do we apply them to solve problems?

What You Will Learn

- Goals and Overview
- Defining a List
- Pattern Matching

- Course is divided into 4 modules
  - (2 for early release)
- Module 1: Lists and Loop Patterns
  - How does the List type work?
  - How do we implement core loop patterns?
- Module 2: Data Structures
  - How are structures different in Haskell?
  - How do we apply them to solve problems?
- **Module 3: Essential Algorithms**
  - How do we implement common algorithms?
  - What patterns do we use to apply these in problems?

What You Will Learn

- **Goals and Overview**
- Defining a List
- Pattern Matching

- Course is divided into 4 modules
  - (2 for early release)
- Module 1: Lists and Loop Patterns
  - How does the List type work?
  - How do we implement core loop patterns?
- Module 2: Data Structures
  - How are structures different in Haskell?
  - How do we apply them to solve problems?
- Module 3: Essential Algorithms
  - How do we implement common algorithms?
  - What patterns do we use to apply these in problems?
- **Module 4: Parsing**
  - How does a Haskell parser work under the hood?
  - What useful libraries exist for parsing?

Core Principles

- Goals and Overview
- Defining a List
- Pattern Matching

- **Explicitly Enumerate Patterns**
  - (and compare patterns to other languages)
- **Implement From Scratch**
  - Better understanding of how things work
  - Intuitively apply ideas
- **Lots of Practice**
  - Practice problems help solidify ideas.
  - This course has lots of them!

What you *Won't* Learn

- Goals and Overview
- Defining a List
- Pattern Matching

- **Haskell Toolchain Setup**
    - See **Setup.hs** (free course)
    - https://academy.mondaymorninghaskell.com/p/setup-hs
- **Basic Haskell Syntax**
    - See Haskell From Scratch
        - https://academy.mondaymorninghaskell.com/p/haskell-from-scratch
    - Or Liftoff Series
        - https://mmhaskell.com/liftoff
- **Haskell Language Concepts**
    - e.g. Monads, Laziness, Immutability
    - We'll touch on these concepts when they're useful
    - But we won't learn about them from scratch
    - https://mmhaskell.com/monads
- Not great if you've *never* written Haskell before
- But still good for most **beginners** and **intermediates**.

Course Materials

- Goals and Overview
- Defining a List
- Pattern Matching

- **Lectures**
  - We'll go over conceptual concepts with accompanying slides and simple examples
- **Exercises**
  - The exercise instruction document and code zip file are downloadable attachments to this lecture.
  - Read the instructions to learn how to use the code.
  - Solve the problems to practice your skills from the lecture!
- **Solutions**
  - In the **final** lecture for each module, you'll find another zip file with my code solutions for each of the practice problems.

Comment on AI Usage

- **Goals and Overview**
- Defining a List
- Pattern Matching

- **This course is the result of my own manual effort, not generative AI**
- ~~"Chat GPT, write a Haskell course for me"~~
- Everything is "hand" written over months of labor
    - Course outline
    - Lecture slides and narrations
    - Practice problem descriptions, code, and test cases
- I *may* use AI in the future to help come up with practice problem **ideas**
    - (But have not done so for Modules 1 & 2)

Module 1 Overview

- Goals and Overview
- Defining a List
- Pattern Matching

- Loop Patterns
  - While loops, for loops, etc.
- Haskell handles these using **recursion**
  - Many helper functions exist
  - (which use recursion under the hood)
- Lists
  - A fundamentally recursive data structure
  - You'll build a version from scratch
- List API
  - Explore all the different helper functions

Defining our own List Type

- Goals and Overview
- **Defining a List**
- Pattern Matching

```
data MyList a =
  Nil |
  Cons a (MyList a)
```

Defining our own List Type

- Goals and Overview
- **Defining a List**
- Pattern Matching

```
data MyList a =
  Nil |
  Cons a (MyList a)

empty :: MyList Int
empty = Nil

single :: MyList Int
single = Cons 5 Nil

triple :: MyList Int
triple = Cons 1 (Cons 2 (Cons 3 Nil))
```

Pattern Matching Constructors

- Goals and Overview
- Defining a List
- **Pattern Matching**

```haskell
-- Our List Type
listFunction :: MyList Int -> Int
listFunction Nil = 0
listFunction (Cons x rest) = x + 1
```

Pattern Matching Constructors

- Goals and Overview
- Defining a List
- **Pattern Matching**

```haskell
-- Our List Type
listFunction :: MyList Int -> Int
listFunction Nil = 0
listFunction (Cons x rest) = x + 1

-- Maybe
maybeFunction :: Maybe Int -> Int
maybeFunction Nothing = 0
maybeFunction (Just x) = x + 1
```

Pattern Matching Constructors

- Goals and Overview
- Defining a List
- **Pattern Matching**

```haskell
-- Our List Type
listFunction :: MyList Int -> Int
listFunction Nil = 0
listFunction (Cons x rest) = x + 1

-- Maybe
maybeFunction :: Maybe Int -> Int
maybeFunction Nothing = 0
maybeFunction (Just x) = x + 1

-- A different constructed type
data MyType =
  Point Int Int |
  Radius Int |
  Message String

myTypeFunction :: MyType -> Int
myTypeFunction (Point x y) = x + y
myTypeFunction (Radius x) = x + 1
myTypeFunction (Message s) = length s
```

Pattern Matching Constructors

- Goals and Overview
- Defining a List
- **Pattern Matching**

```haskell
listFunction :: MyList Int -> Int
listFunction myList = case myList of
  Nil -> 0
  (Cons x rest) -> x + 1

maybeFunction :: Maybe Int -> Int
maybeFunction val = case val of
  Nothing -> 0
 (Just x) -> x + 1

data MyType =
  Point Int Int |
  Radius Int |
  Message String

myTypeFunction :: MyType -> Int
myTypeFunction myType = case myType of
  (Point x y) -> x + y
  (Radius x) -> x + 1
  (Message s) -> length s
```

Pattern Matching Values

- Goals and Overview
- Defining a List
- **Pattern Matching**

```haskell
-- Numbers
numberFunction :: Int -> Int
numberFunction 0 = 5
numberFunction 1 = 10
numberFunction x = 3 * x

-- Strings
stringFunction :: String -> Int
stringFunction "Hello" = 0
stringFunction s = length s

-- Tuples
tupleFunction :: (Int, Int) -> Int
tupleFunction (5, y) = y * 2
tupleFunction (6, y) = y * 5
tupleFunction (x, 6) = x + 3
tupleFunction (x, y) = x + y
```

Pattern Matching Values

- Goals and Overview
- Defining a List
- **Pattern Matching**

```
-- Haskell Base List
listFunction :: [Int] -> Int
listFunction [] = -3
listFunction [5] = 2
listFunction (3 : rest) = 2 * length rest
listFunction [x] = x + 2
listFunction (x : y : z : _) = x + y + z
listFunction _ = 0
```

Lecture 2 - Recursion Basics

What is a `while` Loop?

- **While Loops**
- Recursion
- 4 Steps

- Code that repeats until a condition is no longer met
  - (or a `break` is encountered)

```
bool keepGoing = true;
while (keepGoing) {
  keepGoing = doSomething(...);
}
```

- More fundamental than a `for` loop
  - (For loops become while loops at compile time)

```
for (int i = 0; i < 5; ++i) {
  doSomething(...);
}
```

While Loop Examples

- **While Loops**
- Recursion
- 4 Steps

```
uint8_t modulo(uint8_t input, uint8_t mod) {
  while (input >= mod) {
    input -= mod;
  }
  return input;
}
```

While Loop Examples

- **While Loops**
- Recursion
- 4 Steps

```
uint8_t modulo(uint8_t input, uint8_t mod) {
  if (mod == 0) {
    return input; // Or throw error
  }
  while (input >= mod) {
    input -= mod;
  }
  return input;
}
```

While Loop Examples

- **While Loops**
- Recursion
- 4 Steps

```
int last(std::list<int> inputs) {
  auto first = inputs.begin();
  if (first == inputs.end()) {
    return 0; // Or throw error
  }
  auto next = ++(inputs.begin());

  while (next != inputs.end()) {
    ++first;
    ++next;
  }
  return *first;
}
```

What is Recursion?

- While Loops
- **Recursion**
- 4 Steps

- A function **calling itself**
  - Within foo() there is a call to foo()
- Must do this with a different input
  - (Or it will loop infinitely)
- Must have a "base case" that is the terminal condition
  - Acts like the loop condition
- Inputs to recursive call must get us closer to base case

Writing Recursive Solutions

- While Loops
- **Recursion**
- 4 Steps

```
uint8_t modulo(uint8_t input, uint8_t mod) {
  if (input < mod || mod == 0) {
    return input;
  } else {
    return modulo(input - mod, mod);
  }
}
```

Writing Recursive Solutions

- While Loops
- **Recursion**
- 4 Steps

```
using namespace std;
int last(list<int> inputs) {
  auto first = inputs.begin();
  if (first == inputs.end()) {
    return 0; // Or throw error
  }
  auto next = ++(inputs.begin());
  return lastT(inputs, first, next);
}

int lastT(list<int> inputs,
          list<int>::iterator first,
          list<int>::iterator next
          ) {
  if (next == inputs.end()) {
    return *first;
  } else {
    return lastT(inputs, ++first, ++next);
  }
}
```

What Changed?

- While Loops
- **Recursion**
- 4 Steps

- Instead of repeating a block of code, the function itself is the block that is repeating.
- This gives less opportunity for mutating state within the loop.
- In a recursive *function*, any stateful values must be **inputs** and **outputs** of the function.
- (This is why Haskell prefers recursion)

```
int a = 0;
int b = 5;
bool c = True;
while (c) {
  c = doSomething(a)
  ++a;
  b += 2;
}
```

Haskell Solutions

- While Loops
- **Recursion**
- 4 Steps

```
modulo :: Word -> Word -> Word
modulo input mod = if input < mod || mod == 0
  then input
  else modulo (input - mod) mod


last :: [Int] -> Int
last [] = error "No last on empty list!"
last [x] = x
last (_ : xs) = last xs
```

4 Steps to Recursion

- While Loops
- Recursion
- **4 Steps**

1. Define the Base Case

4 Steps to Recursion

- While Loops
- Recursion
- **4 Steps**

1. Define the Base Case
2. Separate one "piece" from the general case and "solve it"

4 Steps to Recursion

- While Loops
- Recursion
- **4 Steps**

1. Define the Base Case
2. Separate one "piece" from the general case and "solve it"
3. Call recursive function on remaining piece

4 Steps to Recursion

- While Loops
- Recursion
- **4 Steps**

1. Define the Base Case
2. Separate one "piece" from the general case and "solve it"
3. Call recursive function on remaining piece
4. Combine recursive answer with remainder
    a. (We'll skip this for this lecture)

4 Steps to Recursion

- While Loops
- Recursion
- **4 Steps**

1. **Define the Base Case**
2. **Separate one "piece" from the general case and "solve it"**
3. **Call recursive function on remaining piece**
4. **Combine recursive answer with remainder**

Slow Motion Solutions

- While Loops
- Recursion
- 4 Steps

```
modulo :: Word -> Word -> Word
modulo input mod = if input < mod || mod == 0
  then input
  ...
```

Slow Motion Solutions

- While Loops
- Recursion
- 4 Steps

```
modulo :: Word -> Word -> Word
modulo input mod = if input < mod || mod == 0
  then input
  else
    let nextInput = input - mod
    in ...
```

Slow Motion Solutions

- While Loops
- Recursion
- 4 Steps

```
modulo :: Word -> Word -> Word
modulo input mod = if input < mod || mod == 0
  then input
  else
    let nextInput = input - mod
    in modulo nextInput mod
```

Slow Motion Solutions

- While Loops
- Recursion
- 4 Steps

```
last :: [Int] -> Int
last [] = error "No last on empty list!"
last [x] = x
...
```

Slow Motion Solutions

- While Loops
- Recursion
- 4 Steps

```
last :: [Int] -> Int
last [] = error "No last on empty list!"
last [x] = x
last (_ : xs) = ...
```

Slow Motion Solutions

- While Loops
- Recursion
- **4 Steps**

```
last :: [Int] -> Int
last [] = error "No last on empty list!"
last [x] = x
last (_ : xs) = last xs
```

Lecture 3 - Recursion with Accumulation

The 4th Step

- **Accumulation Basics**
- Examples

1. **Define the Base Case**
2. **Separate one "piece" from the general case and "solve it"**
3. **Call recursive function on remaining piece**
4. **Combine recursive answer with remainder**

The 4th Step

- **Accumulation Basics**
- Examples

1. Define the Base Case
2. Separate one "piece" from the general case and "solve it"
3. Call recursive function on remaining piece
4. **Combine recursive answer with remainder**

Factorial (C++)

- Accumulation Basics
- Examples

```
long factorial(int n) {
  long result = 1;
  int i = n;
  while (i > 0) {
    result *= i;
    --i;
  }
}
```

Factorial (C++)

- Accumulation Basics
- Examples

```
long factorial(uint8 n) {
  if (n <= 1) {
    return 1;
  } else {
    uint8 i = n - 1;
    long recurse = factorial(i);
    return n * recurse;
  }
}
```

Factorial (C++)

- Accumulation Basics
- Examples

```
long factorial(uint8 n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Factorial (Haskell)

- Accumulation Basics
- Examples

```haskell
factorial :: Word -> Integer
factorial 0 = 1
factorial 1 = 1
factorial n =
  let i = n - 1
      recurse = factorial i
  in  n * recurse
```

Factorial (Haskell)

- Accumulation Basics
- **Examples**

```
factorial :: Word -> Integer
factorial 0 = 1
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

Minimum

- Accumulation Basics
- **Examples**

```
minimum :: [Int] -> Int
minimum [] = error "Can't take min of empty"
minimum [x] = x
minimum (x : xs) = ...
```

Minimum

- Accumulation Basics
- **Examples**

```
minimum :: [Int] -> Int
minimum [] = error "Can't take min of empty"
minimum [x] = x
minimum (x : xs) =
  let recurse = minimum xs
  in  ...
```

Minimum

- Accumulation Basics
- Examples

```
minimum :: [Int] -> Int
minimum [] = error "Can't take min of empty"
minimum [x] = x
minimum (x : xs) =
  let recurse = minimum xs
  in  min x recurse
```

Largest Triplet

- Accumulation Basics
- **Examples**

```
-- Find three consecutive numbers with
-- largest sum
largestTriplet :: [Word] -> (Word, Word, Word)
largestTriplet = ...
```

Largest Triplet

- Accumulation Basics
- Examples

```haskell
-- Find three consecutive numbers with
-- largest sum
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet [x, y, z] = (x, y, z)
largestTriplet (x : y : z : r) = ...
largestTriplet _ = error "Not enough ints!"
```

Largest Triplet

- Accumulation Basics
- Examples

```haskell
-- Find three consecutive numbers with
-- largest sum
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet [x, y, z] = (x, y, z)
largestTriplet (x : y : z : r) =
  let (a, b, c) = largestTriplet (y : z : r)
  ...
largestTriplet _ = error "Not enough ints!"
```

Largest Triplet

- Accumulation Basics
- Examples

```haskell
-- Find three consecutive numbers with
-- largest sum
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet [x, y, z] = (x, y, z)
largestTriplet (x : y : z : r) =
  let (a, b, c) = largestTriplet (y : z : r)
  in  if a + b + c > x + y + z
        then (a, b, c)
        else (x, y, z)
largestTriplet _ = error "Not enough ints!"
```

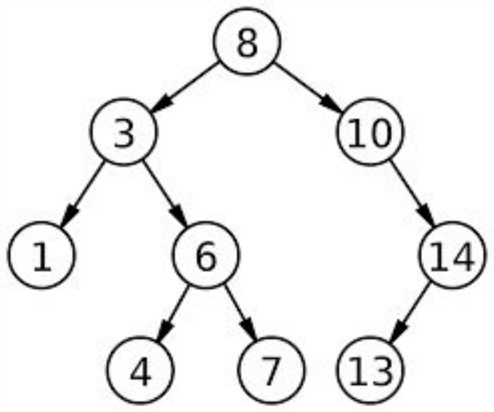
Binary Tree Traversal

- Accumulation Basics
- **Examples**

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder = ...

preOrder :: Tree -> [Int]
preOrder = ...

postOrder :: Tree -> [Int]
postOrder = ...
```

Binary Tree Traversal

- Accumulation Basics
- **Examples**




```
inOrder = [1,3,4,6,7,8,10,13,14]

preOrder = [8,3,1,6,4,7,10,14,13]

postOrder = [1,4,7,6,3,13,14,10,8]
```

Binary Tree Traversal

- Accumulation Basics
- **Examples**

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder Nil = []
...
```

Binary Tree Traversal

- Accumulation Basics
- Examples

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder Nil = []
inOrder (Node x left right) = ...
```

Binary Tree Traversal

- Accumulation Basics
- Examples

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder Nil = []
inOrder (Node x left right) =
  let left' = inOrder left
      right' = inOrder right
  ...
```

Binary Tree Traversal

- Accumulation Basics
- Examples

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder Nil = []
inOrder (Node x left right) =
  let left' = inOrder left
      right' = inOrder right
  in  left' ++ [x] ++ right'
```

Binary Tree Traversal

- Accumulation Basics
- **Examples**

```haskell
inOrder :: Tree -> [Int]
inOrder Nil = []
inOrder (Node x left right) =
  let left' = inOrder left
      right' = inOrder right
  in  left' ++ [x] ++ right'

preOrder :: Tree -> [Int]
preOrder Nil = []
preOrder (Node x left right) =
  let left' = preOrder left
      right' = preOrder right
  in  x : left' ++ right'

postOrder :: Tree -> [Int]
postOrder Nil = []
postOrder (Node x left right) =
  let left' = postOrder left
      right' = postOrder right
  in  left' ++ right' ++ [x]
```

Lecture 4 - Tail Recursion

Recursion's Weakness

- **Stack Memory**
- Tail Call Optimization
- How to Accumulate
- Examples

```
-- Iterative Solution
-- Time: ??? Space: ???
long factorial(uint8 n) {
  long result = 1;
  int i = n;
  while (i > 0) {
    result *= i;
    --i;
  }
}

-- Recursive Solution
-- Time: ??? Space: ???
long factorial(uint8 n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

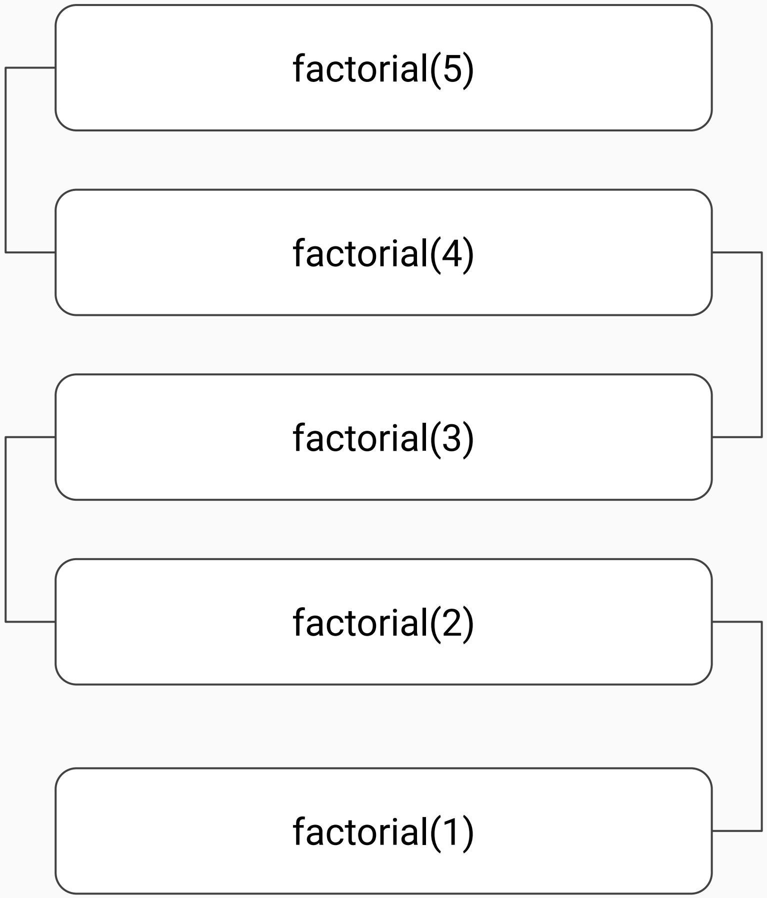
Recursion's Weakness

- **Stack Memory**
- Tail Call Optimization
- How to Accumulate
- Examples

```
-- Iterative Solution
-- Time: O(n) Space: O(1)
long factorial(uint8 n) {
  long result = 1;
  int i = n;
  while (i > 0) {
    result *= i;
    --i;
  }
}

-- Recursive Solution
-- Time: O(n) Space: O(n)
long factorial(uint8 n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Stack Memory Usage

- **Stack Memory**
- Tail Call Optimization
- How to Accumulate
- Examples



Tail Call Optimization

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- Examples

- If the **last** thing a function does is make a recursive call...
- GHC will optimize the stack frame usage.
- Program will re-use the stack frame for the recursive call.
- Many functional languages use this
  - Haskell, Elm, Erlang, OCaml, PureScript, Scala
- Common languages often don't
  - (or require compiler flags or extra mechanics)
  - (or is spotty depending on version)
  - Python, Go, Rust, Javascript

A Simple TCO Example

- Stack Memory
- **Tail Call Optimization**
- How to Accumulate
- Examples

```haskell
last :: [Int] -> Int
last [] = error "No last on empty list!"
last [x] = x
-- Uses Tail Call Optimization
-- No work except recursive call
last (_ : xs) = last xs
```

How to Accumulate?

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```
factorial :: Word -> Int64
factorial 0 = 1
factorial 1 = 1
-- How to optimize?
factorial n = n * factorial (n - 1)
```

Extra Arguments

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

- Uses helper functions
  - (Real recursion will take place here)
- Add extra **accumulation argument(s)** to helpers
  - Step 4 will happen with argument
- Top level function will call helper with baseline starters
- Helper is "tail recursive"

Factorial

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```
factorial :: Word -> Int64
factorial x = factorialTail 1 x
  where
    -- Extra Int64 is accumulator
    factorialTail :: Int64 -> Word -> Int64
    factorialTail accum n = ...
```

Factorial

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```
factorial :: Word -> Int64
factorial x = factorialTail 1 x
  where
    -- Extra Int64 is accumulator
    factorialTail :: Int64 -> Word -> Int64
    factorialTail accum n = if n <= 1
      then accum
      else ...
```

Factorial

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```
factorial :: Word -> Int64
factorial x = factorialTail 1 x
  where
    -- Extra Int64 is accumulator
    factorialTail :: Int64 -> Word -> Int64
    factorialTail accum n = if n <= 1
      then accum
      else
        let i = n - 1
        in  ...
```

Factorial

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```haskell
factorial :: Word -> Int64
factorial x = factorialTail 1 x
  where
    -- Extra Int64 is accumulator
    factorialTail :: Int64 -> Word -> Int64
    factorialTail accum n = if n <= 1
      then accum
      else
        let i = n - 1
        in  factorialTail (...) i
```

Factorial

- Stack Memory
- Tail Call Optimization
- **How to Accumulate**
- Examples

```haskell
factorial :: Word -> Int64
factorial x = factorialTail 1 x
  where
    -- Extra Int64 is accumulator
    factorialTail :: Int64 -> Word -> Int64
    factorialTail accum n = if n <= 1
      then accum
      else
        let i = n - 1
        in  factorialTail (n * accum) i
```

Minimum

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
minimum :: [Int] -> Int
minimum [] = error "No elements"
minimum (x : xs) = minimumTail x xs
  where
     -- First arg is "current min"
     minimumTail :: Int -> [Int] -> Int
     minimumTail = ...
```

Minimum

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```haskell
minimum :: [Int] -> Int
minimum [] = error "No elements"
minimum (x : xs) = minimumTail x xs
  where
    -- First arg is "current min"
    minimumTail :: Int -> [Int] -> Int
    minimumTail current [] = current
    ...
```

Minimum

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
minimum :: [Int] -> Int
minimum [] = error "No elements"
minimum (x : xs) = minimumTail x xs
  where
     -- First arg is "current min"
     minimumTail :: Int -> [Int] -> Int
     minimumTail current [] = current
     minimumTail current (n : ns) =
       minimumTail (min current n) ns
```

Minimum

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
minimum :: (Bounded a, Ord a) => [a] -> a
minimum = minimumTail maxBound
  where
    minimumTail current [] = current
    minimumTail current (x : xs) =
      minimumTail (min current x) xs
```

Largest Triplet

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> error "Not enough ints!"
  where
    f = ...
```

Largest Triplet

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> error "Not enough ints!"
  where
    f (x, y, z) (a : b : c : r) = ...
    f (x, y, z) _ = (x, y, z)
```

Largest Triplet

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> error "Not enough ints!"
  where
    f (x, y, z) (a : b : c : r) =
      if a + b + c > x + y + z
        then f (a, b, c) (b : c : r)
        else f (x, y, z) (b : c : r)
    f (x, y, z) _ = (x, y, z)
```

Can't Always Avoid Stack Memory!

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

```
data Tree =
  Nil |
  Node Int Tree Tree


inOrder :: Tree -> [Int]
inOrder Nil = []
inOrder (Node x left right) =
  -- Uses 2 recursive calls!
  -- Difficult to Tail-Call Optimize!
  let left' = inOrder left
      right' = inOrder right
  in  left' ++ [x] ++ right'
```

Helper Function Hints

- Stack Memory
- Tail Call Optimization
- How to Accumulate
- **Examples**

- Don't focus too hard on naming
  - Either functionName**Tail**
  - Or just use **f, g, h**, etc.
- Use accumulator as *first* argument.
  - Allows eta reduction
  - Partial function application
- For tail function used in multiple places
  - (but not part of public API)
  - Just use accumulator arguments on topline definition

Lecture 5 - List Accumulation

List Accumulation

- One pattern so far
  - While loops
  - Tail-call optimized recursion
- But only accumulating simple values
  - E.g. numbers
- **List accumulation** is very common and important
  - One trick to do this efficiently in Haskell

Python For Loops

- **List Accumulation**
- Tail Reversing
- Map Pattern
- Filter Pattern

```
def multPairs(input1: [int], input2: [int]):
  result = []
  i = 0
  while i < len(input1) and i < len(input2):
    result.append(input1[i] * input2[i])
  return result
```

Naive Haskell List Accumulation

- List Accumulation
- **Tail Reversing**
- Map Pattern
- Filter Pattern

```haskell
multPairs :: [Int] -> [Int] -> [Int]
multPairs = f []
  where
    f accum [] _ = accum
    f accum _ [] = accum
    f accum (x : xs) (y : ys) =
      -- This accumulation is inefficient!
      -- Each step is O(n)!
      f (accum ++ [x * y]) xs ys
```

Improved Haskell List Accumulation

- List Accumulation
- **Tail Reversing**
- Map Pattern
- Filter Pattern

```
multPairs :: [Int] -> [Int] -> [Int]
multPairs = f []
  where
    f accum [] _ = ???
    f accum _ [] = ???
    f accum (x : xs) (y : ys) =
      -- This is efficient! (O(1))
      f ((x * y) : accum) xs ys
```

Improved Haskell List Accumulation

- List Accumulation
- **Tail Reversing**
- Map Pattern
- Filter Pattern

```haskell
multPairs :: [Int] -> [Int] -> [Int]
multPairs = f []
  where
    f accum [] _ = reverse accum
    f accum _ [] = reverse accum
    f accum (x : xs) (y : ys) =
      -- This is efficient! (O(1))
      f ((x * y) : accum) xs ys
```

What is the Map Pattern?

- List Accumulation
- Tail Reversing
- **Map Pattern**
- Filter Pattern

- "Map" Pattern
- Create a new list
- For each input item, apply a function
    - Place in output list
- Applies to more than just lists
    - (Functor pattern)

Basic Python

- List Accumulation
- Tail Reversing
- **Map Pattern**
- Filter Pattern

```
def doubleAll(input: [int]):
  result = []
  for i in input:
    result.append(2 * i)
  return result

# Alternatively...
doubled = map(lambda i: return 2 * i, input)
```

Haskell Implementation

- List Accumulation
- Tail Reversing
- **Map Pattern**
- Filter Pattern

```
map :: (a -> b) -> [a] -> [b]
map f = map' []
  where
    map' acc [] = reverse acc
    map' acc (x : xs) = map' (f x : acc) xs
```

Using Recursion for Map

- List Accumulation
- Tail Reversing
- **Map Pattern**
- Filter Pattern

```
doubleAll :: [Int] -> [Int]
doubleAll = f []
  where
    f acc [] = reverse acc
    f acc (x : xs) = f (2 * x : acc) xs
```

Improved Map

- List Accumulation
- Tail Reversing
- **Map Pattern**
- Filter Pattern

```
doubleAll :: [Int] -> [Int]
doubleAll = map (2 *)
```

What is the Filter Pattern?

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

- "Filter" pattern
- Select a subset of elements satisfying some criteria
- Very common pattern in problem solving

Filter in Python

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

```
def onlyDiv3(input: [int]):
  result = []
  for i in input:
    if i % 3 == 0:
      result.append(i)
  return result

# Alternatively...
filtered = filter(
  lambda i: return (i % 3) == 0,
  input
)
```

Using Recursion for Filter

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

```
onlyDiv3 :: [Int] -> [Int]
onlyDiv3 = f []
  where
    f acc [] = reverse acc
    f acc (x : xs) = if x `mod` 3 == 0
      then filter' (x : acc) xs
      else filter' acc xs
```

Improved Filter

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

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

onlyDiv3 :: [Int] -> [Int]
onlyDiv3 = filter (\i -> i `mod` 3 == 0)
```

Combining Map and Filter

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

```
add5ToDiv3 :: [Int] -> [Int]
add5ToDiv3 xs =
  map (+5) $ filter (\i -> i `mod` 3 == 0) xs
```

Combining Map and Filter

- List Accumulation
- Tail Reversing
- Map Pattern
- **Filter Pattern**

```
add5ToDiv3 :: [Int] -> [Int]
add5ToDiv3 =
  map (+5) . filter (\i -> i `mod` 3 == 0)
```

Lecture 6 - Folds

Loop Patterns So Far

- **What is a Fold?**
- Fold Types
- Steps of a Fold
- Examples
- Short-Circuiting

- While loops
  - (with accumulation)
- For-each loops
  - (no side effects)
- (Tail) Recursion
- Map, Filter

What is a Fold?

- For-each loop that **accumulates a stateful value**
- Starts with an initial value
- Each list item "updates" the value
  - "Accumulator function"
  - Takes previous value and list item
  - Produces new value
- Always goes through **entire** list
  - (No short-circuiting)

Basic Type Signature

- What is a Fold?
- **Fold Types**
- Steps of a Fold
- Examples
- Short-Circuiting

```
foldl ::
   (val -> item -> val) -> -- Accumulator
   val                  -> -- Initial State
   [item]               -> -- List
   val                     -- Result

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

Basic Type Signature

- What is a Fold?
- **Fold Types**
- Steps of a Fold
- Examples
- Short-Circuiting

```
foldl ::
   (val -> item -> val) -> -- Accumulator
   val                  -> -- Initial State
   [item]               -> -- List
   val                     -- Result

>> foldl (+) 5 [6, 2, 3]
16 -- (5 + 6 + 2 + 3)
>> foldl (&&) True [True, False, True]
False -- True && True && False && True
```

Other Folds

- What is a Fold?
- **Fold Types**
- Steps of a Fold
- Examples
- Short-Circuiting

```haskell
-- Acts strictly, useful for performance
foldl' ::
  (val -> item -> val) -> -- Accumulator
  val                  -> -- Initial State
  [item]               -> -- List
  val                     -- Result

-- Starts from RIGHT
foldr ::
  (item -> val -> val) -> -- Accumulator
  val                  -> -- Initial State
  [item]               -> -- List
  val                     -- Result

>> foldr (+) 5 [6, 2, 3]
16 -- (6 + (2 + (3 + 5)))
>> foldl (-) 16 [6, 2, 3]
5 -- (((16 - 6) - 2) - 3)
>> foldr (-) 16 [6, 2, 3]
-9 -- (6 - (2 - (3 - 16)))
```

Other Folds

- What is a Fold?
- **Fold Types**
- Steps of a Fold
- Examples
- Short-Circuiting

```
-- Acts strictly, useful for performance
foldl' ::
  (val -> item -> val) -> -- Accumulator
  val                  -> -- Initial State
  [item]               -> -- List
  val                     -- Result

-- Starts from RIGHT
foldr ::
  (item -> val -> val) -> -- Accumulator
  val                  -> -- Initial State
  [item]               -> -- List
  val                     -- Result

>> foldr (+) 5 [6, 2, 3]
16 -- (6 + (2 + (3 + 5)))
>> foldl (-) 16 [6, 2, 3]
5 -- (((16 - 6) - 2) - 3)
>> foldr (flip (-)) 16 [6, 2, 3]
5 -- (((16 - 3) - 2) - 6)
```

Steps of a Fold

- What is a Fold?
- Fold Types
- **Steps of a Fold**
- Examples
- Short-Circuiting

1. **Identify the stateful value type**
2. **Write the accumulator function**
3. **Determine the initial value**
4. **Determine the input list**
5. **Postprocess**

Steps of a Fold

- What is a Fold?
- Fold Types
- **Steps of a Fold**
- Examples
- Short-Circuiting

1. **Identify the stateful value type**

Steps of a Fold

- What is a Fold?
- Fold Types
- **Steps of a Fold**
- Examples
- Short-Circuiting

1. **Identify the stateful value type**
2. **Write the accumulator function**

Steps of a Fold

1. **Identify the stateful value type**
2. **Write the accumulator function**
3. **Determine the initial value**

Steps of a Fold

- What is a Fold?
- Fold Types
- **Steps of a Fold**
- Examples
- Short-Circuiting

1. **Identify the stateful value type**
2. **Write the accumulator function**
3. **Determine the initial value**
4. **Determine the input list**

Steps of a Fold

- What is a Fold?
- Fold Types
- **Steps of a Fold**
- Examples
- Short-Circuiting

1. **Identify the stateful value type**
2. **Write the accumulator function**
3. **Determine the initial value**
4. **Determine the input list**
5. **Postprocess**

Factorial with Fold

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

1. Identify the stateful value type
2. **Write the accumulator function**
3. Determine the initial value
4. Determine the input list
5. Postprocess

```
factorial :: Int -> Int
factorial n = ...
```

Factorial with Fold

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

1. **Identify the stateful value type**
2. **Write the accumulator function**
3. **Determine the initial value**
4. **Determine the input list**
5. **Postprocess**

```
factorial :: Int -> Int
factorial n = ...
  -- Accumulate Int
  -- Acc function is (*)
  -- Initial value is 1 (multiplication)
  -- List is range from 1 to n
  -- No post-processing
```

Factorial with Fold

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

1. Identify the stateful value type
2. Write the accumulator function
3. Determine the initial value
4. Determine the input list
5. Postprocess

```
factorial :: Int -> Int
factorial n = foldl (*) 1 [1..n]
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```python
# Start with the first element in the list,
# then find each element that flips from
# even to odd
# [1,3,5,6,8,9,10] -> [1,6,9,10]

def altEvenOdds(inputs: [int]):
  if not inputs:
    return []
  first = inputs[0]
  prevIsEven = first % 2 == 0
  results = [first]
  for i in inputs[1:]:
    thisIsEven = i % 2 == 0
    if thisIsEven != prevIsEven:
      results.append(i)
      prevIsEven = thisIsEven
  return results
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```
altEvenOdds :: [Int] -> [Int]
altEvenOdds inputs = ...
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```
-- Step 1: Identify accumulated type
altEvenOdds :: [Int] -> [Int]
altEvenOdds inputs =
  ??? $ foldl f ??? ???
  where
    -- Value is:
    --   prevIsEven value (Bool)
    --   Accumulated result list ([Int])
    f :: (Bool, [Int]) -> Int -> (Bool, [Int])
    f = ...
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```haskell
-- Step 2: Write accumulator function
altEvenOdds :: [Int] -> [Int]
altEvenOdds inputs =
  ??? $ foldl f ??? ???
  where
    f :: (Bool, [Int]) -> Int -> (Bool, [Int])
    f (prevIsEven, acc) x =
      let thisIsEven = x `mod` 2 == 0
      in  if thisIsEven /= prevIsEven
            then (thisIsEven, x : acc)
            else (prevIsEven, acc)
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```
-- Step 3: Determine initial value
altEvenOdds :: [Int] -> [Int]
altEvenOdds [] = []
altEvenOdds (x : xs) =
  ??? $ foldl f (x `mod` 2 == 0, [x]) ???
  where
    f :: (Bool, [Int]) -> Int -> (Bool, [Int])
    f (prevIsEven, acc) x =
      let thisIsEven = x `mod` 2 == 0
      in  if thisIsEven /= prevIsEven
            then (thisIsEven, x : acc)
            else (prevIsEven, acc)
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```
-- Step 4: Determine the input list
altEvenOdds :: [Int] -> [Int]
altEvenOdds [] = []
altEvenOdds (x : xs) =
  ??? $ foldl f (x `mod` 2 == 0, [x]) xs
  where
    f :: (Bool, [Int]) -> Int -> (Bool, [Int])
    f (prevIsEven, acc) x =
      let thisIsEven = x `mod` 2 == 0
      in  if thisIsEven /= prevIsEven
            then (thisIsEven, x : acc)
            else (prevIsEven, acc)
```

Alternating Even/Odd List

- What is a Fold?
- Fold Types
- Steps of a Fold
- **Examples**
- Short-Circuiting

```haskell
-- Step 5: Postprocess
altEvenOdds :: [Int] -> [Int]
altEvenOdds [] = []
altEvenOdds (x : xs) = reverse . snd $
  foldl f (x `mod` 2 == 0, [x]) xs
  where
    f :: (Bool, [Int]) -> Int -> (Bool, [Int])
    f (prevIsEven, acc) x =
      let thisIsEven = x `mod` 2 == 0
      in  if thisIsEven /= prevIsEven
            then (thisIsEven, x : acc)
            else (prevIsEven, acc)
```

How to Short-Circuit

- What is a Fold?
- Fold Types
- Steps of a Fold
- Examples
- **Short-Circuiting**

```
# Find the product of the numbers
# until the absolute value exceeds 100
def productUntil100(inputs: [int]):
  product = 1
  for i in inputs:
    product *= i
    if abs(product) > 100:
      break
  return product
```

How to Short-Circuit


```
productUntil100 :: [Int] -> [Int]
productUntil100 xs = ???
  where
    -- Bool is for "are we done?"
    f :: (Bool, Int) -> Int -> (Bool, Int)
```

How to Short-Circuit

- What is a Fold?
- Fold Types
- Steps of a Fold
- Examples
- **Short-Circuiting**

```
productUntil100 :: [Int] -> [Int]
productUntil100 xs = ???
  where
    -- Bool is for "are we done?"
    f :: (Bool, Int) -> Int -> (Bool, Int)
    f (isDone, prev) x = if isDone
        then prev
        else
          let newX = x * prev
          in  (abs newX > 100, newX)
```

How to Short-Circuit

- What is a Fold?
- Fold Types
- Steps of a Fold
- Examples
- **Short-Circuiting**

```haskell
productUntil100 :: [Int] -> [Int]
productUntil100 xs = snd $
  foldl f (False, 1) xs
  where
    -- Bool is for "are we done?"
    f :: (Bool, Int) -> Int -> (Bool, Int)
    f (isDone, prev) x = if isDone
        then prev
        else
          let newX = x * prev
          in  (abs newX > 100, newX)
```

Lecture 7 - Higher Order Function Patterns

What is a Higher Order Function?

- **HOFs**
- Sorting
- Grouping
- "On"

- A function that takes **another function as input**
  - Map, Filter and Fold
- Important part of Haskell
  - Functions are "first class citizens"
- Much easier to customize behavior
  - (and write re-usable, polymorphic code)

Sorting

- HOFs
- **Sorting**
- Grouping
- "On"

```haskell
sort :: (Ord a) => [a] -> [a]

-- Example sort:
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x : xs) = left ++ (x : right)
  where
    left = quicksort $ filter (<= x) xs
    right = quicksort $ filter (> x) xs
```

Sorting

- HOFs
- **Sorting**
- Grouping
- "On"

```
sort :: (Ord a) => [a] -> [a]

data Ordering = LT | EQ | GT

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

- What if our type doesn't have `Ord`?
- What if we want to compare using a different ordering?

Sorting

- HOFs
- **Sorting**
- Grouping
- "On"

```
data Person = Person
  { firstName :: String
  , lastName :: String
  }

people :: [Person]
people =
  [ Person "Joe" "Anderson"
  , Person "Betsy" "Zimmerman"
  , Person" "Gary" "Melanson"
  ]

>> sort people
error!
```

Sorting

- HOFs
- **Sorting**
- Grouping
- "On"

```haskell
data Person = Person
  { firstName :: String
  , lastName :: String
  } deriving (Eq, Ord)

people :: [Person]
people =
  [ Person "Joe" "Anderson"
  , Person "Betsy" "Zimmerman"
  , Person" "Gary" "Melanson"
  ]

>> sort people
[ Person "Betsy" "Zimmerman"
, Person "Gary" "Melanson"
, Person "Joe" "Anderson"
]
```

Custom Comparator

- HOFs
- **Sorting**
- Grouping
- "On"

```
sortBy ::
  (a -> a-> Ordering) -> -- Custom Comparator
  [a]                 -> -- Input List
  [a]                    -- Sorted List
```

Custom Comparator

- HOFs
- **Sorting**
- Grouping
- "On"

```
sortBy ::
  (a -> a-> Ordering) -> -- Custom Comparator
  [a]                 -> -- Input List
  [a]                    -- Sorted List

lastnameFirst :: Person -> Person -> Ordering
lastnameFirst (Person f1 l1) (Person f2 l2) =
  case l1 `compare` l2 of
    EQ -> f1 `compare` f2
    o -> o
```

Custom Comparator

- HOFs
- **Sorting**
- Grouping
- "On"

```
sortBy ::
   (a -> a-> Ordering) -> -- Custom Comparator
   [a]                 -> -- Input List
   [a]                    -- Sorted List

lastnameFirst :: Person -> Person -> Ordering
lastnameFirst (Person f1 l1) (Person f2 l2) =
  case l1 `compare` l2 of
    EQ -> f1 `compare` f2
    o -> o

>> sortBy lastnameFirst people
[ Person "Joe" "Anderson"
, Person "Gary" "Melanson"
, Person "Betsy" "Zimmerman"
]
```

Sorting "On"

- HOFs
- **Sorting**
- Grouping
- "On"

```
sortOn ::
   (Ord b) =>
   (a -> b) -> -- Field to Compare On
   [a]      -> -- Input List
   [a]         -- Sorted List
```

Sorting "On"

- HOFs
- **Sorting**
- Grouping
- "On"

```
sortOn ::
   (Ord b) =>
   (a -> b) -> -- Field to Compare On
   [a]      -> -- Input List
   [a]         -- Sorted List

>> :t lastName
Person -> String

>> sortOn lastName people
[ Person "Joe" "Anderson"
, Person "Gary" "Melanson"
, Person "Betsy" "Zimmerman"
]
```

Grouping

- HOFs
- Sorting
- **Grouping**
- "On"

```
group ::
  (Eq a) =>
  [a]     ->
  [[a]]

>> group "mississippi"
["m", "i", "ss", "i", "ss", "i", "pp", "i"]

>> group [1, 1, 1, 3, 2, 2, 3, 3, 3, 1, 1]
[[1,1,1], [3], [2,2], [3,3,3], [1,1]]
```

Grouping

- HOFs
- Sorting
- Grouping
- "On"

```
group ::
  (Eq a) =>
  [a]    ->
  [[a]]

>> group $ sort "mississippi"
["iiii", "m", "pp", "ssss"]

>> group $ sort [1,1,1,3,2,2,3,3,3,1,1]
[[1,1,1,1,1], [2,2], [3,3,3,3]]
```

Custom Grouping

- HOFs
- Sorting
- **Grouping**
- "On"

```
groupBy ::
   (a -> a -> Bool) -> -- Equality Test
   [a]              -> -- Input list
   [[a]]               -- Nested List

>> groupBy sndsEq
   [(1, 3), (4, 3), (4, 2), (1, 2), (1,3)]
[ [(1,3), (4,3)]
, [(4,2), (1,2)]
, [(1,3)]
]
```

The "On" Pattern

- HOFs
- Sorting
- Grouping
- "On"

```
on ::
   (b -> b -> c) -> -- Field Comparator
   (a -> b)      -> -- Field Accessor
   (a -> a -> c)    -- Result Comparator
```

Grouping "On"

- HOFs
- Sorting
- Grouping
- "On"

```
groupBy ::
  (a -> a -> Bool) -> -- Equality Test
  [a]              -> -- Input list
  [[a]]               -- Nested List

on ::
  (b -> b -> c) -> -- Field Comparator
  (a -> b)      -> -- Field Accessor
  (a -> a -> c)    -- Result Comparator

>> groupBy (on (==) snd)
  [(1, 3), (4, 3), (4, 2), (1, 2), (1,3)]
[ [(1,3), (4,3)]
, [(4,2), (1,2)]
, [(1,3)]
]
```

Grouping "On"

- HOFs
- Sorting
- Grouping
- "On"

```
on ::
   (b -> b -> c) -> -- Field Comparator
   (a -> b)      -> -- Field Accessor
   (a -> a -> c)    -- Result Comparator

(==) :: (Eq b) => b -> b -> Bool

snd :: (x, y) -> y

>> :t (on (==) snd)
((x, y) -> (x, y) -> Bool)
```

Grouping "On"

- HOFs
- Sorting
- Grouping
- "On"

```
on ::
   (b -> b -> c) -> -- Field Comparator
   (a -> b)      -> -- Field Accessor
   (a -> a -> c)    -- Result Comparator

(==) :: (Eq b) => b -> b -> Bool

snd :: (x, y) -> y

>> :t (on (==) snd)
((x, y) -> (x, y) -> Bool)

groupBy ::
   ((x,y) -> (x,y) -> Bool) ->
   [(x,y)]                  ->
   [[(x,y)]]
```

Using "On"

- HOFs
- Sorting
- Grouping
- "On"

```
maximumBy
nubBy
deleteBy
insertBy
unionBy
intersectBy
```

Using "On"

- HOFs
- Sorting
- Grouping
- "On"

```
-- Retrieve the Person with the longest
-- last name

maximumBy :: (a -> a -> Ordering) -> [a] -> a

compare :: Int -> Int -> Ordering

>> :t (length . lastName)
Person -> Int

>> maximumBy
 (on compare (length . lastName)) people

Person "Betsy" "Zimmerman"
```

Lecture 8 - Set Operations

Treating Lists like Sets

- **Lists as Sets**
- Operations
- Examples

- Many loop patterns
- Helper functions allow us to **avoid** writing out these rote patterns every time
- Lists are a basic for of a *Set*
  - Can still use many similar operation
- Operations are less efficient on Lists than Sets
  - But oftentimes better than running two conversions

Nub

- Lists as Sets
- **Operations**
- Examples

```haskell
-- Deduplicates input list
nub :: (Eq a) => [a] -> [a]

>> nub [2, 1, 2, 2, 3, 1, 4, 5, 6, 4, 5, 6]
[2, 1, 3, 4, 5, 6]

>> nub ["Hello", "Hello", "Hello", "Hello"]
["Hello"]

>> nub ["Hello", "World", "!"]
["Hello", "World", "!"]
```

Delete

- Lists as Sets
- Operations
- Examples

```
-- Deletes _first_ instance of input from list
delete :: (Eq a) => a -> [a] -> [a]

>> delete 2 [2, 1, 2, 2]
[1, 2, 2]

>> delete "Hello" ["!", "Hello", "Hello"]
["!", "Hello"]

>> delete "Water" ["Hello", "World", "!"]
["Hello", "World", "!"]
```

Set Difference

- Lists as Sets
- **Operations**
- Examples

```
(\\) :: (Eq a) => [a] -> [a] -> [a]

>> [1, 2, 3] \\ [1, 3]
[2]

>> [2, 3, 1, 1, 2, 3] \\ [1, 3]
[2, 1, 2, 3]

>> [2, 3, 1, 1, 2, 3] \\ [1, 3, 3, 2]
[1, 2]

>> ["Hello", "World", "Hello"] \\ ["!"]
["Hello", "World", "Hello"]
```

Intersect

- Lists as Sets
- Operations
- Examples

```
intersect :: (Eq a) => [a] -> [a] -> [a]

>> intersect [1, 2, 3] [1, 3, 4, 5]
[1, 3]

>> intersect ["Hello"] ["World"]
[]

>> [1, 4, 5, 1, 1] `intersect` [5, 1]
[1, 5, 1, 1]

>> [1, 3, 6] `intersect` [6, 6, 1, 1]
[1, 6]
```

Union

- Lists as Sets
- **Operations**
- Examples

```
union :: (Eq a) => [a] -> [a] -> [a]

>> union [1, 2, 3] [1, 3, 4, 5]
[1, 2, 3, 4, 5]

>> union ["Hello"] ["World"]
["Hello", "World"]

>> [1, 4, 5, 1, 1] `union` [5, 1]
[1, 4, 5, 1, 1]

>> [1, 3, 6] `union` [6, 6, 1, 1]
[1, 3, 6]
```

"By" Functions

- Lists as Sets
- **Operations**
- Examples

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

deleteBy :: (a -> a -> Bool)
  -> a -> [a] -> [a]

deleteFirstsBy :: (a -> a -> Bool)
  -> [a] -> [a] -> [a]

unionBy :: (a -> a -> Bool)
  -> [a] -> [a] -> [a]

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

Day Shift Positions

- Lists as Sets
- Operations
- **Examples**

```
data Employee = Employee
  { name :: String
  , role :: String
  , department :: String
  }

onlyDayShifts :: [Employee] -> [Employee]
  -> [(String, String)]
onlyDayShifts = ...
```

Day Shift Positions

- Lists as Sets
- Operations
- **Examples**

```
data Employee = Employee
  { name :: String
  , role :: String
  , department :: String
  }

onlyDayShifts :: [Employee] -> [Employee]
  -> [(String, String)]
onlyDayShifts days nights = ...
  where
    f (Employee n r d) = (r, d)
    days' = map f days
    nights' = map f nights
```

Day Shift Positions

- Lists as Sets
- Operations
- **Examples**

```
data Employee = Employee
  { name :: String
  , role :: String
  , department :: String
  }

onlyDayShifts :: [Employee] -> [Employee]
  -> [(String, String)]
onlyDayShifts days nights =
  nub days' \\ nights'
  where
    f (Employee n r d) = (r, d)
    days' = map f days
    nights' = map f nights
```

Student Sample

- Lists as Sets
- Operations
- Examples

```
data Student = Student
  { major :: String
  , name :: String
  , age :: Int
  }

-- Select one student with each major
sampleMajors :: [Student] -> [Student]
sampleMajors students = ...
```

Student Sample

- Lists as Sets
- Operations
- **Examples**

```haskell
data Student = Student
  { major :: String
  , name :: String
  , age :: Int
  }

-- Select one student with each major
sampleMajors :: [Student] -> [Student]
sampleMajors students = nubBy
  (on (==) major) students
```

Student Sample

- Lists as Sets
- Operations
- **Examples**

```haskell
sampleMajors :: [Student] -> [Student]
sampleMajors students = nubBy
  (on (==) major) students


allStudents =
  [ Student "Math" "Jason" 21
  , Student "Chemistry "Kelsey" 19
  , Student "Math" "Allie" 23
  , Student "Physics" "Allen" 25
  , Student "Physics" "Paul" 23
  , Student "Literature" "Chris" 24
  , Student "Chemistry" "Sally" 18
  ]

>> sampleMajors allStudents
[ Student "Math" "Jason" 21
, Student "Chemistry" "Kelsey" 19
, Student "Physics" "Allen" 25
, Student "Literature" "Chris" 24 ]
```

Lecture 9 - Monadic Operations

Monads in Problem Solving

- **Monads**
- Logging
- Fail
- State
- Extra Functions

- Monads are important in Haskell!
- They can help a lot with puzzle problems
    - (even if they aren't strictly necessary)
- Certain monads are particularly helpful
    - Logger
    - Fail
    - State

Monad Logger

- Monads
- **Logging**
- Fail
- State
- Extra Functions

- Allows logging of messages
- Super useful for debugging

```
class MonadLogger m where
  ...
```

Monad Logger

- Monads
- Logging
- Fail
- State
- Extra Functions

- Allows logging of messages
- Super useful for debugging

```
class MonadLogger m where
  ...

logDebugN :: (MonadLogger m) => Text -> m ()

add :: (MonadLogger m) => Int -> Int -> m Int
add x y = do
  logDebugN $ pack $
    "Adding " <> show x <> " " <> show y
  return (x + y)

-- Also...
logInfoN
logWarnN
logErrorN
```

Stdout Logging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```haskell
add :: (MonadLogger m) => Int -> Int -> m Int
add x y = do
  logDebugN $ pack $
    "Adding " <> show x <> " " <> show y
  return (x + y)

runStdoutLoggingT :: (MonadIO m) =>
  LoggingT m a -> m a

main :: IO ()
main = void $ runStdoutLoggingT $ do
  add 5 4
  add 6 7

-- Output...
"... Adding 5 4"
"... Adding 6 7"
```

Other Loggers

- Monads
- Logging
- Fail
- State
- Extra Functions

```
runWriterLoggingT :: (Functor m) =>
  WriterLoggingT m a -> m (a, [LogLine])

runStderrLoggingT :: MonadIO m =>
  LoggingT m a -> m a

runFileLoggingT :: MonadBaseControl IO m =>
  FilePath -> LoggingT m a -> m a

runChanLoggingT :: MonadIO m =>
  Chan LogLine -> LoggingT m a -> m a
```

Python Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```python
def largestTriplet(ls: [int]):
  if len(ls) < 3:
    raise RuntimeError("Not enough!")
  (x, y, z) = (ls[0], ls[1], ls[2])
  i = 3
  while i < len(ls) - 3:
    (a, b, c) = (ls[i], ls[i+1], ls[i+2])
    if a + b + c > x + y + z:
      (x, y, z) = (a, b, c)
    i += 3
  return (x, y, z)

>> largestTriplet([1, 2, 4, 5, 6, 1])
(5, 6, 1)
```

Python Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```
def largestTriplet(ls: [int]):
  if len(ls) < 3:
    raise RuntimeError("Not enough!")
  (x, y, z) = (ls[0], ls[1], ls[2])
  i = 3
  while i < len(ls) - 3:
    (a, b, c) = (ls[i], ls[i+1], ls[i+2])
    if a + b + c > x + y + z:
      (x, y, z) = (a, b, c)
    i += 3
    print str((a, b, c))
  return (x, y, z)

>> largestTriplet([1, 2, 4, 5, 6, 1])
"(5, 6, 1)"
(5, 6, 1)
```

Python Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```
def largestTriplet(ls: [int]):
  if len(ls) < 3:
    raise RuntimeError("Not enough!")
  (x, y, z) = (ls[0], ls[1], ls[2])
  i = 1
  while i < len(ls) - 3:
    (a, b, c) = (ls[i], ls[i+1], ls[i+2])
    if a + b + c > x + y + z:
      (x, y, z) = (a, b, c)
    i += 1
    print str((a, b, c))
  return (x, y, z)

>> largestTriplet([1, 2, 4, 5, 6, 1])
"(2, 4, 5)"
"(4, 5, 6)"
"(5, 6, 1)"
(4, 5, 6)
```

Haskell Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```
largestTriplet :: [Int] -> (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3) rest
  _ -> error "Not enough ints!"
  where
    f (x, y, z) (a : b : c : r) =
      if a + b + c > x + y + z
        then f (a, b, c) r
        else f (x, y, z) r
    f (x, y, z) _ = (x, y, z)

>> largestTriple [1, 2, 4, 5, 6, 1]
(5, 6, 1)
```

Haskell Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```
largestTriplet :: (MonadLogger m) =>
  [Int] -> m (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3) rest
  _ -> error "Not enough ints!"
  where
    f (x, y, z) (a : b : c : r) = do
      logDebugN $ pack . show $ (a, b, c)
      if a + b + c > x + y + z
        then f (a, b, c) r
        else f (x, y, z) r
    f (x, y, z) _ = return (x, y, z)

>> runStdoutLoggingT $
  largestTriple [1, 2, 4, 5, 6, 1]
"... (5, 6, 1)"
(5, 6, 1)
```

Haskell Debugging

- Monads
- **Logging**
- Fail
- State
- Extra Functions

```
largestTriplet :: (MonadLogger m) =>
  [Int] -> m (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> error "Not enough ints!"
  where
    f (x, y, z) (a : b : c : r) = do
      logDebugN $ pack . show $ (a, b, c)
      if a + b + c > x + y + z
        then f (a, b, c) (b : c : r)
        else f (x, y, z) (b : c : r)
    f (x, y, z) _ = return (x, y, z)

>> runStdoutLoggingT $
  largestTriple [1, 2, 4, 5, 6, 1]
"... (2, 4, 5)"
"... (4, 5, 6)"
"... (5, 6, 1)"
(5, 6, 1)
```

MonadFail

- Monads
- Logging
- Fail
- State
- Extra Functions

```
class MonadFail m where
  fail :: String -> m a
```

- State conditions where function "fails"
- Useful when you know the puzzle places certain constraints on the input
    - (Avoid dealing with unnecessary edge cases)
- Works like `error`, but is more flexible

Using Maybe

- Monads
- Logging
- Fail
- State
- Extra Functions

```haskell
-- If constraints aren't met, return Nothing!
instance MonadFail Maybe where
  fail _ = Nothing
```

Using Maybe

- Monads
- Logging
- Fail
- State
- Extra Functions

```
-- If constraints aren't met, return Nothing!
instance MonadFail Maybe where
  fail _ = Nothing

largestTriplet :: (MonadFail m) =>
  [Int] -> m (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> fail "Not enough ints!"
  where
    ...

>> largestTriple [] :: Maybe (Int, Int, Int)
Nothing
>> largestTriple [1,2,3]
  :: Maybe (Int, Int, Int)
Just (1, 2, 3)
```

Using IO

- Monads
- Logging
- Fail
- State
- Extra Functions

```
instance MonadFail IO where
  fail = failIO

largestTriplet :: (MonadFail m) =>
  [Int] -> m (Int, Int, Int)
largestTriplet xs = case xs of
  (x1 : x2 : x3 : rest) -> f (x1, x2, x3)
    (x2 : x3 : rest)
  _ -> fail "Not enough ints!"
  where
    ...

>> largestTriple [] :: IO (Int, Int, Int)
IO Error "Not enough ints!"
>> largestTriple [1,2,3] :: IO (Int, Int, Int)
(1, 2, 3)
```

The State Monad

- Monads
- Logging
- Fail
- **State**
- Extra Functions

```
get :: State s s
put :: s -> State s ()
modify :: (s -> s) -> State s ()
runState :: State s a -> s -> (a, s)
```

Extra Functions

- Monads
- Logging
- Fail
- State
- **Extra Functions**

```
mapM :: (a -> m b) -> [a] -> m [b]

printAdd :: Int -> Int -> IO Int
printAdd x y = do
  putStrLn $ "Add " <> show x <> " " <> show y
  return $ x + y

>> mapM (printAdd 5) [3, 2, 1]
Add 5 3
Add 5 2
Add 5 1
[8, 7, 6]

mapM_ :: (a -> m b) -> [a] -> m ()

>> mapM_ (printAdd 5) [3, 2, 1]
Add 5 3
Add 5 2
Add 5 1
()
```

Extra Functions

- Monads
- Logging
- Fail
- State
- **Extra Functions**

```haskell
forM :: [a] -> (a -> m b) -> m [b]

printAdd :: Int -> Int -> IO Int
printAdd x y = do
  putStrLn $ "Add " <> show x <> " " <> show y
  return $ x + y

>> forM [3, 2, 1] (printAdd 5)
Add 5 3
Add 5 2
Add 5 1
[8, 7, 6]

-- Or just...
>> forM [3, 2] $ \i -> do
  putStrLn $ "Add 5 " <> show i
  return $ 5 + i
Add 5 3
Add 5 2
[8, 7]
```

Extra Functions

- Monads
- Logging
- Fail
- State
- **Extra Functions**

```haskell
foldM :: (a -> b -> m a) -> a -> [b] -> m a

printAdd :: Int -> Int -> IO Int
printAdd x y = do
  putStrLn $ "Add " <> show x <> " " <> show y
  return $ x + y

>> foldM printAdd 1 [5, 9, 12]
Add 1 5
Add 6 9
Add 15 12
27
```

Extra Functions

- Monads
- Logging
- Fail
- State
- **Extra Functions**

```
sequence :: [(m a)] -> m [a]
```

Extra Functions

- Monads
- Logging
- Fail
- State
- Extra Functions

```
sequence :: [(m a)] -> m [a]

actions :: [State Int ()]
actions = [put 5, modify (+4), modify (-3)]

result :: Int
result = execState (sequence actions) 0

>> result
6
```

Monads by Default

- Monads
- Logging
- Fail
- State
- **Extra Functions**

- Get used to using monads by default!
- Can be hard to introduce a monad (e.g. Logger) after you've written a long function
- But even a basic `Monad` constraint gets you to use do-syntax.
  - (Can always treat it as `Identity` monad and use `runIdentity` for it)
  - `runIdentity :: Identity a -> a`

Lecture 10 - State Evolution Patterns

State Evolution Patterns

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

- State Evolution Patterns
  - Can be similar to folds, but not exactly
- Folding functions always receive a new input from a function.
  - `a -> b -> a`
- State evolution updates strictly on the state
  - `a -> a`
- Track over a number of evolutions
  - (Or until a condition is met)
- Finite State Machines and Pushdown Automata
  - More like folds
  - Interesting (and common) use cases

Evolve State Function

- Evolve State
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
evolveState ::
  (a -> a) -> -- State Evolution Function
  a        -> -- Initial State
  Int      -> -- Number of iterations to run
  a           -- Final State
```

Mortgage Value

- Evolve State
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
rate :: Double
rate = 0.05

monthlyPayment :: Double
monthlyPayment = 1000.0

payMonth :: Double -> Double
payMonth principal =
  principal - (monthlyPayment - interest)
  where
    interest = principal * (rate / 12.0)
```

Mortgage Value

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
rate :: Double
rate = 0.05

monthlyPayment :: Double
monthlyPayment = 1000.0

payMonth :: Double -> Double
payMonth principal =
  principal - (monthlyPayment - interest)
  where
    interest = principal * (rate / 12.0)

>> evolveState payMonth 100000.0 12
92837.33
>> evolveState payMonth 100000.0 24
85308.21
>> evolveState payMonth 100000.0 129
626.35
```

Pressure Buildup

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
Initial: [0,0,50,0,100]

After One Step:

[ 0
  16 (1/3 of 50)
  25 (1/2 of 50)
  33 (1/3 of 100)
  50 (1/2 of 100)
]
```

Pressure Buildup

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
After One Step: [0,16,25,33,50]

After Two Steps:

[ 5  (1/3 of 16)
  16 (1/2 of 16 + 1/3 of 25)
  23 (1/2 of 25 + 1/3 of 33)
  32 (1/2 of 33 + 1/3 of 50)
  25 (1/2 of 50)
]
```

Pressure Buildup

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
After Two Steps: [5,16,23,32,25]

After Three Step:

[ 1  (1/3 of 5)
  7  (1/2 of 5 + 1/3 of 16)
  15 (1/2 of 16 + 1/3 of 23)
  21 (1/2 of 23 + 1/3 of 32)
  24 (1/2 of 32 + 1/3 of 25)
  12 (1/2 of 25)
]
```

Pressure Buildup

- **Evolve State**
- Evolve Until
- Finite State Machines
- Pushdown Automata

```
movePressure :: [Int] -> [Int]
movePressure pressures =
  f (0, []) (reverse pressures)
  where
    f :: (Int, [Int]) -> [Int] -> [Int]
    f (right, acc) [] = if right > 2
      then (quot right 3 : acc)
      else acc
    f (right, acc) (here : rst) =
      let newP = quot right 3 + quot here 2
      in  f (here, newP : acc) rst

>> evolveState movePressure [0,0,50,0,100] 1
[0,16,25,33,50]
>> evolveState movePressure [0,0,50,0,100] 2
[5,16,23,32,25]
>> evolveState movePressure [0,0,50,0,100] 3
[1,7,15,21,24,12]
>> evolveState movePressure [0,0,50,0,100] 10
[0,2,3,2,1,0,0]
```

Evolve Until Function

- Evolve State
- **Evolve Until**
- Finite State Machines
- Pushdown Automata

```
evolveUntil ::
   (a -> a)      -> -- State Evolution Function
   a             -> -- Initial State
   (a -> Bool)   -> -- Completion Predicate
   a                -- Final State
```

Evolve Until Mortgage

- Evolve State
- **Evolve Until**
- Finite State Machines
- Pushdown Automata

```
payMonth :: (Int, Double) -> (Int, Double)
payMonth (x, principal) = (x + 1,
  principal - (monthlyPayment - interest))
  where
    interest = principal * (rate / 12.0)



>> evolveUntil payMonth (0, 100000.0)
  ((<= 0.0) . snd)
(130, -371.04)
>> evolveUntil payMonth (0, 120000.0)
  ((<= 0.0) . snd)
(167, -297.91)
```

Evolve Until Pressure

- Evolve State
- **Evolve Until**
- Finite State Machines
- Pushdown Automata

```
movePressure :: [Int] -> [Int]
movePressure = ...



>> length $ evolveUntil movePressure
  [0,0,50,0,100] (all (== 0))
7
>> length $ evolveUntil movePressure
  [0,0,500,0,1000] (all (== 0))
12
```

Finite State Machines

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

- AKA Deterministic Finite Automata
  - (Non-deterministic also exists)
- Simple computational model
- Can recognize a regular language
  - (Determine if an input string satisfies a particular regular expression)

Finite State Machines

- Evolve State
- Evolve Until
- Finite State Machines
- Pushdown Automata

- AKA Deterministic Finite Automata
  - (Non-deterministic also exists)
- Simple computational model
- Can recognize a regular language
  - (Determine if an input string satisfies a particular regular expression)
- Works in a stateful way, but **without memory.**
  - Only knowledge is current state (enum) and the next input character

Modeling FSMs

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

- Use an enumerated type for each state
- Have a state transition function
  - Takes next character and current state
  - Produces the following state

```
data FSMState =
  Initial | Success | Failure | ...
  deriving (Eq, Enum)

transition :: Char -> FSMState -> FSMState
...
```

Modeling FSMs

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

- Use an enumerated type for each state
- Have a state transition function
  - Takes next character and current state
  - Produces the following state

```
data FSMState =
  Initial | Success | Failure | ...
  deriving (Eq, Enum)

transition :: Char -> FSMState -> FSMState
...

runFSM ::
  (Enum a) =>
  (Char -> a -> a) -> -- Transition Function
  (a -> Bool)      -> -- Completion Predicate
  a                -> -- Initial State
  String           -> -- Input String
  Bool                -- Does String pass?
```

Parsing "ab"

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

```
data ABState =
  Initial | Failure | ReceivedA | ReceivedB
  deriving (Eq, Enum)
```

Parsing "ab"

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

```
data ABState =
  Initial | Failure | ReceivedA | ReceivedB
  deriving (Eq, Enum)

transition :: Char -> ABState -> ABState
transition ('a', Initial) = ReceivedA
transition ('b', ReceivedA) = ReceivedB
transition _ = Failure
```

Parsing "ab"

- Evolve State
- Evolve Until
- **Finite State Machines**
- Pushdown Automata

```
data ABState =
  Initial | Failure | ReceivedA | ReceivedB
  deriving (Eq, Enum)

transition :: Char -> ABState -> ABState
transition ('a', Initial) = ReceivedA
transition ('b', ReceivedA) = ReceivedB
transition _ = Failure

parseAB :: String -> Bool
parseAB =
  runFSM transition (== ReceivedB) Initial

>> parseAB "a"
False
>> parseAB "ab"
True
>> parseAB "abb"
False
```

Pushdown Automata

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

- Similar to FSMs
  - Enumerated State
- But **also** allows a **stack of tokens**
- Each transition can push or pop a token from the stack
  - (And top of the stack can determine which state we go to)

Pushdown Automata

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

```haskell
runPDA ::
  (Enum a) =>
  -- Transition Function
  -- Uses stack in input and output
  (Char -> (a, [Char]) -> (a, [Char])) ->
  (a -> Bool)       -> -- Completion Predicate
  a                 -> -- Initial State
  String            -> -- Input String
  Bool                 -- Does String pass?
```

Solving a New Problem

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

- 0^n1^n
  - Some 0's and then an equal number of 1's
- FSM cannot solve this!
  - (Can't "count" the number of 0's)
- This is a **Context-Free Language**
- PDA can "count" 0's by storing them on the stack

Solving a New Problem

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

```
data PDAState = Initial | Success | Failure
   | Count0s | Count1s
  deriving (Eq, Enum)
```

PDA Transition Function

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

```
data PDAState = Initial | Success | Failure
  | Count0s | Count1s
  deriving (Eq, Enum)

f :: Char ->
  (PDAState, [Char]) -> (PDAState, [Char])
f c (st, stack) = case (st, stack, c) of
  (Initial, [], '0') -> (Count0s, ['0'])
  (Success, _, _) -> (Failure, stack)
  (Failure, _, _) -> (Failure, stack)
  ...
  _ -> (Failure, stack)
```

PDA Transition Function

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

```
data PDAState = Initial | Success | Failure
  | Count0s | Count1s
  deriving (Eq, Enum)

f :: Char ->
  (PDAState, [Char]) -> (PDAState, [Char])
f c (st, stack) = case (st, stack, c) of
  (Initial, [], '0') -> (Count0s, ['0'])
  (Success, _, _) -> (Failure, stack)
  (Failure, _, _) -> (Failure, stack)
  (Count0s, _ '0') -> (Count0s, '0' : stack)
  (Count0s, ['0'], '1') -> (Success, [])
  (Count0s, ('0' : r), '1') -> (Count1s, r)
  (Count1s, ['0'], '1') -> (Success, [])
  (Count1s, ('0' : r), '1') -> (Count1s, r)
  _ -> (Failure, stack)
```

PDA Transition Function

- Evolve State
- Evolve Until
- Finite State Machines
- **Pushdown Automata**

```
data PDAState = Initial | Success | Failure
  | Count0s | Count1s
  deriving (Eq, Enum)

f :: Char ->
  (PDAState, [Char]) -> (PDAState, [Char])

eval0n1n :: String -> Bool
eval0n1n = runPDA f (== Success) Initial

>> eval0n1n "000111"
True
>> eval0n1n "0001111"
False
>> eval0n1n "01"
True
>> eval0n1n "0"
False
```

Lecture 11 - Module Review

Module Review

- **Module Review**

- Extra practice problems in the exercises!
  - Use `MonadLogger` by default
- Let's do some review!

Module Review

- **Module Review**

- Extra practice problems in the exercises!
  - Use `MonadLogger` by default
- Let's do some review!
- Inner workings of Haskell Lists
- Extensive look at the API

Module Review

- **Module Review**

- Extra practice problems in the exercises!
  - Use `MonadLogger` by default
- Let's do some review!
- Inner workings of Haskell Lists
- Extensive look at the API
- Loop Patterns
  - How they translate from other languages to Haskell

Loop Patterns Chart

- **Module Review**

While Loops	Recursion: 1. **Base Case** 2. **Separate cases** 3. **Run recursion** 4. **Combine results**
For-each Loops	Folding: 1. **Determine State** 2. **Write Folding Function** 3. **Initial Value** 4. **List Input** 5. **Postprocess**
Specific For-each patterns	(e.g. Map, Filter)
For Loops with Odd Rules	General Recursion
Stateful Problems	State Evolution Patterns, Finite Automata