

# CS252 – Midterm Exam Study Guide

By: Zayd Hammoudeh

## Lecture #01 – General Introduction

Reasons for Different Programming Languages		Programming Language Design Choices		Features of Good Programming Languages	
<ol style="list-style-type: none"> <li>1. <b>Different domains</b> (e.g. web, security, bioinformatics)</li> <li>2. <b>Legacy code and libraries</b></li> <li>3. <b>Personal preference</b></li> </ol>		<ol style="list-style-type: none"> <li>1. <b>Flexibility</b></li> <li>2. <b>Type safety</b></li> <li>3. <b>Performance</b></li> <li>4. <b>Build Time</b></li> <li>5. <b>Concurrency</b></li> </ol>		<ol style="list-style-type: none"> <li>1. <b>Simplicity</b></li> <li>2. <b>Readability</b></li> <li>3. <b>Learnability</b></li> <li>4. <b>Safety</b> (e.g. security and can errors be caught at compile time)</li> <li>5. <b>Machine independence</b></li> <li>6. <b>Efficiency</b></li> </ol>	
				Goals almost always conflict	
<b>Conflict: Type Systems</b> <ul style="list-style-type: none"> <li>• <b>Advantage:</b> Prevents bad programs.</li> <li>• <b>Disadvantage:</b> Reduces programmer flexibility.</li> </ul>		<b>Blub Paradox:</b> Why do I need advanced programming language techniques (e.g. monads, closures, type inference, etc.)? My language does not have it, and it works just fine.		<b>Current Programming Language Issues</b> <ul style="list-style-type: none"> <li>• <b>Multi-code “explosion”</b></li> <li>• <b>Big Data</b></li> <li>• <b>Mobile Devices</b></li> </ul>	
				<b>Advantages of Web and Scripting Languages</b> <ul style="list-style-type: none"> <li>• <b>Examples:</b> Perl, Python, Ruby, PHP, JavaScript</li> <li>• <b>Highly flexible</b></li> <li>• <b>Dynamic typing</b></li> <li>• <b>Easy to get started</b></li> <li>• <b>Minimal typing</b> (i.e. type systems)</li> </ul>	
<b>Major Programming Language Research Contributions</b> <ul style="list-style-type: none"> <li>• Garbage collection</li> <li>• <b>Sound</b> type systems</li> <li>• Concurrency tools</li> <li>• Closures</li> </ul>		<b>Programs that Manipulate Other Programs</b> <ul style="list-style-type: none"> <li>• <b>Compilers &amp; interpreters</b></li> <li>• <b>JavaScript rewriting</b></li> <li>• <b>Instrumentation</b></li> <li>• <b>Program Analyzers</b></li> <li>• <b>IDEs</b></li> </ul>		<b>Formal Semantics</b> <ul style="list-style-type: none"> <li>• Used to <b>share information unambiguously</b></li> <li>• <b>Can formally prove a language supports a given property</b></li> <li>• <b>Crisply define how a language works</b></li> </ul>	
				<b>Types of Formal Semantics</b> <ul style="list-style-type: none"> <li>• <b>Operational</b> <ul style="list-style-type: none"> <li>◦ Big Step “<b>natural</b>”</li> <li>◦ Small Step “<b>structural</b>”</li> </ul> </li> <li>• <b>Axiomatic</b></li> <li>• <b>Denotational</b></li> </ul>	

### Haskell

<ul style="list-style-type: none"> <li>• <b>Purely functional</b> – Define “<i>what stuff is</i>”</li> <li>• <b>No side effects</b></li> <li>• <b>Referential transparency</b> – A function with the same input parameters will always have the same result.             <ul style="list-style-type: none"> <li>◦ An expression can be replaced with its value and nothing will change.</li> </ul> </li> <li>• Supports type inference.</li> </ul>		<b>Duck Typing</b> – Suitability of an object for some function is determined not by its type but by presence of certain methods and properties. <ul style="list-style-type: none"> <li>◦ <b>More flexible</b> but <b>less safe</b>.</li> <li>◦ <b>Supported by Haskell</b></li> <li>◦ <b>Common in scripting languages</b> (e.g. Python, Ruby)</li> </ul>	
		<b>Side Effects in Haskell</b> <ul style="list-style-type: none"> <li>• Generally not supported.</li> <li>• <b>Example of Support Side Effects:</b> File IO</li> <li>• Functions that do have side effects must be separated from other functions.</li> </ul>	
		<b>Lazy Evaluation</b> <ul style="list-style-type: none"> <li>• <b>Results are not calculated until they are needed</b></li> <li>• <b>Allows for the representation of infinite data structures</b></li> </ul>	

## Lecture #02 – Introduction to Haskell

<b>Key Traits of Haskell</b> <ol style="list-style-type: none"> <li>1. <b>Purely functional</b></li> <li>2. <b>Lazy evaluation</b></li> <li>3. <b>Statically typed</b></li> <li>4. <b>Type Inference</b></li> <li>5. <b>Fully curried functions</b></li> </ol>		<b>ghci</b> – Interactive Haskell.  <b>let</b> – Keyword required in ghci to set a variable value. <b>Example:</b> <code>&gt; let f x = x + 1</code> <code>&gt; f 3</code> <code>4</code>	
		<b>Run Haskell from Command Line</b> Use <b>runhaskell</b> keyword. Example:  <code>&gt; runhaskell &lt;FileName&gt;.hs</code>	
		<b>Hello World in Haskell</b>  <pre>main :: IO () main = do     putStrLn "Hello World"</pre>	

Primitive Classes in Haskell	Lists		Ranges
	<ul style="list-style-type: none"> <li>• <b>Base 0</b></li> <li>• Comma separated in square brackets</li> <li>• <b>Operators</b> <ul style="list-style-type: none"> <li>◦ <b>:</b> Prepend</li> <li>◦ <b>++</b> Concatenate</li> <li>◦ <b>!!</b> Get element a specific index</li> <li>◦ <b>head</b> First element in list</li> <li>◦ <b>tail</b> All elements after head</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>◦ <b>last</b> Last element in the list</li> <li>◦ <b>init</b> All elements except the last</li> <li>◦ <b>take n</b> Take first n elements from a list</li> <li>◦ <b>replicate l m</b> Create a list of length l containing only m</li> <li>◦ <b>repeat m</b> Create an in</li> </ul>	<ul style="list-style-type: none"> <li>• Can be infinite or bounded</li> <li>• Use the “<b>..</b>” notation. <b>Examples:</b>  <code>&gt; [1..4]</code>  <code>[1, 2, 3, 4]</code>  <code>&gt; [1,2..6]</code>  <code>[1, 2, 3, 4, 5, 6]</code>  <code>&gt; [1,3..10]</code>  <code>[1, 3, 5, 7, 9]</code> </li> </ul>
<b>Hello World in Haskell</b> <pre>main :: IO () main = do     putStrLn "Hello World"</pre>	<b>List Examples</b> <code>&gt; putStrLn \$ "Hello " ++ "World"</code> <code>"Hello World"</code>  <code>&gt; let s = bra in s !! 2 : s ++ 'c' : last s : 'd' : s</code> <code>"abracadabra"</code>		<b>Infinite List Example</b> <code>&gt; let even = [2,4..]</code> <code>&gt; take 5 even</code> <code>[2, 4, 6, 8, 10]</code>

<p><b>List Comprehension</b></p> <ul style="list-style-type: none"> <li>Based off set notation.</li> <li>Supports filtering as shown in second example</li> <li>If multiple variables (e.g. a, b, c) are specified, iterates through them like nested for loops.</li> <li>Uses the pipe ( ) operator. Examples:</li> </ul> <pre>&gt; [ 2*x   x &lt;- [1..5]] [2, 4, 6, 8, 10]</pre> <pre>&gt; [(a, b, c)   a &lt;- [1..10], b &lt;- [1..10],                c &lt;- [1..10], a^2 + b ^2 == c^2] [(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)]</pre>	<p><b>A Simple Function</b></p> <pre>&gt; let inc x = x + 1 &gt; inc 3 4  &gt; inc 4.5 5.5  &gt; inc (-5) -- Negative -4</pre> <p><b>Type Signature</b></p> <ul style="list-style-type: none"> <li>Uses symbols ":" and "&gt;"</li> <li>Example:</li> </ul> <pre>inc :: Int -&gt; Int inc x = x + 1</pre>	<p><b>Pattern Matching</b></p> <ul style="list-style-type: none"> <li>Used to handle different input data</li> <li>Guard uses the pipe ( ) operator</li> <li>Example:</li> </ul> <pre>inc :: Int -&gt; Int inc x     x &lt; 0 = error "invalid x" inc x = x + 1</pre>
--	---	---

<p><b>Recursion</b></p> <ul style="list-style-type: none"> <li>Base Case – Says when recursion should stop.</li> <li>Recursive Step – Calls the function with a smaller version of the problem</li> </ul> <p>Example:</p> <pre>addNum :: [Int] -&gt; Int addNum [] = 0 addNum (x:xs) = x + addNum xs</pre>	<p><b>Lab #01 – Max Number</b></p> <pre>&gt; maxNum :: [Int] -&gt; Int &gt; maxNum [] = error "Invalid Input" &gt; maxNum [x] = x &gt; maxNum (x:xs) = if x &gt; maxxs then x else maxxs &gt; where maxxs = maxNum xs</pre>	
--	---	--

## Lecture #03 – Operational Semantics

<p><b>Formal Semantics</b></p> <p>Crisply define how the language features work.</p> <p><b>Abstract Syntax Tree</b></p> <p>Tree representation of the abstract syntactic structure of a program's source code. Example is Bool* language below.</p> <p><b>Bool* Language</b></p> <pre>e ::=   true   false   if e   then e   else e  v ::=   true   false</pre> <p><b>Expressions:</b></p> <pre>constant true constant false conditional</pre> <p><b>Values:</b></p> <pre>constant true constant false</pre>	<p><b>Formal Semantic Styles</b></p> <ul style="list-style-type: none"> <li>Operational             <ul style="list-style-type: none"> <li>Big-Step ("Natural")</li> <li>Small-Step ("structural")</li> </ul> </li> <li>Axiomatic</li> <li>Denotational</li> </ul> <p><b>Big Step Operational Semantics</b></p> <ul style="list-style-type: none"> <li>Evaluates every expression to a value</li> </ul> <p>↓ : "Evaluates to" symbol in Big-Step operational semantics.</p> <p>Example Formatting:</p> $e \Downarrow v$ <ul style="list-style-type: none"> <li>Read as: "Expression e evaluates to the value v"</li> </ul>	<p><b>A Review of Compilers</b></p> <pre>graph LR     SC[source code] --&gt; LT[Lexical Analyzer/Tokenizer]     LT -- tokens --&gt; P[Parser]     P --&gt; AST[Abstract Syntax Tree AST]     AST --&gt; C[Compiler]     AST --&gt; I[Interpreter]     C --&gt; MC[Machine code]     I --&gt; Com[Commands]</pre> <p>We don't care about lexing or parsing.</p> <p>We don't care if we have a compiler or interpreter</p>
--	--	--

<p><b>Small-Step Operational Semantics</b></p> <ul style="list-style-type: none"> <li>Evaluate an expression until it is in normal form</li> <li>Normal Form – Any form that cannot be evaluated further.</li> <li>→ : "Evaluates to" symbol in small step operational semantics. Example:</li> </ul> $e \rightarrow e' \rightarrow e'' \rightarrow v$ <ul style="list-style-type: none"> <li>→* : Many evaluation steps required. Example:</li> </ul> $e \rightarrow^* v$	<p><b>Bool* Small-Step Operational Semantics Rules</b></p> <p><b>E-IfTrue:</b></p> $\frac{}{\text{if true then } e_2 \text{ else } e_3 \rightarrow e_2}$ <p><b>E-IfFalse:</b></p> $\frac{}{\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3}$ <p><b>E-If:</b></p> $\frac{e_1 \rightarrow e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3}$	<p>Example: Reduce the expression</p> <pre>if (if true then false else true) then true else false</pre> <p>Step #1: Use rule E-IfTrue</p> <pre>if false then true else false</pre> <p>Step #2: Use rule E-IfFalse (Now in normal form)</p> <pre>false</pre>
--	--	---

<p><b>Bool* Extension: Numbers</b></p> <ul style="list-style-type: none"> <li>0 : The Number "0"</li> <li>succ 0 : Represents "1"</li> <li>succ succ 0 : Represents "2"</li> <li>pred n : Gets the predecessor of "n"</li> </ul>	<p><b>Extended Bool* Language</b></p> <pre>e ::=   true   false   if e then e else e   0   succ e   pred e  v ::= true   false         IntV  IntV ::= 0   succ IntV</pre>	<p><b>Literate Haskell</b></p> <ul style="list-style-type: none"> <li>File Extension: ".lhs"</li> <li>Code lines begin with "&gt;"</li> <li>All other lines are comments.</li> <li>"Essentially swaps code with comments."</li> </ul>	<p><b>Case Statement in Haskell</b></p> <ul style="list-style-type: none"> <li>Keywords: case, of, otherwise</li> <li>Operator: -&gt;</li> </ul> <p>Example:</p> <pre>case x of   val1 -&gt; "Value 1"   val2 -&gt; "Value 2"   otherwise -&gt; "Everything else."</pre>
--	---	---	--

## Lab #02 Review

Bool Expression Type	BoolVal Type	Type Constructors: BoolExp, BoolVal, BVInt
<pre>&gt; data BoolExp = BTrue &gt;   BFalse &gt;   Bif BoolExp BoolExp BoolExp &gt;   B0 &gt;   Bsucc BoolExp &gt;   Bpred BoolExp &gt; deriving Show</pre>	<pre>&gt; data BoolVal = BVTrue &gt;   BVFalse &gt;   BVNum BVInt &gt; deriving Show  &gt; data BVInt = BV0 &gt;   BVSucc BVInt &gt; deriving Show</pre>	<p><b>Non-nullary Value Constructors:</b> Blf, Bsucc, Bpred, BVSucc, BVNum</p> <p><b>Note:</b> Even constants like B0, BTrue, BFalse, BVTrue, and BVFalse are nullary value constructors (since they take no arguments)</p>

## Lecture #04 – Higher Order Functions

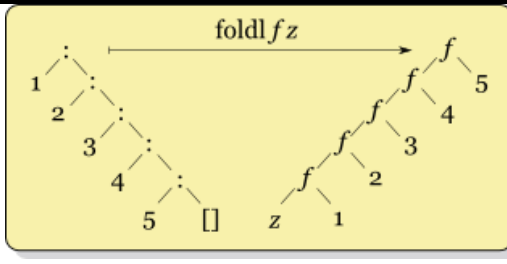
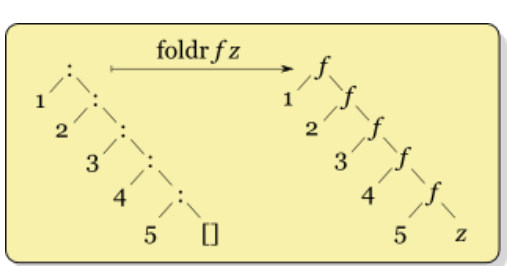
Lambda	Function Composition	Point-Free Style	Example: Lambda with Function Composition
<ul style="list-style-type: none"> <li>Analogous to anonymous classes in Java.</li> <li>Based off Lambda calculus</li> <li><b>Example:</b></li> </ul> <pre>&gt; (\x -&gt; x + 1) 1 2 &gt; (\x y -&gt; x + y) 2 3 5</pre>	<ul style="list-style-type: none"> <li>Uses the <b>period (.)</b></li> <li><math>f(g(x))</math> can be rewritten <math>(f \cdot g) x</math></li> </ul>	<ul style="list-style-type: none"> <li>Pass function arguments no arguments.</li> <li>Example:</li> </ul> <pre>&gt; let inc = (+1) -- No args &gt; inc 3 4</pre>	<pre>&gt; let f = (\x -&gt; x - 5) &gt; . (\y -&gt; y * 2)  &gt; f 7 9  &gt; let f = (\x y -&gt; x - y) &gt; . (\z -&gt; z * (-1))  &gt; f 3 4 -7</pre>

Iterative vs. Recursive	Not Tail Recursive	Tail Recursive Factorial
<ul style="list-style-type: none"> <li><b>Iterative tends to be more efficient than recursive.</b></li> <li><b>Compiler can optimize tail recursive function.</b></li> </ul> <p><b>Tail Recursive Function</b> – The recursive call is the last step performed before returning a value.</p>	<pre>public int factorial(int n) {     if (n==1) return 1;     else {         return n * factorial(n-1);     } }</pre> <p>Last step is the multiplication so not tail recursive.</p>	<pre>public int factorialAcc(int n, int acc) {     if (n==1) return acc;     else {         return factorialAcc(n-1, n*acc);     } }</pre> <p><b>Tail recursive code often uses the accumulator pattern like above.</b></p>

Tail Recursion in Haskell		
<pre>fact' :: Int -&gt; Int -&gt; Int fact' 0 acc = acc fact' n acc = fact' (n - 1) (n * acc)</pre>		

## Higher Order Functions

Functions in Functional Programming	Qualities of Functional Programming	Higher Order Function	
<ul style="list-style-type: none"> <li><b>Functional languages treat programs as mathematical functions.</b></li> <li><b>Mathematical Definition of a Function:</b> A function <math>f</math> is a rule that associates to each <math>x</math> from some set <math>X</math> of values a unique <math>y</math> from a set of <math>Y</math> values.</li> </ul> $(x \in X \wedge y \in Y) \rightarrow y = f(x)$ <ul style="list-style-type: none"> <li><math>f</math> – Name of the function</li> <li><math>x</math> – Independent variable</li> <li><math>y</math> – Dependent variable</li> <li><math>X</math> – Domain</li> <li><math>Y</math> – Range</li> </ul>	<ul style="list-style-type: none"> <li><b>Functions clearly distinguish:</b> <ul style="list-style-type: none"> <li>Incoming values (<b>parameters</b>)</li> <li>Outgoing Values (<b>results</b>)</li> </ul> </li> <li><b>No (re)assignment</b></li> <li><b>No loops</b></li> <li><b>Return values depend only on input parameters</b></li> <li><b>Functions are first class values;</b> this means they can: <ul style="list-style-type: none"> <li>Passed as arguments to a function</li> <li>Be returned from a function</li> <li>Construct new functions dynamically</li> </ul> </li> </ul>	<p>Any function that <b>takes a function as a parameter or returns a function as a result.</b></p> <p><b>Function Currying</b> Transform a function with multiple arguments into multiple functions that each take exactly one argument.</p> <p>Named after Haskell Brooks Curry.</p> <p><b>Currying Example</b></p> <pre>addNums :: Num a =&gt; a -&gt; a -&gt; a</pre> <p><b>addNums</b> is a function that takes in a number and returns a function that takes in another number.</p>	

<p><b>map</b></p> <ul style="list-style-type: none"> <li>Built in Haskell higher order function</li> <li><b>Applies a function to all elements of a list.</b></li> </ul> <pre>map :: (a -&gt; b) -&gt; [a] -&gt; [b]</pre> <pre>&gt; map (+1) [1, 2, 3] [2, 3, 4]</pre>	<p><b>foldl</b></p> <ul style="list-style-type: none"> <li>Built in higher order function</li> <li><b>Does not support infinite lists.</b></li> <li><b>Should only be used for special cases.</b></li> </ul> <pre>foldl :: (b -&gt; a -&gt; b) -&gt; b -&gt; a -&gt; b</pre> <p>Example:</p> <pre>&gt; foldl (\x y -&gt; x - y) 0 [1, 2, 3, 4] -10 -- ((0-1) - 2) - 3) - 4</pre>	
<p><b>filter</b></p> <ul style="list-style-type: none"> <li>Built in Haskell higher order function</li> <li><b>Removes all elements from a list that do not satisfy (i.e. make true) some predicate.</b></li> </ul> <pre>filter :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</pre> <pre>&gt; filter (&gt;2) [1, 2, 3, 4] [3, 4]</pre>	<p><b>foldr</b></p> <ul style="list-style-type: none"> <li>Built in higher order function</li> <li><b>Supports infinite lists.</b></li> <li><b>"Usually the right fold to use"</b></li> </ul> <pre>foldr :: (b -&gt; a -&gt; a) -&gt; a -&gt; b -&gt; a</pre> <p>Example:</p> <pre>&gt; foldr (\x y -&gt; x + y) 0 [1, 2, 3, 4] -2 -- 1 - (2 - (3 - (4 - 0)))</pre>	
<p><b>Thunk</b> – A delayed computation</p> <p>Due to lazy evaluation, <b>foldl</b> and <b>foldr</b> build thunks rather than calculate the results as they go.</p>	<p><b>foldl'</b></p> <ul style="list-style-type: none"> <li><b>Data.list.foldl'</b> evaluates its results eagerly (i.e. does not use thunks)</li> <li><b>Good for large, but finite lists.</b></li> </ul>	

## Lecture #05 – Small-Step Operational Semantics

<p><b>WHILE Language</b></p> <ul style="list-style-type: none"> <li>Unlike the Bool* language, <b>WHILE supports mutable references.</b></li> </ul> <div> <div> <pre>e ::= a       v       a := e       e; e       e op e       if e then e else e       while (e) e</pre> </div> <div> <p>Variable/addresses Values Assignment Sequence Binary Operations Conditional While Loops</p> </div> </div> <div> <pre>v ::= i       b</pre> <p>Integers Boolean</p> </div> <div> <pre>op ::= +   -   *   /         &gt;=   &gt;   &lt;=   &lt;</pre> </div>	<p><b>Small Step Semantics with State</b></p> <ul style="list-style-type: none"> <li>Since the WHILE language supports mutable references, the grammar must be updated to support it.</li> </ul> <p><b>While Relation:</b></p> $e, \sigma \rightarrow e', \sigma'$ <ul style="list-style-type: none"> <li><math>\sigma</math> – Store. <b>Maps references to values.</b></li> </ul> <p><b>Example Operations:</b></p> <ul style="list-style-type: none"> <li><math>\sigma(a)</math> – Retrieves the value at address “a”</li> <li><math>\sigma[a := v]</math> – Identical to the original store with the exception that it now stores the value v at address “a”</li> </ul>	<p><b>Evaluation Order Rules</b></p> <ul style="list-style-type: none"> <li><b>Tend to be repetitive and clutter the semantics.</b></li> <li><b>Context based rules tend to represent the same information as evaluation order rules but more concisely.</b></li> </ul> <p><b>Reduction Rule</b></p> <p>Rewrites the expression. Example:</p> <p><b>E-IfFalse:</b></p> $\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3$ <p><b>Context Rule</b></p> <p><b>Specify the order for evaluating expressions.</b> Example:</p> <p><b>E-If:</b></p> $\frac{e_1 \rightarrow e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3}$
---	---	--

<p><b>Reducible Expression (Redex)</b> – Any expression that can be transformed (reduced) in one step.</p>	<p><b>Example: Redex</b></p> <p><b>if true then (if true then false else false) else true</b></p> <p>This reduces to “<b>if true then false else false</b>”</p>	<p><b>Example: Not a Redex</b></p> <p><b>if (if true then false else false) then true else true</b></p> <p>Not a redex as expression “<b>if true then false else false</b>” must be evaluated first.</p>
--	---	--

<p><b>Evaluation Contexts</b></p> <ul style="list-style-type: none"> <li><b>Alternative to evaluation order rules.</b></li> <li><b>Marker (•) / hole</b> indicate the <b>next place for evaluation</b> (i.e. where we will do the work).</li> </ul> <p>Example:</p> <pre>C[r] = if (if true then false else false) then true else true</pre> <p><b>r = if true then false else false</b></p> <p><b>C = if • then true else true</b></p> <p><b>C[r]</b> is the original expression.</p>	<p><b>Rewriting Evaluation Order Rules</b></p> <p><b>Context based rules only apply to reducible expressions (redexs). Example:</b></p> <p><b>EC-IfFalse:</b></p> $C[\text{if false then } e_2 \text{ else } e_3] \rightarrow C[e_3]$ <p><b>Context Syntax</b></p> <pre>C ::= •       if C then e else e       C op e       v op C       ...</pre>	<p><b>Data.Map</b></p> <ul style="list-style-type: none"> <li><b>Library:</b> Data.Map</li> <li><b>Immutable</b></li> <li><b>Example Methods:</b> <ul style="list-style-type: none"> <li><b>Map.empty</b> – Creates and returns an empty map</li> <li><b>Map.insert k v m</b> – Inserts a value “v” at key “k” into map “m”. <b>Returns a new, updated map.</b></li> <li><b>Map.lookup k m</b> – Returns the value at key “k” in map “m”. <b>Wrapped in a maybe.</b></li> </ul> </li> </ul>
--	--	---

<p><b>Precondition</b> – Text above the line in a rule.</p>	<p><b>Context Rule for Binary Op:</b></p> $\frac{v_3 = v_1 \text{ op } v_2}{C[v_1 \text{ op } v_2] \rightarrow C[v_3]}$	<p><b>How to Read a Small Step Semantic Rule:</b> “Given &lt;Precondition&gt;, then &lt;LeftSideArrow&gt; evaluates to &lt;RightSideArrow&gt;.”</p>
---	---	---

## Lecture #06 – LaTeX

<p><b>TeX</b></p> <ul style="list-style-type: none"> <li>Created by Donald Knuth</li> <li>Precisely controls the interface of content.</li> <li>Type of <b>Literate Programming</b> – Logic is in natural language and code is interspersed.</li> </ul>	<p><b>LaTeX</b></p> <ul style="list-style-type: none"> <li>Developed by Leslie Lamport. Derives from TeX.</li> <li>Type of <b>Domain Specific Language (DSL)</b> – A <b>computer language that is specialized for a particular application domain</b>.</li> <li>Enforces <b>separation of concerns</b> – Design principle for separating a computer program into different sections, such that each section addresses a separate concern. <ul style="list-style-type: none"> <li>Example: LaTeX separates formatting from content.</li> </ul> </li> <li><b>Literate Programming</b></li> </ul>	<p><b>Specify Document Type</b></p> <pre>\documentclass{article}</pre> <p><b>Specify Title Block Content</b></p> <pre>\title{Hello World!}</pre> <p><b>Start Document</b></p> <pre>\begin{document}</pre> <p><b>Generate Title from Title Information</b></p> <pre>\title{Hello World!}</pre> <p><b>Close the Document</b></p> <pre>\end{document}</pre>	<p><b>Cross-Reference</b></p> <pre>\ref{&lt;referenceName&gt;}</pre> <p><b>Reference a Bibliography Citation</b></p> <pre>\cite{&lt;citationName&gt;}</pre> <p><b>Create a Reference</b></p> <pre>\label{&lt;referenceName&gt;}</pre> <p><b>Create a Bibliography</b></p> <pre>\bibliography{&lt;bibFileName&gt;}</pre> <p><b>Create a List</b></p> <pre>\begin{itemize} \item Text for #1 \item Text for #2 \end{itemize}</pre>
---	--	--	--

<p><b>Create Section with Label</b></p> <pre>\section{Section #1} \label{sec:one}</pre> <p><b>Create Subsection with Label</b></p> <pre>\subsection{&lt;SubsectionName&gt;} \label{sec:&lt;refName&gt;}</pre> <p><b>Use of Tilde (~)</b></p> <p>Creates an undividable space so the text "Section~\ref{sec:one}" will appear on one line</p>	<p><b>BibTeX</b></p> <ul style="list-style-type: none"> <li>References are tedious to <b>reformat</b> and <b>renumber</b>.</li> <li>Reference details shorted in a <b>"*.bib"</b> file.</li> </ul> <p><b>Create a Bibliography</b></p> <pre>\bibliography{biblio}</pre> <p>BibTeX filename for the example would be <b>"biblio.bib"</b></p> <p><b>Define Bibliography Style</b></p> <pre>\bibliographystyle{plainurl}</pre>	<p><b>BibTeX Article Reference Example</b></p> <pre>@article{citationName, author = {Donald Knuth}, title = {Literate Programming}, journal = {}, year = {1984}, volume = {27}, number = {2}, pages = {97-111}, }</pre>
--	---	---

## Lecture #07 – Types and Typeclasses

<p><b>Maybe Type</b></p> <ul style="list-style-type: none"> <li><b>Example of an algebraic data type</b></li> <li>Enables behavior similar to <b>null</b> in Java</li> <li><b>Used when:</b> <ul style="list-style-type: none"> <li>A function may not return a value</li> <li>A caller may not pass an argument</li> </ul> </li> <li><b>Definition:</b></li> </ul> <pre>data Maybe a = Nothing                 Just a</pre>	<p><b>Maybe "Divide" Example</b></p> <pre>divide :: Int -&gt; Int -&gt; Maybe Int divide _ 0 = Nothing divide x y = Just \$ x `div` y</pre> <pre>&gt; divide 5 2 2 &gt; divide 4 0 Nothing</pre> <p><b>DO NOT FORGET THE Just IN CORRECT SOLUTION</b></p>	<p><b>Maybe Map Example</b></p> <pre>import Data.Map  m = Map.empty m' = Map.insert "a" 42 m case (Map.lookup "a") of   Nothing -&gt; error "Element not in map"   Just x -&gt; putStrLn \$ show x</pre> <p>Since element may not be in the map, you need to use a maybe</p>
--	---	--

<p><b>Algebraic Data Type</b></p> <ul style="list-style-type: none"> <li>A <b>composite data type</b> (i.e. a type made from other types).</li> <li><b>Created via the Keyword: data</b></li> <li><b>Examples:</b> Either, Maybe, Tree</li> </ul>	<p><b>Example Algebraic Data Type</b></p> <pre>data Tree k = EmptyTree               Node (Tree k) (Tree k) val               deriving (Show)</pre> <p><b>k – Type parameter. Specifies a type not a value.</b></p> <p><b>Node: Value Constructor that creates values of type "Tree k"</b></p>	<ul style="list-style-type: none"> <li><b>Tree and Tree Int have no types since they themselves form a concrete type.</b></li> <li><b>Node</b> does have a type:</li> </ul> <pre>&gt; :t Node Node :: (Tree k) -&gt; (Tree k) -&gt; k -&gt; (Tree k)</pre> <p><b>Explanation:</b> To make a complete <b>Node</b> object, you pass it two objects of type <b>"Tree k"</b> and another object of type <b>"k"</b> and that returns a <b>"Tree k"</b> object.</p>
	<p><b>Partially Applying a Value Constructor</b></p> <ul style="list-style-type: none"> <li>Value constructors can be partially applied similar to functions.</li> </ul> <p><b>Example:</b></p> <pre>&gt; let leaf = Node EmptyTree EmptyTree &gt; Node (leaf 3) (leaf 7) 5</pre> <p>This creates a three node tree with value 5 at the root and values 3 and 7 at the leaves.</p>	<p><b>Type of the "+" Operator</b></p> <pre>&gt; :t (+) (+) :: (Num a) =&gt; a -&gt; a -&gt; a</pre> <p><b>Explanation:</b> The plus sign takes two numbers of type <b>"a"</b> and returns an object of type <b>"a"</b>.</p> <p><b>Type of a Number</b></p> <pre>&gt; :t 3 3 :: (Num a) =&gt; a</pre> <p><b>Explanation:</b> Since <b>"3"</b> has no explicit type, it can for now be any type that satisfies the <b>"Num"</b> type class.</p>

Kinds		Typeclasses	
<ul style="list-style-type: none"> <li>• <b>"The type of types"</b>.</li> <li>• <b>Concrete types have a kind of "*"</b></li> <li>• <b>Keyword</b> :k, :kind</li> <li>• <b>Example:</b></li> </ul> <pre>&gt; :k Tree Tree :: * -&gt; *</pre> <p><b>Explanation:</b> A Tree requires one type parameter to be made a concrete type.</p>	<p><b>String Kind</b></p> <pre>&gt; :kind String String :: *</pre> <p><b>Map Kind</b></p> <pre>&gt; :k Map Map :: * -&gt; * -&gt; *</pre> <p><b>Maybe Kind</b></p> <pre>&gt; :k Maybe Map :: * -&gt; *</pre> <p><b>Map String Kind</b></p> <pre>&gt; :kind (Map String) (Map String) :: * -&gt; *</pre> <p><b>Explanation:</b> Map String is has one of the two type parameters filled so it has one less asterisk.</p>	<ul style="list-style-type: none"> <li>• <b>Similar to interfaces in Java.</b> <ul style="list-style-type: none"> <li>○ Like a contract.</li> <li>○ <b>Implementation details can be included in typeclass definition.</b></li> </ul> </li> <li>• No relation to classes in object-oriented programming. <ul style="list-style-type: none"> <li>○ <b>Example:</b> Do not have any data associated with them.</li> </ul> </li> <li>• <b>Simplify polymorphism.</b></li> </ul> <p><b>Example:</b> Eq Typeclass</p> <pre>class Eq a where   (==) :: a -&gt; a -&gt; Bool   (/=) :: a -&gt; a -&gt; Bool   x == y = not (x /= y)   x /= y = not (x == y)</pre> <p>The last two lines in the type class definition allow the developer to program either (==) or (/=) but not necessarily both.</p>	<p><b>Example:</b> Make <b>Maybe</b> an Instance of <b>Eq</b></p> <pre>instance (Eq a) =&gt; Eq (Maybe a) of   (==) Nothing Nothing = true   (==) (Just x) (Just y) = x == y   (==) _ _ = false</pre> <p>Need to ensure type "a" supports "Eq" so add that as a <b>class constraint</b>.</p> <p><b>Class Constraint</b></p> <ul style="list-style-type: none"> <li>• <b>Operator:</b> =&gt;</li> <li>• Ensures that a type parameter satisfies some type class requirement.</li> </ul> <p><b>Kind of Typeclasses</b></p> <pre>&gt; :k Eq Eq :: * -&gt; Constraint  &gt; :k Num Num :: * -&gt; Constraint</pre> <p><b>Note:</b> Typeclasses are a class constraint (not a type) so their kind is different.</p>

## Lecture #08 – Functors

<p><b>Functor Type Class Definition</b></p> <pre>class Functor f where   fmap :: (a -&gt; b) -&gt; f a -&gt; f b</pre> <p>This is very similar to the definition of the higher order function "map"</p> <pre>map :: (a -&gt; b) -&gt; [a] -&gt; [b]</pre>	<p><b>Functor – Something that can be mapped over.</b></p> <ul style="list-style-type: none"> <li>• Handles things "inside a box"</li> </ul> <p><b>Example:</b> List ([]) as an Instance of Functor</p> <pre>instance Functor [] where   fmap = map</pre> <p><b>Explanation:</b> map is a specialized version of fmap for lists.</p>	<p><b>Examples: map and fmap on Lists</b></p> <pre>&gt; map (+1) [1, 2, 3] [2, 3, 4]  &gt; fmap (+1) [1, 2, 3] [2, 3, 4]  &gt; fmap (+1) [] []</pre>	<p><b>Examples: fmap on Maybes</b></p> <pre>&gt; fmap (+1) (Just 3) Just 4  &gt; fmap (+1) Nothing Nothing</pre>
<p><b>Example:</b> Maybe as an Instance of Functor</p> <pre>instance Functor Maybe where   fmap _ Nothing = Nothing   fmap f (Just x) = Just (f x)</pre> <p><b>DO NOT FORGET THE Just IN VALID SOLUTION</b></p>	<p><b>Either Algebraic Data Type</b></p> <pre>data Either a b = Left a                   Right b   deriving (Eq,Ord,Read,Show)</pre> <ul style="list-style-type: none"> <li>• <b>Left – Error type that is not mappable.</b></li> <li>• <b>Right – Expected type</b></li> </ul>	<p><b>Example:</b> Either as an Instance of Functor</p> <pre>instance Functor (Either a) where   fmap _ (Left x) = Left x   fmap f (Right y) = Right (f y)</pre> <pre>&gt; fmap (+1) Leftt 20 20 -- No Change  &gt; fmap (+1) Right 20 21 -- Changed</pre>	

## IO in Haskell

<ul style="list-style-type: none"> <li>• Haskell avoids side effects but they are inevitable in real programs.</li> <li>• <b>Monads</b> <ul style="list-style-type: none"> <li>○ Related to Functors</li> <li>○ Compartmentalize side effects.</li> </ul> </li> <li>• <b>()</b> <ul style="list-style-type: none"> <li>○ Unit type in Haskell</li> </ul> </li> </ul>	<p><b>Type Signature of the main Function in Haskell</b></p> <pre>main :: IO ()</pre> <p><b>Hello World in Haskell</b></p> <pre>main = putStrLn "Hello World"</pre> <p><b>Type Signature of getLine</b></p> <pre>getLine :: IO String</pre>	<ul style="list-style-type: none"> <li>• <b>do</b> – Allows for the chaining of multiple IO commands together</li> <li>• <b>&lt;-</b> Extracts data out of an "IO Box"</li> <li>• <b>return</b> – Places data into an "IO Box"</li> </ul>	
--	---	---	--



<p><b>do Example</b></p> <pre>main = do   line &lt;- getLine   if null line -- Checks for empty str   then return ()   else putStrLn \$ reverseWords line  reverseWords :: String -&gt; String reverseWords = unwords .   map reverse . words</pre>	<p><b>return in Haskell</b></p> <ul style="list-style-type: none"> <li>• <b>Unrelated to “return” in other languages</b></li> <li>• <b>Better described as “wrap” or “box”</b></li> </ul> <p><b>Summary:</b>  <b>return</b> – Boxes an IO (since IO is a monad)  <b>&lt;-</b> Unboxes an IO</p>	<p><b>Type of the Unit Type ()</b></p> <ul style="list-style-type: none"> <li>• Base type</li> </ul> <pre>&gt; :t () () :: ()</pre> <hr/> <p><b>Type of return</b></p> <pre>&gt; :t (return ()) (return ()) :: Monad m =&gt; m ()</pre> <p><b>Monad is a typeclass.</b></p>
---	---	---

<p><b>Using IO as a Functor</b></p> <pre>main = do   line &lt;- fmap (++"!!!") getLine   putStrLn line</pre> <p><b>Explanation:</b> This function takes a string input from standard in and appends “!!!” at which point it prints it to the console.</p>	<p><b>Definition of IO as a Functor</b></p> <pre>instance Functor IO where   fmap f action = do     result &lt;- action     return (f result)</pre> <p><b>Explanation:</b> The action object is taken out of the IO box, the function “f” applied to it, and then returned to the IO box.</p>	<p><b>id Function</b></p> <ul style="list-style-type: none"> <li>• <b>Takes one input parameter and returns that input parameter unmodified. Examples:</b></li> </ul> <pre>&gt; id 3 3  &gt; id "Hello World" "Hello World"</pre>
---	---	---

## Functor Laws

<p><b>Functor Law #1:</b> If we map the id function over a Functor, the Functor that we get back should be the same as the original Functor.</p> <p><b>Examples:</b></p> <pre>&gt; fmap id (Just 3) Just 3 &gt; fmap id Nothing Nothing &gt; fmap id [1, 2, 3] [1, 2, 3]</pre>	<p><b>Functor Law #2:</b> Composing two functions and then mapping the resulting (composed) function over a Functor should be the same as first mapping one function over the Functor and then mapping the other one.</p> <p><b>Law #2 Written Formally</b></p> <pre>fmap (f . g) = fmap f . fmap g</pre>	<p>The Functor laws are NOT enforced. They are good practice that makes the code easier to reason about.</p>
--	---	--

## Lecture #09 – Applicative Functors

<p><b>Functor – Something that can be mapped over. Allow you to map functions over different data types. Examples:</b></p> <ul style="list-style-type: none"> <li>• Maybe</li> <li>• Either</li> <li>• IO</li> <li>• Lists</li> <li>• &lt;*&gt;</li> </ul> <p><b>Functors return boxed up values.</b></p>	<p><b>Functor Example</b></p> <pre>&gt; fmap (+1) [1, 2, 3] [2, 3, 4]  &gt; let x = fmap (+) [1, 2, 3]</pre> <p><b>Explanation:</b> In this case x is: [(1+), (2+), (3+)]</p>	<p><b>Applicative Functor</b></p> <ul style="list-style-type: none"> <li>• Requires the importing of a special library as shown below:</li> </ul> <pre>import Control.Applicative</pre> <p>Functions in Applicative Typeclass:</p> <ul style="list-style-type: none"> <li>• <b>pure</b> – Wraps/boxes a value</li> <li>• <b>&lt;*&gt;</b> - Infix version of <b>fmap</b>. Is itself a Functor.</li> </ul>	<p><b>Example Uses of pure</b></p> <pre>&gt; pure 7 7  &gt; pure 7 :: Maybe Int Just 7  &gt; pure 7 :: [Int] [7]</pre>
---	---	---	--

<p><b>Type Class Definition of Applicative</b></p> <pre>class (Functor f) =&gt; Applicative f where   pure :: a -&gt; f a   &lt;*&gt; :: f (a -&gt; b) -&gt; f a -&gt; f b</pre> <p><b>Only difference between &lt;*&gt; and fmap is that the function in &lt;*&gt; is boxed while it is not in fmap (see the green f).</b></p>	<p><b>Make Maybe an Instance of Applicative</b></p> <pre>instance Applicative Maybe where   pure = Just   Nothing &lt;*&gt; _ = Nothing   (Just f) &lt;*&gt; x = fmap f x</pre> <p><b>Explanation:</b> pure simply wraps the value in Just. No need to explicitly check if “x” is maybe as fmap will do that for you.</p>	<p><b>Examples of Applicative Maybe</b></p> <pre>&gt; Just (+3) &lt;*&gt; Just 4 Just 7 &gt; pure (+3) &lt;*&gt; Just 4 Just 7 &gt; pure (+) &lt;*&gt; Just 3 &lt;*&gt; Just 4 Just 7 &gt; (+) &lt;*&gt; Just 3 &lt;*&gt; Just 4 Just 7</pre> <p><b>Explanation: x &lt;*&gt; is fmap as an infix operator. It is NOT necessarily the same as pure x &lt;*&gt;</b></p>
---	---	---

<p><b>Making [] an Instance of Applicative</b></p> <pre>instance Applicative [] where   pure x = [x]   fs &lt;*&gt; xs = [f x   f &lt;- fs, x &lt;- xs]</pre> <p><b>Explanation:</b> The function is actually a list of functions so list comprehension is needed.</p>	<p><b>Example Use of Applicative on Lists</b></p> <pre>&gt; (*) &lt;*&gt; [1, 2, 3] &lt;*&gt; [1,0,0,1] [1,0,0,1,2,0,0,2,3,0,0,3]  &gt; pure 7 7 -- No change &gt; pure 7 :: [Int] [7]</pre>	<p><b>Definition of IO as an Instance of Applicative</b></p> <pre>instance Applicative IO where   pure = return   a &lt;*&gt; b = do     f &lt;- a     x &lt;- b     return (f x)</pre>
--	--	---

<p><b>Example of Applicative IO</b></p> <pre>import Control.Applicative  main = do   a &lt;- (++) &lt;\$&gt; getLine &lt;*&gt; getLine   putStrLn a</pre>	<p><b>liftA2</b></p> <p>A function that simplifies the application of a normal function to two Functors.</p> <pre>liftA2 :: (Applicative f) =&gt; (a -&gt; b -&gt; c) -&gt; f a -&gt; f b -&gt; fc liftA2 f x y = f &lt;\$&gt; a &lt;*&gt; b</pre>
---	--

<p><b>Example of liftA2</b></p> <pre>&gt; (:) &lt;\$&gt; Just 3 &lt;*&gt; Just [4] Just [3, 4] &gt; liftA2 (:) (Just 3) (Just [4]) Just [3, 4]</pre>	
--	--

## Applicative Functor Laws

<p><b>Law 1:</b></p> <pre>pure f &lt;*&gt; x = fmap f x</pre>	<p><b>Law 2:</b></p> <pre>pure id &lt;*&gt; v = v</pre>	<p><b>Law 3:</b></p> <pre>pure (.) &lt;*&gt; u &lt;*&gt; v &lt;*&gt; w = u &lt;*&gt; (v &lt;*&gt; w)</pre>
<p><b>Law 4:</b></p> <pre>pure f &lt;*&gt; pure x = pure (f x)</pre>	<p><b>Law 5:</b></p> <pre>u &lt;*&gt; pure y = pure (\$y) &lt;*&gt; u</pre>	<p>Similar to Functor Laws, these are not strictly enforced but are good practice to make it easier to reason about the code.</p>

## Monoids

<p><b>Monoid:</b> An associative binary function and a value that acts as an identity with respect to that function.</p> <p><b>Examples</b></p> <ul style="list-style-type: none"> <li>• x / 1 Identity of <b>Division</b></li> <li>• x * 1 Identity of <b>Multiplication</b></li> <li>• lst ++ [] Identity of <b>Concatenation</b></li> <li>• x + 0 Identity of <b>Addition</b></li> <li>• x - 0 Identity of <b>Subtraction</b></li> </ul>	<p><b>Definition of Monoid Typeclass</b></p> <pre>class Monoid m where   mempty :: m   mappend :: m -&gt; m -&gt; m   mconcat :: [m] -&gt; m   mconcat = foldr mappend mempty</pre>	
---	---	--

## Monoid Rules

<p><b>Rule #1:</b></p> <pre>mempty `mappend` x = x</pre>	<p><b>Rule #2:</b></p> <pre>x `mappend` empty = x</pre>	<p><b>Rule #3:</b></p> <pre>(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)</pre>
--	---	---

## Lecture #10 – Monads

<p><b>Functor – Something that can be mapped over.</b></p> <p><b>Definition:</b></p> <pre>instance Functor f where   fmap :: (a -&gt; b) -&gt; f a -&gt; f b</pre>	<p><b>Problem with Functors:</b> Do not support chaining of multiple commands. <b>Example:</b></p> <pre>&gt; fmap (+) (Just 3) (Just 4)</pre> <p>Returns an error since it cannot resolve (Just 3+) and (Just 4)</p>	<p><b>Applicative Functor:</b> A Functor that can be applied to other Functors.</p> <pre>class (Functor f) =&gt; Applicative f where   (&lt;*&gt;) :: f (a -&gt; b) -&gt; f a -&gt; f b</pre> <p>Requires library <b>Control.Applicative</b></p>
--	--	--

<p>Even <b>with Applicative Functors</b>, it is not possible to chain together multiple commands. <b>Example:</b></p> <pre>&gt; Just (+3) &lt;*&gt; Just (+4) &lt;*&gt; Just (+5)</pre> <p>Returns error</p>	<p><b>Monads: Each the chaining together of multiple functions.</b></p> <p><b>Key Operator:</b></p> <pre>&gt;&gt;= (Bind)</pre>	<p><b>Example #1:</b> Using Just</p> <pre>&gt; (Just 3) &gt;&gt;= (\x -&gt; Just (x + 4)) &gt;&gt;= (\y -&gt; Just (y+5)) 12</pre> <p><b>Example #2:</b> Using return</p> <pre>&gt; (return 3) &gt;&gt;= (\x -&gt; return (x + 4)) &gt;&gt;= (\y -&gt; return (y+5)) 12</pre>
--	---	---

<p><b>Comparing &lt;*&gt; and &gt;&gt;=</b></p> <p><b>Functor:</b></p> <pre>(&lt;*&gt;) :: Applicative f =&gt; f (a -&gt; b) -&gt; f a -&gt; f b</pre> <p><b>Monad:</b></p> <pre>(&gt;&gt;=) :: Monad m =&gt; m a -&gt; (a -&gt; m b) -&gt; m b</pre> <p><b>Differences:</b></p> <ol style="list-style-type: none"> <li>1. Order of the arguments changed.</li> <li>2. The function is boxed in Functor but not Monad</li> <li>3. Monad function returns a boxed result.</li> </ol>	<p><b>Example of &lt;\$&gt;, &lt;*&gt; and &gt;&gt;=</b></p> <pre>&gt; (\x -&gt; x + 1) &lt;\$&gt; Just 3 Just 4</pre> <pre>&gt; Just (\x -&gt; x + 1) &lt;*&gt; Just 3 Just 4</pre> <pre>&gt; (Just 3) &gt;&gt;= (\x -&gt; Just (x+1)) Just 4</pre>	<p><b>Example:</b> Implement <b>applyMaybe</b> that applies a function to a <b>Maybe</b></p> <pre>applyMaybe :: Maybe a -&gt; (a -&gt; b) -&gt; Maybe b applyMaybe Nothing _ = Nothing applyMaybe (Just x) f = Just (f x)</pre>
---	--	---



<p><b>Example:</b> Implement <code>applyMaybe</code> that applies a function to a <code>Maybe</code></p> <pre> applyMaybe :: Maybe a -&gt; (a -&gt; Maybe b)               -&gt; (Maybe b)  applyMaybe Nothing _ = Nothing applyMaybe (Just x) f = Just (f x) </pre>	<p><b>Chaining <code>applyMaybe</code></b></p> <pre> &gt; (Just 3) \x -&gt; Just (x*2) \y -&gt; Just (y-1) Just 5  &gt; (Just 3) \_ -&gt; Nothing \y -&gt; Just (y-1) Nothing </pre>	
--	--	--

<p><b>Monad Typeclass Definition</b></p> <pre> class Monad m where   return :: a -&gt; m a   (&gt;&gt;=) :: m a -&gt; (a -&gt; m b) -&gt; m b    (&gt;&gt;) :: m a -&gt; m b -&gt; m b   x &gt;&gt; y = x &gt;&gt;= (\_ -&gt; y) --Lamda    fail :: String -&gt; m a   fail msg = error msg </pre>	<p><b>Example a Robot Moving Towards a Goal (Not Failure)</b></p> <pre> --Location type Robot = (Int, Int)  -- Functions up (x,y) = (x, y+1) down (x,y) = (x, y-1) left (x,y) = (x-1, y) right (x,y) = (x+1, y)  -- Define Operator and start location x -: f = f x start = (0, 0)  &gt; start -: up -: right (1, 1)  &gt; start -: up -: left -: left -: right -: down (-1, 0) </pre>
--	--

<p><b>Maybe as an Instance of the Monad Typeclass</b></p> <pre> instance Monad Maybe where    return = Just    (&gt;&gt;=) Nothing _ = Nothing   (&gt;&gt;=) (Just x) f = Just (f x)    fail _ = Nothing </pre>	<p><b>Example a Robot Moving Towards a Goal (with Failure)</b></p> <pre> -- Once the goal is reached, -- the robot stops goal := Map.empty       -: (Map.insert (0, 2) True)       -: (Map.insert (-1, 3) True)       -: (Map.insert (-3, -8) True)  moveTo :: Pos -&gt; Maybe Pos moveTo p = if Map.member p goal            then Nothing            else Just p  -- Since these are in bind, no need -- to handle Nothing. Bind handles it. up (x,y) = moveTo (x, y+1) down (x,y) = moveTo (x, y-1) left (x,y) = moveTo (x-1, y) right (x,y) = moveTo (x+1, y) </pre>	<pre> start = (0, 0)  &gt; return start &gt;= up &gt;= left &gt;= left   &gt;= right &gt;= down Just (-1, 0)  &gt; return start &gt;= left &gt;= left &gt;= up   &gt;= up &gt;= right &gt;= up   &gt;= right &gt;= right &gt;= down Nothing  Explanation: Reached one of the goals (-1, 3) at the red up </pre>
---	---	---

## Integer Division Using Monads

<p><b>Integer Division with Bind and No "do"</b></p> <pre> mydiv :: Maybe Int -&gt; Maybe Int -&gt; Maybe Int mydiv x y = x &gt;&gt;= (\number -&gt;   y &gt;&gt;= (\denom -&gt;     if denom &gt; 0     then Just (div number denom)     else fail "Div by zero") </pre>	<p><b>Integer Division with Bind with "do"</b></p> <pre> mydiv :: Maybe Int -&gt; Maybe Int -&gt; Maybe Int mydiv x y = do   number &lt;- x   denom &lt;- y   if denom &gt; 0   then Just (div number denom)   else fail "Div by 0" </pre>	<p><b>Integer Division with Bind with "do" and return</b></p> <pre> mydiv :: Maybe Int -&gt; Maybe Int -&gt; Maybe Int mydiv x y = do   number &lt;- x   denom &lt;- y   if denom &gt; 0   then return \$ div number denom   else fail "Div by 0" </pre>
---	--	--

## List Monad

<p><b>Making List an Instance of Monad</b></p> <pre> instance Monad [] where   return x = [x]   (&gt;&gt;=) xs f = concat(map f xs)   fail _ = [] </pre> <p><b>Explnation:</b> <code>concat</code> is needed here as <code>f</code> returns elements already in a list. As such, <code>concat</code> merges the individual lists (from each call to <code>f</code>) into a single list.</p>	<p><b>Example Use of List as a Monad</b></p> <pre> listOfTuples :: [(Int, Char)] listOfTuples = do   n &lt;- [1, 2]   ch &lt;- ['a', 'b']   return (n, ch)  &gt; listOfTuples [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')] </pre>	
---	---	--

## Miscellaneous

<p><b>Kind of Show and show</b></p> <pre>&gt; :k Show Show :: * -&gt; Constraint</pre> <p><b>Type and Kind of show</b></p> <pre>&gt; :k show Error (A function not a type) &gt; :t show show :: (Show a) =&gt; a -&gt; String</pre>	<p><b>Lambda and ADT Combined</b></p> <pre>&gt; (\x -&gt; Just (x+1)) 1 Just 2</pre> <p><b>Creating Type Alias</b></p> <pre>type String = [Char]</pre> <p>Allows for more readable code as developer can use a type name that makes more sense for a given application.</p>	<p><b>Example:</b> <code>applyMaybe</code> that takes a <code>(Maybe a)</code> and applies to it a function that takes a normal <code>a</code> and returns a <code>(Maybe b)</code></p> <pre>applyMaybe :: (Maybe a) -&gt; (a -&gt; Maybe b) -&gt; (Maybe b) applyMaybe Nothing _ = Nothing applyMaybe (Just x) f = f x</pre> <p><b>Explanation:</b> Since the function “<code>f</code>” already returns a <code>Maybe</code>, you do not need to re-box it. However, since it does not take a <code>Maybe</code>, you need to unbox the first input parameter.</p>
<p><b>Applying return to Items</b></p> <pre>&gt; return 7 7 return &gt; return 7 :: Maybe Int Just 7 &gt; return 7 :: [Int] [7] -- Need Int or get an error</pre> <p><b>Conclusion:</b> Behavior for <code>return</code> is the same as <code>pure</code>. Both put the object in the minimum default context that still yields that value.</p>	<p>List comprehension is syntactic sugar for using lists as monads.</p>	
<p><b>Monads and Lambda</b></p> <p>When trying to chain multiple functions together in a <code>Monad</code>, remember the <code>Monad</code> must return a <b>boxed value</b>. Hence, <code>Lambda</code> often work well as they simplifying boxing.</p>	<p><b>Applicative Typeclass</b> – Allows you to use normal functions on values that have a context (i.e. are inside a <code>Functor</code>).</p> <p><b>Monad:</b> Given a value of type, <code>a</code>, in a context, <code>m</code>, apply a function that takes a normal value of type <code>a</code> and returns a value in the context <code>m</code>.</p> <pre>(&gt;&gt;=) :: (Monad m) =&gt; m a -&gt; (a -&gt; m b) -&gt; m b</pre> <p><b>Monads are just applicative functors that support bind (&gt;&gt;=).</b></p> <p><b>Key Difference:</b> <code>Applicative</code> functors support normal functions that take and return unboxed values while <code>Monads</code> return boxed values.</p>	<p><b>return</b> – <code>Monad</code> equivalent of “pure” for <code>Applicative</code> <code>Functors</code>.</p> <p><b>Cannot use <code>fmap</code> in the definition of a <code>Monad</code> since <code>fmap</code> returns a boxed value while the function of the <code>Monad</code> returns a boxed value. Hence, if you used <code>fmap</code> with a <code>Monad</code>, you would return a double boxed value.</b></p>