

# CS252 – Final 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-core “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>• JavaScript rewriting</li> <li>• Instrumentation</li> <li>• Program Analyzers</li> <li>• IDEs</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>◦ A function call 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> <pre>&gt; let f x = x + 1 &gt; f 3 4</pre>	
		<b>Run Haskell from Command Line</b> Use <b>runhaskell</b> keyword.  <b>Example:</b> <pre>&gt; runhaskell &lt;FileName&gt;.hs</pre>	
		<b>Hello World in Haskell</b> <pre>main :: IO () main = do     putStrLn "Hello World"</pre>	

<div>Primitive Classes in Haskell</div> <div>1. <b>Int</b> – <b>Bounded</b> Integers</div> <div>2. <b>Integer</b> – <b>Unbounded</b></div> <div>3. <b>Float</b></div> <div>4. <b>Double</b></div> <div>5. <b>Bool</b></div> <div>6. <b>Char</b></div>	<div>Lists</div> <div><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></div> <div><ul style="list-style-type: none"><li>◦ <b>last</b> Last element in the list</li><li>◦ <b>init</b> All elements in the list except the last one</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 <b>l</b> containing only <b>m</b></li><li>◦ <b>repeat m</b> Create an infinite list containing only <b>m</b></li></ul></div>		<div>Ranges</div> <div><ul style="list-style-type: none"><li>• Can be infinite or bounded</li><li>• Use the “<b>..</b>” notation. <b>Examples:</b></li></ul><div>&gt; [1..4] [1, 2, 3, 4]</div><div>&gt; [1,2..6] [1, 2, 3, 4, 5, 6]</div><div>&gt; [1,3..10] [1, 3, 5, 7, 9]</div><div>&gt; [5, 4..1] [5, 4, 3, 2, 1]</div></div>
	<div>Hello World in Haskell</div> <div>main :: IO ()</div> <div>main = do</div> <div>putStrLn "Hello World"</div>	<div>List Examples</div> <div>&gt; putStrLn \$ "Hello " ++ "World"</div> <div>"Hello World"</div> <div>&gt; let s = bra in s !! 2 : s ++ 'c' : last s : 'd' : s</div> <div>"abracadabra"</div>	<div>Infinite List Example</div> <div>&gt; let even = [2,4..]</div> <div>&gt; take 5 even</div> <div>[2, 4, 6, 8, 10]</div>

<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>	<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>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>
<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>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>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; max xs then x else max xs &gt; where max xs = maxNum xs</pre>	<p><b>Reasons for a Large Number of Programming Languages</b></p> <ul style="list-style-type: none"> <li>Different domains</li> <li>Different design choices</li> </ul>
--	--	---

<p><b>Recursion</b></p> <ul style="list-style-type: none"> <li>:t or :type – Gets the type of a variable or function.</li> </ul> <p>Example:</p> <pre>&gt; :type 'A' 'A' :: Char &gt; :t "Hello" "Hello" :: [Char]</pre>	<p><b>Haskell's Base Typeclasses</b></p> <ul style="list-style-type: none"> <li>Ord – Can be ordered</li> <li>Eq – Can perform equality check</li> <li>Show – Can convert to String</li> <li>Read – Can convert from String</li> <li>Enum – Sequentially Ordered</li> <li>Bounded – Has upper and lower bound.</li> </ul>	
--	---	--

## Lecture #03 – Operational Semantics

<div><div>Formal Semantics</div><div>Crisply define how the language features work.</div></div>	<div><div>Formal Semantic Styles</div><div><ul style="list-style-type: none"><li>Operational – Specify how expressions should be evaluated.<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></div></div>	<div><div>A Review of Compilers</div><div></div></div>				
<div><div>Abstract Syntax Tree</div><div>Tree representation of the abstract syntactic structure of a program’s source code. Example is Bool* language below.</div></div>	<div><div>Big Step Operational Semantics</div><div><ul style="list-style-type: none"><li>Evaluates every expression to a value</li></ul></div></div>					
<div><div>Bool * Language</div><div><table><tr><td><div><div>e ::=</div><div>true</div><div>  false</div><div>  if e</div><div>  then e</div><div>  else e</div></div></td><td><div>Expressions:</div><div>constant true</div><div>constant false</div><div>conditional</div></td></tr><tr><td><div><div>v ::=</div><div>true</div><div>  false</div></div></td><td><div>Values:</div><div>constant true</div><div>constant false</div></td></tr></table></div></div>	<div><div>e ::=</div><div>true</div><div>  false</div><div>  if e</div><div>  then e</div><div>  else e</div></div>	<div>Expressions:</div> <div>constant true</div> <div>constant false</div> <div>conditional</div>	<div><div>v ::=</div><div>true</div><div>  false</div></div>	<div>Values:</div> <div>constant true</div> <div>constant false</div>	<div><div><math display="block">e \Downarrow v</math></div><div><ul style="list-style-type: none"><li>Read as: “Expression e evaluates to the value v”</li></ul></div></div>	
<div><div>e ::=</div><div>true</div><div>  false</div><div>  if e</div><div>  then e</div><div>  else e</div></div>	<div>Expressions:</div> <div>constant true</div> <div>constant false</div> <div>conditional</div>					
<div><div>v ::=</div><div>true</div><div>  false</div></div>	<div>Values:</div> <div>constant true</div> <div>constant false</div>					

<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:             <math display="block">e \rightarrow e' \rightarrow v</math> </li> <li>→* : Many evaluation steps required. Example:             <math display="block">e \rightarrow^* v</math> </li> </ul>	<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" with "E-If"</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>• <b>0</b> : The Number "0"</li> <li>• <b>succ 0</b> : Represents "1"</li> <li>• <b>succ succ 0</b> : Represents "2"</li> <li>• <b>pred n</b> : 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>• <b>Code lines begin with "&gt;"</b></li> <li>• <b>All other lines are comments.</b></li> <li>• "Essentially swaps code with comments."</li> </ul>	<p><b>Case Statement in Haskell</b></p> <ul style="list-style-type: none"> <li>• <b>Keywords:</b> <b>case</b>, <b>of</b>, <b>otherwise</b></li> <li>• <b>Operator:</b> <b>-&gt;</b></li> </ul> <p><b>Example:</b></p> <pre> case x of   val1 -&gt; "Value 1"   val2 -&gt; "Value 2"   otherwise -&gt; "Everything else."         </pre>
--	--	---	---

## Lab #02 Review

<p><b>Bool Expression Type</b></p> <pre> &gt; data BoolExp = BTrue     BFalse     Bif BoolExp BoolExp BoolExp     B0     Bsucc BoolExp     Bpred BoolExp   deriving Show         </pre>	<p><b>BoolVal Type</b></p> <pre> &gt; data BoolVal = BVTrue     BVFalse     BVNum BVInt   deriving Show  &gt; data BVInt = BV0     BVSucc BVInt   deriving Show         </pre>	<p><b>Type Constructors:</b> BoolExp, BoolVal, BVInt</p> <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

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

<p><b>Iterative vs. Recursive</b></p> <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>	<p><b>Not Tail Recursive</b></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>	<p><b>Tail Recursive Factorial</b></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>
--	---	--

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

## Higher Order Functions

<p><b>Functions in Functional Programming</b></p> <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 <i>f</i> is a rule that associates to each <i>x</i> from some set <i>X</i> of values a unique <i>y</i> from a set of <i>Y</i> values.</li> </ul> $(x \in X \wedge y \in Y) \rightarrow y = f(x)$ <ul style="list-style-type: none"> <li>• <i>f</i> – Name of the function</li> <li>• <i>x</i> – Independent variable</li> <li>• <i>y</i> – Dependent variable</li> <li>• <i>X</i> – Domain</li> <li>• <i>Y</i> – Range</li> </ul>	<p><b>Qualities of Functional Programming</b></p> <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>◦ <b>Passed as arguments to a function</b></li> <li>◦ <b>Be returned from a function</b></li> <li>◦ <b>Construct new functions dynamically</b></li> </ul> </li> </ul>	<p><b>Higher Order Function</b></p> <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></p> <p><b>Transform a function with multiple arguments into multiple functions that each take exactly one argument.</b></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 <b>thunks</b> 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 <b>thunks</b>)</li> <li><b>Good for large, but finite lists.</b></li> </ul>	<p><b>foldl in terms of foldr</b></p> <pre>myFoldl' f acc x = foldr (flip f) acc (reverse x)</pre>

## 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>	<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 <b>v</b> 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> <pre>if false then e2 else e3 → e3</pre> <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> <pre>if true then (if true then false else false) else true</pre> <p>This reduces to "if true then false else false"</p>	<p><b>Example: Not a Redex</b></p> <pre>if (if true then false else false) then true else true</pre> <p>Not a redex as expression "if true then false else false" 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]</pre> <pre>= 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).</b> Example:</p> <p><b>EC-IfFalse:</b></p> <pre>C[if false then e2 else e3] → C[e3]</pre> <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> import Data.Map as <b>Map</b></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> <li><b>Map.member k m</b> – Returns true if k is in map "m" and false otherwise.</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

<b>TeX</b> <ul style="list-style-type: none"> <li>Created by Donald Knuth</li> <li><b>Domain specific language for typesetting documents.</b></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. <i>"Mark code instead of marking comments."</i></li> </ul>	<b>LaTeX</b> <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 <b>separating a computer program into different sections, such that each section addresses a separate concern.</b> <ul style="list-style-type: none"> <li><b>Example:</b> LaTeX separates formatting from content.</li> </ul> </li> <li><b>Literate Programming</b></li> </ul>	<b>Specify Document Type</b> <code>\documentclass{article}</code> <b>Specify Title Block Content</b> <code>\title{Hello World!}</code> <b>Start Document</b> <code>\begin{document}</code> <b>Generate Title from Title Information</b> <code>\title{Hello World!}</code> <b>Close the Document</b> <code>\end{document}</code>	<b>Cross-Reference</b> <code>\ref{&lt;referenceName&gt;}</code> <b>Reference a Bibliography Citation</b> <code>\cite{&lt;citationName&gt;}</code> <b>Create a Reference</b> <code>\label{&lt;referenceName&gt;}</code> <b>Create a Bibliography</b> <code>\bibliography{&lt;bibFileName&gt;}</code> <b>Create a List</b> <code>\begin{itemize}</code> <code>\item Text for #1</code> <code>\item Text for #2</code> <code>\end{itemize}</code>
--	---	--	--

<b>Create Section with Label</b> <code>\section{Section #1}</code> <code>\label{sec:one}</code> <b>Create Subsection with Label</b> <code>\subsection{&lt;SubsectionName&gt;}</code> <code>\label{sec:&lt;refName&gt;}</code> <b>Use of Tilde (~)</b> Creates an undividable space so the text "Section~\ref{sec:one}" will appear on one line	<b>BibTeX</b> <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> <b>Create a Bibliography</b> <code>\bibliography{biblio}</code> BibTeX filename for the example would be <b>"biblio.bib"</b> <b>Define Bibliography Style</b> <code>\bibliographystyle{plainurl}</code>	<b>BibTeX Article Reference Example</b> <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

<b>Maybe Type</b> <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>Can be used to provide context.</li> <li><b>Used when:</b> <ul style="list-style-type: none"> <li><b>A function may not return a value</b></li> <li><b>A caller may not pass an argument</b></li> </ul> </li> <li><b>Definition:</b> <pre>data Maybe a = Nothing                 Just a</pre> </li> </ul>	<b>Maybe "Divide" Example</b> <pre>divide :: Int -&gt; Int -&gt; Maybe Int divide _ 0 = Nothing divide x y = Just \$ x `div` y  &gt; divide 5 2 2 &gt; divide 4 0 Nothing</pre> <p><b>DO NOT FORGET THE Just IN CORRECT SOLUTION</b></p>	<b>Maybe Map Example</b> <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>
---	--	---

<b>Algebraic Data Type</b> <ul style="list-style-type: none"> <li>A <b>composite data type</b> (i.e. a type made from other types).</li> <li>Created via the <b>Keyword: data</b></li> <li><b>Examples:</b> <ul style="list-style-type: none"> <li><b>Either</b></li> <li><b>Maybe</b></li> <li><b>Tree</b></li> </ul> </li> </ul>	<b>Example Algebraic Data Type</b> <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:           <pre>&gt; :t Node Node :: (Tree k) -&gt; (Tree k) -&gt; k -&gt; (Tree k)</pre> <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.         </li> </ul>
	<b>Partially Applying a Value Constructor</b> <ul style="list-style-type: none"> <li>Value constructors can be partially applied similar to functions. <b>Example:</b> <pre>&gt; let leaf = Node EmptyTree EmptyTree</pre> <pre>&gt; Node (leaf 3) (leaf 7) 5</pre> </li> </ul> <p>This creates a three node tree with value 5 at the root and values 3 and 7 at the leaves.</p>	<b>Type of the "+" Operator</b> <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>
		<b>Type of a Number</b> <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>• “The type of types”.</li> <li>• Concrete types have a kind of “*”</li> <li>• Keyword :k, :kind</li> <li>• Example: <pre>&gt; :k Tree Tree :: * -&gt; *</pre> </li> </ul> <p><b>Explanation:</b> A Tree requires one type parameter (e.g. Int) 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>○ Implementation details can be included in typeclass definition.</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 Maybe an Instance of Eq</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 typeclass requirement.</li> </ul> <p><b>Kind of Typeclasses</b></p> <pre>&gt; :k Eq Eq :: * -&gt; Constraint</pre> <pre>&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</b> – Error type that is not mappable.</li> <li>• <b>Right</b> – Expected type</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/Monad commands together. <b>Syntactic sugar for bind “&gt;&gt;=”</b></li> <li>• <b>&lt;-</b> Extracts data out of an IO/Monad “Box”</li> <li>• <b>return</b> – Places data into an IO/Monad “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>Monad is a <b>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:</b> x &lt;\$&gt; is fmap as an infix operator. It is NOT necessarily the same as pure x &lt;*&gt;. It should be based off Applicative Functor Law #1.</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>	<p><b>Applicative Functor Definition</b></p> <p><b>A functor you can apply to other Functors.</b></p>
--	---

### 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 <b>associative</b> binary function and a value that acts as an <b>identity</b> with respect to that function.</p> <p><b>Examples</b></p> <ul style="list-style-type: none"> <li><math>x * 1</math> Identity of <b>Multiplication</b></li> <li><code>lst ++ []</code> Identity of <b>Concatenation</b></li> <li><math>x + 0</math> Identity of <b>Addition</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` mempty = 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</b> – Something that can be mapped over.</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 <b>Functor</b> that can be applied to other <b>Functors</b>.</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:</b> Can chain through a series of functions.</p> <p><b>Key Operator:</b></p> <pre>&gt;&gt;= (Bind)</pre>	<p><b>Example #1:</b> Using <b>Just</b></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 <b>return</b></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>Order of the arguments changed.</li> <li>The function is boxed in Functor but not Monad</li> <li>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) \applyMaybe (\x -&gt; Just (x*2))            \applyMaybe (\y -&gt; Just (y-1)) Just 5  &gt; (Just 3) \applyMaybe (\_ -&gt; Nothing)            \applyMaybe (\y -&gt; Just (y-1)) Nothing </pre>	<p><b>Additional Names for Monoids</b></p> <ul style="list-style-type: none"> <li>• “Programmable Semicolons”</li> <li>• “Applicative Functors you can chain.”</li> </ul>
--	--	---

<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 = 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;&gt;= up &gt;&gt;= left &gt;&gt;= left   &gt;&gt;= right &gt;&gt;= down Just (-1, 0)  &gt; return start &gt;&gt;= left &gt;&gt;= left &gt;&gt;= up   &gt;&gt;= up &gt;&gt;= right &gt;&gt;= up   &gt;&gt;= right &gt;&gt;= right &gt;&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>	<p><b>Combining a Maybe and a List Monad</b></p> <pre> &gt; Just [2,3] &gt;&gt;= (\x -&gt; Just( fmap (+1) x)) [3, 4] </pre>
---	---	--

## Lecture #11 – Parsing Combinators

<p><b>Semantics:</b> Enumerate <b>what a program means</b>. Defined by the interpreter or compiler.</p> <p><b>Syntax:</b> Enumerate <b>how a program is structured</b>. Defined by the lexer and parser.</p>	<p><b>Compilation Flow</b></p> <p><b>Step #1:</b> Tokenizer/lexer generates a set of tokens.</p> <p><b>Step #2:</b> Parser turns the tokens into an abstract syntax tree.</p> <p><b>Step #3:</b> Compilers and interpreters convert the AST into machine code or commands respectively.</p>	<p><b>Lexer</b></p> <p>Converts the characters of the program into words of the language.</p> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• Lex/Flex (C/C++)</li> <li>• ANTLR &amp; JavaCC (Java)</li> <li>• Parsec (Haskell)</li> </ul>
--	---	---

<p><b>Categories of Tokens</b></p> <ul style="list-style-type: none"> <li>• <b>Reserved Words/Keywords.</b> <ul style="list-style-type: none"> <li>◦ <b>Examples:</b> while, if, then, else</li> </ul> </li> <li>• <b>Literals/Constants.</b> <ul style="list-style-type: none"> <li>◦ <b>Examples:</b> 123, "Hello World!"</li> </ul> </li> <li>• <b>Special symbols.</b> <ul style="list-style-type: none"> <li>◦ <b>Examples:</b> ",", "=&gt;", "&amp;&amp;"</li> </ul> </li> <li>• <b>Identifiers.</b> <ul style="list-style-type: none"> <li>◦ <b>Examples:</b> "balance", "myFunction"</li> </ul> </li> </ul>	<p><b>Parsing</b></p> <ul style="list-style-type: none"> <li>• <b>Parser converts tokens to abstract syntax trees.</b></li> <li>• <b>Defined by context free grammars (CFG)</b></li> <li>• <b>Types of Parsers:</b> <ul style="list-style-type: none"> <li>◦ <b>Bottom-up/Shift-Reduce</b> Parsers</li> <li>◦ <b>Top-down</b> parsers</li> </ul> </li> </ul>	<p><b>Context Free Grammars</b></p> <ul style="list-style-type: none"> <li>• Grammars specify the language.</li> <li>• Specified in Backus-Naur form format. <b>Example:</b></li> </ul> <pre>Expr -&gt; Number         Number + Expr</pre> <ul style="list-style-type: none"> <li>• <b>Terminal</b> – <b>Cannot be broken down</b> further.</li> <li>• <b>Non-terminals</b> – <b>Can be broken down</b> further.</li> </ul> <p><b>Example:</b> "0", "1", "2", ..., "9" are terminals but digit, number, and expression are not.</p>	<p><b>Example Grammar</b></p> <pre>expr -&gt; expr + expr         expr - expr         ( expr )         number  number -&gt; number digit           digit  digit -&gt; 0   1   2   ...   9</pre>
---	--	---	---

<p><b>Bottom-Up / Shift-Reduce Parser</b></p> <ul style="list-style-type: none"> <li>• <b>Shift</b> tokens onto a stack</li> <li>• <b>Reduce</b> the stack to a non-terminal.</li> <li>• <b>LR</b> – <b>L</b>eft to right, <b>R</b>ightmost derivation</li> <li>• <b>LALR</b> – <b>L</b>ook-<b>A</b>head <b>LR</b> parsers are the most popular type of LR parsers. <ul style="list-style-type: none"> <li>◦ <b>Examples:</b> YACC/Bison</li> </ul> </li> <li>• <b>Fading from popularity</b></li> </ul>	<p><b>Top-Down Parser</b></p> <ul style="list-style-type: none"> <li>• <b>Non-terminals are expanded to match tokens.</b></li> <li>• <b>LL</b> – <b>L</b>eft to right, <b>L</b>eftmost derivation</li> <li>• <b>LL(k) Parser</b> – Looks ahead up to <i>k</i> elements. <ul style="list-style-type: none"> <li><b>Examples:</b> Java CC, ANTLR</li> <li>◦ The higher the <i>k</i>, the more difficult language is to parse. <i>k</i> <b>can be arbitrary</b>.</li> <li>◦ <b>LL(1)</b> - Easy to parse using either LL or recursive descent parsers. <b>Many computer languages are designed to be LL(1).</b></li> </ul> </li> </ul>	<p><b>Parser Combinator</b></p> <p><b>Combine simpler parsers to make a more complex parser.</b></p> <p><b>Example:</b> Parsec</p>	<p><b>Useful Parsec Functions</b></p> <ul style="list-style-type: none"> <li>• <b>many</b> – Parses <b>zero or more</b> occurrences of the given parser.</li> <li>• <b>many1</b> – Parses <b>1 or more</b> occurrences of the given parser.</li> <li>• <b>noneOf</b> – Anything but the specified value</li> <li>• <b>spaces</b> – Whitespace characters</li> <li>• <b>char</b> – The specific specified character</li> <li>• <b>string</b> – The specific specified string.</li> <li>• <b>sepBy</b> – Separate tokens by some token.</li> </ul>
--	---	--	--

Example Parsec Code		
<pre>import Text.ParserCombinators.Parsec  num :: GenParser st String num = many1 digit  main = do     print \$ parse num "Hello" "42"</pre>	<pre>import Text.ParserCombinators.Parsec  num :: GenParser st Integer num = do     str &lt;- many1 digit     return \$ read str  main = do     print \$ parse num "World" "42"</pre>	<ul style="list-style-type: none"> <li>• <b>st</b> – "State." Always required for our purposes.</li> <li>• <b>String/Integer</b> – Parser return type</li> <li>• <b>many1</b> – Select one of more digits.</li> <li>• <b>digit</b> – 0, 1, 2, 3, ..., 9 (<b>terminal</b>)</li> <li>• <b>num</b> – Parser entry function</li> <li>• <b>"Hello"/"World"</b> – Debug string.</li> <li>• <b>"42"</b> – String to parse.</li> </ul>

<p><b>Example with try, &lt; &gt;, and &lt;?&gt;</b></p> <pre>eol = try (string "\n")       &lt; &gt; string "\n\r"       &lt;?&gt; "end of line"</pre> <ul style="list-style-type: none"> <li>• <b>try</b> – If an incomplete match is found, rewind.</li> <li>• <b>&lt; &gt;</b> – "Or" Operator for matching tokens.</li> <li>• <b>&lt;?&gt;</b> – Otherwise with an accompanying error message.</li> </ul>		
--	--	--

## Practice Midterm and Review Notes

Question #1	Question #2	Question #3	Question #4	Question #5
a. <b>True</b> b. <b>False</b> – Lazy evaluation c. <b>False</b> – Lazy evaluation d. <b>False</b> – Statically type e. <b>True</b>	a. <b>True</b> b. <b>False</b> – Applicative functor c. <b>True</b> d. <b>True</b> e. <b>True</b>	a. <b>False</b> – Big step b. <b>True</b> c. <b>False</b> – Use store d. <b>True</b> e. <b>False</b>	a. <b>False</b> – Imperative b. <b>True</b> c. <b>False</b> d. <b>True</b> e. <b>True</b>	a. <b>True</b> b. <b>False</b> – Typeclass c. <b>True</b> d. <b>False</b> e. <b>False</b> – Algebraic data type

Haskell	Purely Functional	Functional Languages	Operational Semantics
<ul style="list-style-type: none"> <li>• <b>Purely Functional</b></li> <li>• <b>Lazy evaluation</b></li> <li>• <b>Fully Curried Language</b></li> <li>• <b>Statically Typed</b></li> <li>• <b>Type Inference</b> – Via context, Haskell can deduce the type.</li> </ul>	<ul style="list-style-type: none"> <li>• <b>Referential Transparency</b> – A <b>function call</b> can be replaced with its equivalent value without affecting the program</li> <li>• <b>No (re)assignment</b></li> <li>• <b>No loop</b></li> <li>• <b>No side effects</b></li> </ul>	<ul style="list-style-type: none"> <li>• <b>Functions are first class objects</b> meaning they can be passed to a function, returned from it, or created on the fly.</li> <li>• <b>Higher order function support</b></li> </ul>	<ul style="list-style-type: none"> <li>• <b>Small Step</b> – Structural Semantics</li> <li>• <b>Big Step</b> – Natural Semantics</li> <li>• <b>“Get stuck”</b> – When a function is encountered that does not have an associated rule.</li> </ul>

## CSV Parser Example

### Verbose Approach

```
import Text.ParserCombinator.Parsec
import System.Environment

csvFile :: GenParser Char st [[String]]
csvFile = do
    arr <- many line
    char eof
    return arr

line :: GenParser Char st [String]
line = do
    result <- many1 cell
    char '\n'
    return result

cells :: GenParser Char st [String]
cells = do
    firstCell <- cellContents
    nextCells <- remainingCells
    return (firstCell:nextCells)

cellContent :: GenParser Char st String
cellContent = many $ noneOf "\",\n" -- Two characters

remainingCells :: GenParser Char st [String]
remainingCells = do
    (char "," >> cells)
    <|> return []

main = do
    args <- getArgs
    p <- parseFromFile csvFile (head args)
    case p of
        Left msg -> error msg
        Right csv -> print csv
```

### Concise Approach

```
import Text.ParserCombinator.Parsec
import System.Environment

csvFile = line `sepBy` eof
line = cell `sepBy` string ","
cell = many (noneOf "\n")

eol = try (string "\n\r") -- Try more complex case first
    <|> string "\n"
    <?> "end of line"

main = do
    args <- getArgs
    p <- parseFromFile csvFile (head args)
    case p of
        Left msg -> error msg
        Right csv -> print csv
```

## 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 &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><b>List comprehension is syntactic sugar for using lists as monads.</b></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>return</b> – <code>Monad</code> equivalent of “pure” for <code>Applicative 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>
	<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 <code>bind (&gt;&gt;=)</code>.</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>	

## Functor Definitions

<p><b>Lists</b></p> <pre>instance Functor [] where   fmap = map</pre>	<p><b>Maybe</b></p> <pre>instance Functor Maybe where   fmap _ Nothing = Nothing   fmap f (Just x) = Just (f x)</pre>	<p><b>IO</b></p> <pre>instance Functor IO where   fmap f a = do     x &lt;- a     return (f x)</pre>
---	---	--

## Applicative Functor Definitions

<p><b>Lists</b></p> <pre>instance Applicative [] where   pure x = [x]   (&lt;*&gt;) fs xs = [ f x   f &lt;- fs, x &lt;- xs]</pre>	<p><b>Maybe</b></p> <pre>instance Applicative Maybe where   pure x = Just x   (&lt;*&gt;) Nothing _ = Nothing   (&lt;*&gt;) (Just f) x = fmap f x</pre>	<p><b>IO</b></p> <pre>instance Applicative IO where   a &lt;*&gt; b = do     f &lt;- a     x &lt;- b     return (f x)</pre>
---	---	---

## Monad Definitions

<p><b>Lists</b></p> <pre>instance Monad [] where   return x = [x]   (&gt;&gt;=) xs f = concat \$ map f x   fail _ = []</pre>	<p><b>Maybe</b></p> <pre>instance Monad Maybe where   return x = Just x   (&gt;&gt;=) Nothing _ = Nothing   (&gt;&gt;=) (Just x) f = f x   fail _ = Nothing</pre>	<p><b>IO</b></p> <pre>instance Monad IO where   (&gt;&gt;=) a f = do     x &lt;- a     f x   fail s = ioerror (userError s)</pre>
--	---	---

# Lecture #12 – Introduction to JavaScript

<b>JavaScript</b> <ul style="list-style-type: none"> <li>Developed at Netscape by Brendan Eichs in 10 days</li> <li>Originally named “Mocha”</li> <li>Syntax similar to Java</li> </ul>	<b>Multi-paradigm JavaScript</b> Supported programming paradigms: <ul style="list-style-type: none"> <li>Imperative</li> <li>Functional</li> <li>Object-Oriented (through <b>prototypes</b>)</li> </ul>	<b>Where JavaScript is Run</b> <ul style="list-style-type: none"> <li><b>Client Side Versions</b> <ul style="list-style-type: none"> <li>Runs on user machine</li> </ul> </li> <li><b>Server-side Versions</b> <ul style="list-style-type: none"> <li>JVM: <b>Rhino</b> &amp; <b>Nashorn</b></li> <li><b>Node.js</b></li> </ul> </li> </ul>	<b>Example: Imperative JavaScript</b> <pre>function addList(list){     var = i, sum = 0;     for( i = 0; i &lt; list.length ; i++){         sum += list[i];     }     return sum; }</pre>
---	--	---	---

<b>Example: Functional JavaScript</b> <pre>function addList(list){     if(list.length == 0){         return 0;     }      return list[0]         + addList(list.slice(1)); }</pre> <p><b>slice(begin[, end])</b> – Takes a subset of an array from the “begin” index to the “end” (exclusive). If no “end” is specified, it takes all elements to the end of the list.</p>	<b>Example: Object-Oriented JavaScript</b> <pre>function Adder(amount){     this.amount = amount; } Adder.prototype.add = function(x){     return this.amount + x; } var myAdder = new Adder(1) var y = myAdder.add(7)</pre> <p><b>Adder</b> – Name of a new constructor. <b>Convention is to start constructors with a capital letter.</b></p> <p><b>this</b> – Not optional in JavaScript.</p>	<b>Example: Quirks of JavaScript</b> <pre>var x = 42; // Create with var y = 7; // No error without var function add(a, b){     return a + b; } function noReturnAdd(a, b){     a + b; }  // c is “undefined” since no return var c = noReturnAdd(x, y)  //Lambda Function var myLambda = function(x){return x * x;}</pre>
--	--	--

<b>Printing to the Console in JavaScript</b> <ul style="list-style-type: none"> <li><b>Standard Approach:</b> <pre>console.log(“...”)</pre> <ul style="list-style-type: none"> <li>Not supported by all implementations.</li> </ul> </li> <li>JVM-based JavaScript Approach:           <pre>print</pre> </li> <li>Solution to Support a Single Interface:           <pre>var print = console.log</pre> </li> </ul>	<b>Closures</b> <ul style="list-style-type: none"> <li>Functions whose inner variables refer to independent (free) variables.</li> </ul> <p><b>Example: Closure in JavaScript</b></p> <pre>var getNextInt = function () {     var nextInt = 0;     return function () {         return nextInt++;     } }(); // Double paren to run the func console.log(getNextInt()); // print “0” console.log(getNextInt()); // print “1” console.log(getNextInt()); // print “2”</pre>	<b>Node.js</b> <ul style="list-style-type: none"> <li>JavaScript runtime environment and library <b>designed to run outside the browser.</b></li> <li>Based off Google’s V8 engine.</li> <li><b>npm</b> – Package manager to get new packages.</li> </ul>
		<b>Callback Function</b> <ul style="list-style-type: none"> <li>Functions in JavaScript are first class objects of type “Object”.</li> <li>Not executed immediately.</li> </ul> <p><b>JavaScript supports both “null” and “undefined”</b></p>

<b>Reading from a File with Callbacks in Node.js</b> <pre>var fs = require('fs') fs.readFile('myFile.txt',     function(err, data){         if(err)             throw err;         else             console.log(“” + data);     } ) console.log(“All done”)</pre> <p>“All done” prints before the file contents due to callbacks. <b>require</b> – Includes the JavaScript package “fs”</p>	<b>Synchronous File IO in Node</b> <pre>var fs = require('fs') var data =     fs.readFileSync('myFile.txt'); console.log(“All done”)</pre> <p>To eliminate callbacks, most function names can be appended with “Sync”</p>	<b>Creating a JavaScript Object</b> <pre>var myDog = {age : 3,     weight: 100}</pre> <p><b>Every object is a map.</b></p> <p><b>Adding a Field to a JavaScript Object</b></p> <pre>myDog['height'] = 45 // Add a new height field // Note the single quotes</pre> <p><b>Adding a Function to a JavaScript Object’s Prototype</b></p> <pre>myDog.speak = function(){ console.log(“Grr”); }</pre> <p><b>Delete a Function from a JavaScript Object’s Prototype</b></p> <pre>delete myDog.speak</pre>
	<b>Undeclared Object Fields</b> <p><b>Any undeclared object fields or uninstantiated variables are undefined.</b></p> <pre>var y; // Uninstantiated // Both print ‘undefined’ console.log(y) console.log(myDog.name)</pre>	

## Prototypes

<b>Object Prototypes</b> <p><b>JavaScript prototypes are just like any other object.</b></p> <pre>var dogPrototype = {     speak: function(){         console.log(“bark!”);     } }</pre>	<b>Defining an Object’s Prototype</b> <pre>var rex = { name: “Rex”,     __proto__: dogPrototype}</pre>	<b>Prototypical Inheritance:</b> If an object does not have a method or field, JavaScript looks to the object’s <b>__proto__</b> object.
---	--	--



<p><b>Add a Special "speak" Method to Rex</b></p> <pre> rex.speak = function() {     console.log("Grr"); }; rex.speak(); // Prints "Grr"  delete rex.speak; // Prints "Bark!" from __proto__ rex.speak();  delete rex.speak; // Does nothing rex.speak(); // Prints "Bark!" </pre>	<p><b>Effect of the "new" Keyword</b></p> <pre> function Cat(name, breed) {     var this = {}; // Add when new is used     this.prototype = Cat.prototype; // Also comes from new     this.name = name;     this.breed = breed;     this.speak = function() { console.log("meow"); };     return this; // Also comes from new } </pre>
<p><b>Unspecified Function Arguments:</b> In JavaScript, any unspecified function argument <b>defaults to "undefined"</b>.</p>	<p><b>No "return" in a Function</b></p> <pre> function noReturnAdd(x, y) {     x + y; // without "return" }  // c is "undefined" since no return var c = noReturnAdd(x, y) console.log(c); // Prints "undefined" </pre>

<p><b>Top Prototypes</b></p> <p><b>Object.prototype</b> – Top of all object prototypes</p> <p><b>Function.prototype</b> – Top of all function prototypes.</p>	<p><b>Iterating Using "forEach"</b></p> <pre> var arr = [1, 2, 3]; // Print each element in array arr.forEach(function(val) {     console.log(val); }); </pre> <p><b>Note:</b> This uses parentheses not curly brackets.</p>	<p><b>require</b></p> <ul style="list-style-type: none"> <li>Used to <b>import an external module</b> in Node.js</li> <li>Can be stored in a variable. <b>Example:</b></li> </ul> <pre>var net = require('net');</pre>	<p><b>Running from the Command Line</b></p> <ul style="list-style-type: none"> <li>Use the keyword <b>"node"</b> for Node.js.</li> </ul> <p><b>Example:</b></p> <pre>\$ node my_program.js</pre>
---	--	--	--

<p><b>Example: Create an Object with a Factory Method</b></p> <pre> var Droid = {     speak: function() {         console.log("I am "             + this.name);     },     create: function(name) {         var clone = Object.create(this);         clone.name = name;         return clone;     } }; </pre>	<p><b>Example: Currying in JavaScript</b></p> <pre> Function.prototype.curry = function() {     // Take slice from the Array class' prototype     var slice = Array.prototype.slice;     // Convert arguments to an array     var args = slice.apply(arguments);     var that = this;     return function() {         return that.apply(null,             arg.concat(slice.apply(arguments)));     }; };  function add(x, y) {     return x + y; }  var addOne = add.curry(1); console.log(addOne(3)); // Prints "4" </pre>
---	---

## Lecture #13 – Lambda Calculus

Expressions	Function Application	
$e ::= x \quad (\text{Variables, immutable})$ $  \lambda x. e \quad (\text{Lambda abstraction})$ $  e e \quad (\text{Function application})$ <p><b>Note:</b> Lambda (<math>\lambda</math>) is simply a function.</p> $v ::= (\lambda x. e) \quad (\text{Lambda abstraction})$	<p>Given a function where <math>\lambda</math> is a <b>complex expression</b>:</p> $\lambda(x. E)v$ <p>Then:</p> $\lambda(x. E)v \rightarrow E[x \mapsto v]$ <p>Hence, “<math>v</math>” replaces “<math>x</math>” in “<math>E</math>”.</p>	<p><b>Lambda Calculus is a simple, Turing complete language.</b> Hence it is equal in power to a Turing Machine.</p> <p>Lambda calculus stops evaluating when the result is in <b>normal form</b>.</p>

### Small-Step Evaluation Order Rules for Lambda Calculus

<b>Rule: SS-E1</b> $\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$	<b>Rule: SS-E2</b> $\frac{e_2 \rightarrow e'_2}{(\lambda x. e) e_2 \rightarrow (\lambda x. e) e'_2}$	<b>Rule: SS-Lambda Context</b> $(\lambda x. e) v \rightarrow e[x \mapsto v]$	<b>Optional Rule: Lazy SS-Lambda Context</b> $(\lambda x. e) e_2 \rightarrow e[x \mapsto e_2]$
---	---	---	---

### Evaluation Strategies

Strict Evaluation Strategies		Lazy Evaluation Strategies	
<b>Call by Value:</b> Pass a <b>copy</b> of a parameter	<b>Call by Reference:</b> <b>Implicit reference</b> (e.g., pointer) to the parameter is passed.	<b>Call By Name:</b> <b>Re-evaluate the argument each time</b> it is used.	<b>Call by Need:</b> <b>Memoizes parameter value</b> after first use.

Language Equivalents of $(\lambda x. e)$		True and False in Lambda Calculus	
<b>JavaScript:</b> <code>function(x){return e;}</code>	<b>Haskell:</b> <code>(\x -&gt; e)</code>	<b>True in Lambda Calculus:</b> $getFirstParam = tru = (\lambda x. \lambda y. x)$ <p><b>Note:</b> This returns the <b>first</b> parameter in the pair of values.</p>	<b>True in Lambda Calculus:</b> $getSecondParam = fls = (\lambda x. \lambda y. y)$ <p><b>Note:</b> This returns the <b>second</b> parameter in the pair of values.</p>

Conditional in Lambda Calculus	
$test = \lambda cond. \lambda then. \lambda els. (cond \text{ then } els)$	
<b>Example #1:</b> $test(tru \text{ tru } fls)$ $\lambda cond. \lambda then. \lambda els. (cond \text{ then } els)(tru \text{ tru } fls)$ $\lambda then. \lambda els. (tru \text{ then } els)(tru \text{ fls})$ $\lambda els. (tru \text{ tru } els)(fls)$ $(tru \text{ tru } fls)$ $(\lambda x. \lambda y. x)(tru \text{ fls})$ $(\lambda y. tru)(fls)$ $tru$	<b>Example #2:</b> $test(flz \text{ tru } fls)$ $\lambda cond. \lambda then. \lambda els. (cond \text{ then } els)(flz \text{ tru } fls)$ $\lambda then. \lambda els. (flz \text{ then } els)(tru \text{ fls})$ $\lambda els. (flz \text{ tru } els)(fls)$ $(flz \text{ tru } fls)$ $\lambda x. \lambda y. y (tru \text{ fls})$ $\lambda y. y (fls)$ $flz$

<b>Boolean And</b> $andd = \lambda b. \lambda c. (b \text{ c } fls)$	<b>Pair</b> $pair = \lambda f. \lambda s. \lambda b. (b \text{ f } s)$ <p><b>Pair</b> – A tuple-like data structure in Lambda Calculus.</p>
---	--

### Working with a Pair in Lambda Calculus

<b>First Element in a Pair</b> $first = \lambda p. (p \text{ tru})$	<b>Second Element in a Pair</b> $second = \lambda p. (p \text{ fls})$
<p><b>Note #1:</b> In the case of both <i>first</i> and <i>second</i>, the term <i>p</i> must be a pair.</p> <p><b>Note #2:</b> Both of these rely on the <i>tru</i> or <i>fls</i> being substituted for the “<i>b</i>” in the <i>pair</i> data structure in term selecting either the first or second element.</p>	

<b>Church Encoding Numerals</b>  $zero = \lambda s. \lambda z. z$ $one = \lambda s. \lambda z. s\ z$ $two = \lambda s. \lambda z. s\ s\ z$ $three = \lambda s. \lambda z. s\ s\ s\ z$	<b>Successor Function</b> $scc = \lambda n. \lambda s. \lambda z. s(n\ z)$  <b>Example:</b> $one' = scc\ zero$ $two' = scc\ one' = scc(scc\ zero)$	<b>Plus in Lambda Calculus</b> $plus = \lambda m. \lambda n. (\lambda s. \lambda z. m\ s\ (n\ s\ z))$  <b>Example:</b> Use church encoding numerals $plus\ three\ two \rightarrow^* \lambda s. \lambda z. (three\ s\ (two\ s\ z))$
<b>Omega – Infinite Loop</b>  $\Omega = (\lambda x. x\ x)\ (\lambda x. x\ x)$	<b>Fix Combinator</b>  $fix = \lambda f. (\lambda x. f(\lambda y. x\ x\ y))\ (\lambda x. f(\lambda y. x\ x\ y))$  <b>Note:</b> Can be used to do factorial operations.  <b>Usage:</b> $fix(g\ x) = g((fix\ g)x)$	

## Lecture #14 – JavaScript Scoping

<p><b>Example: First Class Function</b></p> <pre>function makeAdder(x) {   return function(y) {     return x + y;   }; } var addOne = makeAdder(1); // Prints "11" console.log(addOne(10));</pre>	<p><b>Example: Function Application</b></p> <pre>function makeAdderList(arr) {   var i;   var output = [];   for(i = 0; i &lt; output.length; i++){     // Need to create a new scope     function() {       // Can add to arr without append       output[i] = function(y) {         return arr[i] + y;       };     }   } }</pre> <p>JavaScript lacks block scope for the closure to be right, must create the function inside another function.</p>	<p><b>Block Scope</b> – The scope (i.e. visibility) of a variable is limited to a specific block (e.g., for loop, if statement, etc.).</p> <ul style="list-style-type: none"> <li>Unlike most languages, JavaScript does not have block scope.</li> <li>To create a new scope, use an anonymous function.</li> </ul> <p><b>Variable Hoisting</b> – All variable declarations (i.e., use of “var”) are treated as if they are at the beginning of the function.</p>	<p><b>“this” in JavaScript</b></p> <p><b>this</b> – Refers to the scope where the function is called.</p> <ul style="list-style-type: none"> <li><b>In Normal Function Calls</b> – <b>this</b> refers to the global “this”</li> <li><b>Object Methods</b> – The object itself.</li> <li><b>Constructor (using “new”)</b> – The newly created object.</li> <li><b>Exceptions: apply, call, and bind.</b> Inline event handles on DOM elements</li> </ul> <p><b>Any time a new function is created, the other “this” is no longer in scope</b></p>
---	--	--	--

<p><b>Execution Context</b></p> <p>Consists of three part:</p> <ul style="list-style-type: none"> <li><b>A Variable Object</b> – Container for variables and functions.</li> <li><b>Scope Chain</b> – Variable object plus parent scopes</li> <li><b>Context Object</b> – <b>this</b></li> </ul>	<p><b>Global Context</b></p> <ul style="list-style-type: none"> <li><b>Top Level Context</b></li> <li>Variable object is known as the “global object”</li> <li><b>this</b> – Refers to the global object.</li> </ul> <p><b>Any variable declared without var is added to the global context.</b></p>	<p><b>Function Contexts</b></p> <ul style="list-style-type: none"> <li><b>Activation</b> or <b>Variable Objects</b> which include:           <ul style="list-style-type: none"> <li>Arguments passed to the function</li> <li>A special arguments object</li> <li>Local variables</li> </ul> </li> </ul>
--	--	--

<p><b>apply, bind, call Example</b></p> <pre>x = 3; function foo(y) {   console.log(this.x + y); } foo(100); // Prints "103"  // Array passed for args foo.apply(null, [100]); // Update the context foo.apply({x:4}, [100]);  // No array needed foo.call({x:4}, 100);  // Create a new function var bf = foo.bind({x:5}); bf(100);</pre>	<ul style="list-style-type: none"> <li><b>apply</b> – Calls a function with the arguments passed as an array.</li> <li><b>call</b> – Calls a function with the arguments passed in comma separated.</li> <li><b>bind</b> – Used to create a new function with a custom context.</li> </ul>	
--	--	--

## Lecture #14.5 – JSLint and TypeScript

<b>Issues in JavaScript</b> <ul style="list-style-type: none"> <li>• No block scope</li> <li>• Forgetting <b>var</b> can lead to unexpected behavior since variables become global.</li> <li>• Operator <b>"=="</b> is not transitive.</li> <li>• Switch/case statements require <b>"break"</b></li> </ul>	<b>JavaScript Automatically Inserts Semicolons</b> <pre>function makeObject() {   return // Semicolon inserted here   {     madeBy: 'Austin Tech. Sys.'   } } var o = makeObject(); console.log(o.madeBy); // error</pre>	<b>Function <b>"parseInt"</b> can Yield Unexpected Results</b> <pre>// Drops the " tons" console.log("what do you get? "   + parseInt("16 tons"));  // Prints just "1" due to the "Oh" parseInt("101");</pre>
--	---	---

<b>Behavior of <b>"typeof"</b></b> <p><b>typeof</b> – Returns a string. May yield unexpected results.</p> <pre>typeof 5 // "number" typeof "hi" // "string"  typeof NaN // "number"  typeof null // "object"</pre>	<b>Behavior of <b>"typeofChar"</b></b> <p><b>typeofChar</b> – Returns a string. Classifies letters as <b>"digits"</b>.</p> <pre>typeofChar "5" // "digit"  typeofChar "q" // "digit"  // "Other character" typeofChar " "</pre>	<b>JSLint</b> <ul style="list-style-type: none"> <li>• A tool to write cleaner and safer JavaScript.</li> <li>• Requires that <b>"use strict"</b>; (with quotes and followed by semicolon) be added at the beginning of all functions.</li> <li>• Performs static code analysis.</li> <li>• Helps catch common programming errors by requiring:             <ul style="list-style-type: none"> <li>◦ Variables declared before they are used.</li> <li>◦ Semicolons are always used.</li> <li>◦ Double equals never used.</li> </ul> </li> <li>• Inspired by the <b>"lint"</b> tool from C</li> </ul>
--	---	---

<b>Benefits of Type Systems</b> <ul style="list-style-type: none"> <li>• Tips for compilers</li> <li>• Hints for IDEs</li> <li>• Enforced documentation</li> <li>• Prevent code with errors from running.</li> </ul>	<b>TypeScript</b> <ul style="list-style-type: none"> <li>• Developed by Microsoft</li> <li>• <b>Static type checker</b> for JavaScript.</li> <li>• A new "superset" language of JavaScript with:             <ul style="list-style-type: none"> <li>◦ Type annotations</li> <li>◦ Classes</li> </ul> </li> <li>• <b>Compiles to JavaScript</b></li> </ul>	<b>Function Type Annotations in TypeScript</b> <pre>function greet(person: string) {   console.log("Hello "     + person); } var user : string = "Vlad";  // Prints "Hello Vlad" greet(user);</pre>	<b>Types in TypeScript</b> <ul style="list-style-type: none"> <li>• <b>number</b> (<b>var</b> pi : <b>number</b> = 3.14)</li> <li>• <b>boolean</b> (<b>var</b> b : <b>boolean</b> = true)</li> <li>• <b>string</b> (<b>var</b> greet : <b>string</b> = "hi")</li> <li>• <b>array</b> (<b>var</b> lst : <b>number</b>[] = [1, 2])</li> <li>• <b>enum</b></li> <li>• <b>any</b> (<b>var</b> a : <b>any</b> = 3; <b>var</b> b : <b>any</b> = "hi")</li> <li>• <b>void</b></li> </ul>
--	---	---	---

<b>TypeScript Class</b> <pre>class Employee{   name : string;   salary : number;    constructor(name : string, salary : number){     this.name = name;     this.salary = salary;   }   display(){ console.log(this.name); } }  var emp = new Employee('Jon', 50000); emp.display();</pre>	<b>TypeScript Function Example</b> <pre>function swap(arr : number[], i : number,   j : number) {   var tmp : number;   tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp; } function sortAndGetLargest (arr : number[]) {   var tmp : number;   var i : number;   var j : number;   tmp = arr[0]; // largest elem   for (i=0; i&lt;arr.length; i++) {     if (arr[i] &gt; tmp) tmp = arr[i];     for (j=i+1; j&lt;arr.length; j++)       if (arr[i] &lt; arr[j]) swap(arr,i,j);   }   return tmp; } var largest = sortAndGetLargest([99,2,43,8,0,21]); console.log(largest); // Prints 99</pre>	<b>JSLint Requirements</b> <ul style="list-style-type: none"> <li>• <b>++</b> - This should be replaced with <b>"+= 1"</b></li> <li>• <b>var</b> – Each var declaration should be on its own line and must be at the top of the function/file.</li> <li>• <b>Single line if and for statements</b> still need curly brackets.</li> </ul>
---	--	--

# Lecture #15 – Event-Based Programming and Cryptocurrencies

<p><b>JavaScript Embedded in HTML</b></p> <p>Create a button on a website that prints "Hello" when clicked. <b>Inline event handlers are considered bad practice.</b></p> <pre>&lt;html&gt; &lt;input   type='button'   onclick='alert("Hello");'   value='Say hi' /&gt; &lt;/html&gt;</pre>	<p><b>Improved JavaScript in HTML</b></p> <p>Give buttons an "id" and update its "onclick" method</p> <pre>&lt;html&gt; &lt;input id='thebutton'   type='button'   value='Say hi' /&gt;  &lt;script type="text/javascript"&gt;   var btn = document.     getElementById('thebutton');   btn.onclick = function() {     alert('Groovy');   }; &lt;/script&gt; &lt;/html&gt;</pre>	<p><b>Adding an Event Listener</b></p> <ul style="list-style-type: none"> <li>If clicking a button should perform multiple functions, then an event listener should be used.</li> </ul> <pre>function sayGroovy(){   console.log("Groovy"); }  // Add an "onclick" event listener btn.addEventListener('click',   sayGroovy);  // Add another event listener btn.addEventListener('click',   function(){     console.log("Bogus");   });</pre>
--	--	--

<h3>Removing an Event Listener</h3> <ul style="list-style-type: none"><li>Event listeners can be removed by function name. Example: <pre>btn.removeEventListener('click', sayGroovy);</pre></li></ul>	<h3>Event Emitter</h3> <ul style="list-style-type: none"><li>Import the “events” module using the syntax <pre>var ee = require('events').EventEmitter;</pre></li><li>Used to create event via the keyword “on”. Example: <pre>ee.on('die', function() { console.log("Died"); });</pre></li><li>Invoking (emitting) an event using the keyword “emit” Example: <pre>setTimeout( function() { ee.emit('die'); }, 100); // in ms</pre></li></ul>	<h3>Create a TCP Server in Node.js Using Event Listeners</h3> <pre>var net = require('net'); var eol = require('os').EOL; var srvr = net.createServer();  // Add an event listener srvr.on('connection', function(client) {   client.write('Hello there!' + eol);   client.end(); }); srvr.listen(9000);</pre> <p>telnet – Used to connect to a TCP server on the command line.</p> <p>127.0.0.1 – IP address of localhost</p>
<h3>Events in JavaScript</h3> <ul style="list-style-type: none"><li>JavaScript is single threaded.</li></ul> <p>An event must be run to completion before the next event handler can run.</p>		

## Cryptocurrencies

<p><b>Types of Keys</b></p> <ul style="list-style-type: none"> <li><b>Private Key:</b> Known only by the owner</li> <li><b>Public Key:</b> Known by everyone</li> </ul>	<p><b>Digital Signature</b></p> <ul style="list-style-type: none"> <li><b>Non-Repudiation – Involves associating actions or changes to a unique individual.</b> <ul style="list-style-type: none"> <li><b>Solution in Cryptocurrency:</b> Digital signature.</li> </ul> </li> <li><b>Procedure:</b> <ul style="list-style-type: none"> <li><b>Step #1:</b> Owner encrypts the message with his private key</li> <li><b>Step #2:</b> Use the public key to decrypt the message.</li> </ul> </li> <li><b>Analogy:</b> Enclosed Bulletin Board</li> </ul>	<p><b>Private Key Encryption</b></p> <p>Used to transmit sensitive data to a specific recipient.</p> <ul style="list-style-type: none"> <li><b>Procedure:</b> <ul style="list-style-type: none"> <li><b>Step #1:</b> A user encrypts his data using the recipient's public key.</li> <li><b>Step #2:</b> The intended recipient decrypts the data using his private key.</li> </ul> </li> <li><b>Analogy:</b> A public mailbox. Anyone can put letters in, but only the mailman has the key to open the box.</li> </ul>	<ul style="list-style-type: none"> <li><b>update</b> – Used to update the signature with the specified message contents. Each signature object can only be updated once.</li> <li><b>hex</b> – Specifies that the output should be in hexadecimal format.</li> <li><b>Sync</b> – Ensures that the file read is done immediately without relying on a callback.</li> <li><b>SHA</b> – "Secure Hash Algorithm"</li> <li><b>RSA</b> – Signature algorithm</li> </ul>
---	--	---	---

<p><b>Example: JavaScript Signer Example</b></p> <pre>var crypto = require('crypto'); var fs = require('fs');  // Constructor for a "Signer" object function Signer(privKeyFile){   this.privKey = fs.readFileSync(privKeyFile).toString('ascii'); }  // Add a "signMessage" function to the Signer prototype Signer.prototype.signMessage = function(msgFileName){   var msg = fs.readFileSync(msgFileName).toString('ascii');   var sign = crypto.createSign('RSA-SHA256');    return sign.update(msg).sign(this.privKey, 'hex'); }</pre>	<p><b>Double Spending – Spend the same funds in multiple places.</b></p> <p><b>Solutions to Prevent Double Spending:</b></p> <ul style="list-style-type: none"> <li><b>Centralized Authority</b> – Disadvantages include that the central authority would charge a fee and not everyone trusts central authorities.</li> <li><b>Decentralized Authority</b> – Broadcast transactions to everyone.</li> </ul> <p><b>Ledger</b> – Used to keep a <b>history of all transactions</b> and the funds held by all users.</p>
---	--



#### Example: JavaScript Verifier Example

```
var crypto = require('crypto');
var fs = require('fs');

// Constructor for a "Verifier" object
function Verifier(publicKeyFile){
  this.publicKey = fs.readFileSync(privKeyFile).toString('ascii');
}

// Add a "verifySignature" function to the Verifier prototype
Verifier.prototype.verifySignature = function(msgFileName, signature){
  var msg = fs.readFileSync(msgFileName).toString('ascii');

  // Create a verifier
  var ver = crypto.createVerifier('RSA-SHA256').update(msg);

  // Verify signature matches the hash
  var legit = ver.verify(this.publicKey, signature, 'hex');
  return legit;
}
```

#### Bitcoin Mining

- **Block Chain** – Defines the transaction history.
  - **Used to prevent double spending.**
- **Proof of Work** – Verification of the block chain.
- Miners hash transaction details plus a “proof” (i.e. nonce)
  - **Reward:** New bitcoins are mined for the first to find a proof.
- **Cost to Derive a Proof:**  $2^N$  where N is the number of the initial bits that must be “0” for the proof to be valid.
- **Cost to Verify a Proof:** A single hash
- Bitcoin protocol is designed to make mining more profitable than cheating.

#### Attributes of a Good Hash Function

<b>Role of a Hash Function:</b> Compress arbitrary length inputs to small, fixed length outputs.	<b>One Way:</b> Given an output “y”, it is infeasible to find an “x” such that: $h(x) = y$	<b>Collision Resistant:</b> It is infeasible to find any “x” and “y” such that: $h(x) == h(y)$	<b>Compression</b>	<b>Efficient</b>
---	---	---	--------------------	------------------

## Lecture #16 – Typed Arith

Benefits of Type Systems	The Typed Arith Language	Good and Bad Typing Systems	Type Safety = Progress + Preservation
<ul style="list-style-type: none"> <li>• Tips for <b>compilers</b> to make code more <b>efficient</b>.</li> <li>• Tips for <b>IDEs and other tools</b> to make <b>writing code easier</b>.</li> <li>• <b>Enforced documentation</b>.</li> <li>• <b>Prevent code with errors from running</b>.</li> </ul>	<pre> e ::= true       false       0       succ e       pred e       iszero e       if e then e else e </pre>	<ul style="list-style-type: none"> <li>• <b>Good type systems</b> prevent “bad” programs from running.</li> <li>• <b>Bad type systems</b> prevent valid programs from running.</li> </ul>	<p><b>Progress</b></p> <p>A well typed expression does not “get stuck”</p> <p><b>Formal Definition:</b> Given that <math>e : T</math>, then either:</p> <ol style="list-style-type: none"> <li>1. <math>e</math> is a value</li> <li>2. There exists an <math>e'</math> such that <math>e \rightarrow e'</math></li> </ol>
<p><b>Typing Rules</b></p> <p><b>Format:</b></p> $e : T$ <p><b>Meaning:</b> Expression <math>e</math> falls into one of two categories:</p> <ol style="list-style-type: none"> <li>1. <math>e</math> <b>evaluates to a value of type <math>T</math></b></li> <li>2. <b>Goes into an infinite loop.</b></li> </ol>	<pre> v ::= true       false       nv n ::= 0       succ nv T ::= Bool       Int </pre>	<ul style="list-style-type: none"> <li>• <b>Typchecking</b> – In an expression “typechecks”, the expression is either: <ul style="list-style-type: none"> <li>○ A value</li> <li>○ An evaluation rule reduces the expression to a different expression.</li> </ul> </li> </ul>	<p><b>Preservation</b></p> <p>A well-type expression will not change its type during evaluation.</p> <p><b>Formal Definition:</b> Given that <math>e : T</math> and <math>e \rightarrow e'</math>, then:</p> $e' : T$

# Lecture #17 – Macros and Sweet.js

<p><b>Macros</b></p> <ul style="list-style-type: none"> <li>Short for “<i>macroinstruction</i>”</li> <li>Rule specifies how an <b>input sequence</b> maps to a <b>replacement sequence</b>.</li> </ul>	<p><b>Example: C Preprocessor Example</b></p> <pre>#define PI 3.14159 #define SWAP(a,b) {int tmp=a;a=b;b=tmp;}  int main(void){     int x = 4, y=5, diam = 7;     double circum = diam * PI;     SWAP(x,y) }</pre>	<p><b>Basic Compiler Structure with C-Style Macros</b></p> <pre> graph LR     SC[Source Code] --&gt; PP[Pre-Processor]     PP -- Expanded Code --&gt; LT[Lexer/Tokenizer]     LT -- Tokens --&gt; P[Parser]     P --&gt; AST[Abstract Syntax Tree]     AST --&gt; C[Compiler]     C -- Machine Code --&gt; MC[Machine Code]     AST --&gt; I[Interpreter]     I --&gt; IO[Interpreter]     </pre>
<p><b>Macros in C</b></p> <ul style="list-style-type: none"> <li>Performed by a <i>preprocessor</i></li> <li><b>Rely on text substitution.</b></li> <li>Embedded languages like PHP, Ruby, etc. use a similar approach.</li> </ul>	<p><b>C Preprocessor Output</b></p> <pre>int main(void){     int x = 4, y = 5, diam = 7;     double circum = diam * 3.14159;     {int tmp=x;x=y;y=tmp;} }</pre>	

<p><b>Problem with C Macros (Input)</b></p> <pre>// Macro should be on one line #define SWAP(a,b) {int tmp=a;                     a=b;                     b=tmp;}  int main(void){     int x = 4, tmp = 5;     SWAP(x,tmp) }</pre>	<p><b>Hygienic Macro</b> – Any macro whose expansion is <b>guaranteed not to cause the accidental capture of identifiers</b>.</p> <p><b>Syntactic Macros</b></p> <ul style="list-style-type: none"> <li>Derive from Lisp since Lisp programs are essentially one big AST.</li> <li><b>Work at the level of abstract syntax trees.</b></li> <li>Powerful by expensive.</li> <li><b>Hygiene easier to address at the AST level.</b></li> <li>Essentially a <b>source-to-source compiler</b>.</li> </ul>	<p><b>Macros in JavaScript</b></p> <ul style="list-style-type: none"> <li><b>No standard macro system for JavaScript</b></li> <li>Sweet.js has been gaining interest.</li> <li>Recently redesigned.</li> </ul>
<p><b>Problem with C Macros (Output)</b></p> <pre>int main(void){     int x = 4, tmp = 5;     { int tmp = x;       a = tmp;       tmp = tmp;     } }</pre> <p>Hence, a <b>variable name collision</b> between the two variables named “<i>tmp</i>”. This is known as “<b>inadvertent variable capture</b>”</p>		<p><b>Sweet.js</b></p> <ul style="list-style-type: none"> <li><b>Borrows concepts from Racket.</b></li> <li><b>Source-to-source compiler</b> (i.e., transpiler) for JavaScript.</li> <li>Examples of other JavaScript transpilers:             <ul style="list-style-type: none"> <li><b>TypeScript</b></li> <li><b>CoffeeScript</b></li> <li><b>Dart</b> (includes its own VM)</li> </ul> </li> <li>Project backed by Mozilla</li> </ul> <p><b>Invoking Sweet.js</b></p> <ul style="list-style-type: none"> <li>From command line: \$ <b>sjs</b> myfile.js -d out/</li> <li>Compiled files run normally (as shown below for Node): \$ <b>node</b> out/myfile.js</li> </ul>

<p><b>Writing a Swap Function in Sweet.js</b></p> <pre>syntax swap = function(ctx){     let innerCtx = ctx.next().value.inner();     let first = innerCtx.next().value;     // Eat the comma     innerCtx.next(); // No need for "value()"      // Get the second parameter     let second = innerCtx.next().value;     return #`var tmp = \${first};         \${first} = \${second};         \${second} = tmp;`; }  swap(a, b); // Invokes the macro</pre> <p><b>Note #1:</b> The returned string is preceded by a pound (#) sign and is enclosed in backticks (`).</p> <p><b>Note #2:</b> Sweet.js variables are declared with “<i>let</i>”.</p>	<p><b>Concatenating Multiple Result Strings</b></p> <p>This function squares a set of input variables.</p> <pre>syntax square = function(){     var innerCtx = ctx.next().value.inner();     // Start with empty results     result = #``;     while(let stx of innerCtx){         result =             result.concat(`\${stx}=\${stx}*\${stx};`);         // Eat comma         // Ignored if no comma present         innerCtx.next();     } }  square(a, b, c); // Invokes the macro</pre> <p><b>Note #1:</b> Use “<i>.concat</i>” to concatenate multiple result strings.</p> <p><b>Note #1:</b> If a token is not present, “<i>.next()</i>” does not cause an error.</p>	<p><b>Keywords in Sweet.js</b></p> <ul style="list-style-type: none"> <li><b>let</b> – Create a Sweet.js variable.</li> <li><b>ctx.next().value</b> – Get the next value from the context.</li> <li><b>#`...`</b> – Used to define a result string.</li> <li><b>concat</b> – Used to combine two result strings.</li> <li><b>let xxx of yyy</b> – Iterate over a list of tokens.</li> <li><b>isIdentifier</b> – Used to check if a Sweet.js variable matches some string.</li> </ul>
--	--	--

### Input JavaScript **class** Code to be Parsed by Sweet.js

```
class Droid{
  constructor(name, color){
    this.name = name;
    this.color = color;
  }

  rollWithIt(it){
    console.log(this.name + " is rolling "
      + "with " + it);
  }
}
```

### A **class** in Sweet.js

```
syntax class = function(ctx){
  let className = ctx.next().value;
  let bodyCtx = ctx.next().value.inner();

  // By default assume empty constructor
  let construct = #`function() { }`;
  let result = #``;

  while( let item of bodyCtx ){

    // Check if constructor
    if(item.isIdentifier('constructor')){
      // Get arguments then function code
      construct = #`function ${className}
        ${bodyCtx.next().value}
        ${bodyCtx.next().value}`;
    }
    else {
      // Add the function to the class prototype
      result = result.concat(
        #`${className}.prototype.${item} =
          function ${bodyCtx.next.value}
            ${bodyCtx.next.value}`);
    }
  }
  // Return the constructor and methods
  return construct.concat(result);
}
```

## Lecture #18 – Simply Typed Lambda Calculus

<p><b>The Typed Arith Language</b> (with Lambda Functions)</p> <pre> e ::= true       false       0       succ e       pred e       iszero e       if e then e else e       λ x. e       e e  v ::= true       false       i       λ x. e  T ::= Bool       Int       T → T </pre>	<p><b>Options for Determining Function Type</b></p> <ol style="list-style-type: none"> <li>1. <b>Type Inference</b></li> <li>2. <b>Require explicit type annotations</b></li> </ol>	<p><b>Omega Combinator</b></p> <p><b>Definition:</b>  <math>\Omega = (\lambda x. x x)(\lambda x. x x)</math></p>	
	<p><b>Type Annotation</b></p> <p><math>\lambda x:T. e</math></p> <p><b>Note:</b> T is the type of the argument “x”</p>	<p><b>Description:</b> Results in an infinite loop since the combinator keeps returning itself. Omega <b>is a valid a valid program supported by untyped lambda calculus.</b></p>	
	<p><b>Managing the Type of a Variable</b></p> <ul style="list-style-type: none"> <li>• Use a <b>typing environment</b>. <ul style="list-style-type: none"> <li>◦ <b>Maps variables to types</b></li> <li>◦ Referred to using the Greek letter: <math>\Gamma</math> (gamma)</li> </ul> </li> <li>• <b>Typing rules must be defined in terms of the typing environment.</b></li> </ul>	<p><b>Issue:</b> If this expression is type checked, it enters an infinite type checking loop which causes it to fail</p>	
	<p><b>Turnstile</b></p> <ul style="list-style-type: none"> <li>• <math>\vdash</math> - Used in revised typing relations to indicate that the typing rule is with respect to the specified typing environment (<math>\Gamma</math>)</li> </ul>	<p><b>@ Symbol in Haskell</b></p> <p>@ – Allows the programmer to represent an entire expression using a shortened notation.</p> <p><b>Example:</b>  <math>e @ (\text{ESucc } e1)</math></p>	

Arith Typing Rules Using a Typing Environment ( $\Gamma$ )			
[T-True]	$\Gamma \vdash \text{true} : \text{Bool}$	[T-If]	$\frac{\Gamma \vdash e_1 : \text{Bool}, \Gamma \vdash e_2 : T, \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } (e_1) \text{ then } (e_2) \text{ else } (e_3) : T}$
[T-False]	$\Gamma \vdash \text{false} : \text{Bool}$		
[T-succ]	$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{succ } e : \text{Int}}$	[T-FunctionApplication]	$\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2, \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$ <p><b>Note:</b> <math>e_1</math> is a function while <math>e_2</math> is a parameter.</p>
[T-pred]	$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{pred } e : \text{Int}}$	[T-LambdaVariable]	$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$
[T-iszero]	$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{iszero } e : \text{Bool}}$	[T-LambdaContext]	$\frac{\Gamma \vdash x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x. e : T_1 \rightarrow T_2}$

## Lecture #19 – Metaprogramming and JS Proxies

<b>Metaprogramming:</b> Writing programs that manipulate other programs. <ul style="list-style-type: none"> <li>Proposed in ECMAScript 6 for JavaScript.</li> </ul> <b>Terminology in Reflection</b> <ul style="list-style-type: none"> <li><b>Introspection:</b> Ability to examine (but not modify) the structure of a program.</li> <li><b>Self-modification:</b> Ability to modify the structure of a program.</li> </ul>	<b>Introspection</b> Ability to <b>examine</b> (but not modify) <b>the structure of a program</b> . <p><b>JavaScript Examples</b></p> <p><b>Property Lookup</b></p> <pre>"x" in o; //o is an object</pre> <p><b>Iterate Over All Properties of an Object</b></p> <pre>for( prop in o ){   // Do something   ... }</pre>	<b>Self-modification</b> Ability to <b>modify the structure of a program</b> . <p><b>JavaScript Examples</b></p> <pre>o["x"]; // Computed property o.y = 42; // Add new property delete o.y; // Delete property  // Reflected method call O["m"].apply(null, [38]);</pre>	<b>Proxies in JavaScript</b> <ul style="list-style-type: none"> <li><b>Metacircular Interpretation</b> – The language is able to understand its own language.</li> <li>Until recently, JavaScript did not support intercession. <ul style="list-style-type: none"> <li>JavaScript proxies are intended to fix that.</li> </ul> </li> <li><b>Node.js' implementation of proxies lags behind the standard.</b></li> <li><b>Proxies only exist for objects and functions.</b> <ul style="list-style-type: none"> <li><b>Proxies do not exist for primitives.</b></li> </ul> </li> </ul>
---	--	--	--

<b>Proxies and Common Lisp</b> <ul style="list-style-type: none"> <li>Lisp was developed before object oriented languages were popular.</li> <li>Many libraries were create with non-standard OO systems.</li> <li><b>Common Lisp Object System (CLOS)</b> – Standard object oriented system for Lisp.</li> </ul>	<b>Achieving Lisp Object Backwards Compatibility</b> <p><b>Option #1:</b> Rewrite all libraries using CLOS.</p> <p><b>Disadvantages:</b></p> <ul style="list-style-type: none"> <li>Huge number of libraries.</li> <li>Not feasible to rewrite them all.</li> </ul> <p><b>Option #2:</b> Make a complex API.</p> <p><b>Disadvantages:</b></p> <ul style="list-style-type: none"> <li>API difficult to understand.</li> <li>Systems had conflicting features.</li> </ul> <p><b>Option #3:</b> Keep API simple and modify object behavior to fit different systems.</p> <ul style="list-style-type: none"> <li>This approach relies on <b>metaobject protocols</b>.</li> </ul>	<b>Proxies and Handlers</b> <ul style="list-style-type: none"> <li>The behavior of a proxy is determined by <b>traps</b> specified in its <b>handler</b> (i.e., the <b>metaobject</b>).</li> <li><b>Trap</b> – <b>Methods that intercept an operation.</b></li> <li><b>Handler</b> – The metaobject that <b>specifies the details of the trap</b>. The handler itself is <b>usually a normal object</b>.</li> <li>Using proxies in node requires a special flag: <b>"--harmony-proxies"</b>. <b>Example:</b></li> </ul> <pre>\$ node --harmony-proxies prog.js</pre>	<b>Kinds of JavaScript Proxies</b> <ul style="list-style-type: none"> <li><b>Object Proxies</b> – Defined with: <pre>Proxy.create(handler, proto)</pre> </li> <li><b>Functions</b> (with extra traps) - Defined with: <pre>Proxy.createFunction(handler, callTrap, constructTrap)</pre> </li> <li><b>Proxies do not exist for primitives.</b></li> </ul>
---	--	--	--

<p><b>A Simple Proxy</b></p> <pre>var MyHandler = {   get:function(myProxy, name){     console.log(name       + " accessed");     return 1;   } }  var p = Proxy.create(MyHandler); // Prints "hello accessed." var q = p.hello;  // Prints "1" console.log(q);  // Error since no "set" handler p.name = "Me";</pre>	<p><b>A No-op Proxy – All Operations Passed through Unchanged</b></p> <pre>function handlerMaker(obj){   // Delete a property from an object   delete : function(name){ return obj[name]; },    // Check if object has the specified property   has : function(name){ return name in obj; },    // Check if object (not prototype chain) has property   hasOwn : function(name){return Object.property     .hasOwn(obj, name); },    // Get a property value   get : function(name){ return obj[name]; },    // Set a property value   set : function(rcvr, name, val){ obj[name] = val; },    // Get all properties of an object   enumerate : function(){     var props = [];     var prop;     for(prop in obj){ props.push(prop); }     return props;   },    // Get all of the keys of an object   keys: function(){ return Object.keys(obj); } }</pre>	<p><b>Aspect Oriented Programming</b></p> <ul style="list-style-type: none"> <li><b>Some code not well organized into objects.</b> Example: <ul style="list-style-type: none"> <li><b>Cross-cutting concern</b> where code is spread throughout a program.</li> </ul> </li> <li><b>Canonical Example:</b> Logging Statements <ul style="list-style-type: none"> <li><b>Littered throughout the code</b></li> <li><b>Swapping out a logger requires massive code changes.</b></li> </ul> </li> <li><b>Solution:</b> Use a proxy</li> </ul>
<p><b>Read Only Handler</b></p> <ul style="list-style-type: none"> <li><b>Information Control</b> – Share a reference to an object, but do allow it to be modified. <ul style="list-style-type: none"> <li><b>Example:</b> Reference to the DOM.</li> </ul> </li> </ul> <pre>function ReadOnlyHandler(obj){   delete : function(name){     return obj[name];   }   // rcvr can be ignored   set : function(rcvr, name, val){     return true;   } }</pre>		



## Lecture #20 – Introduction to Ruby

<b>Influences of Ruby</b> <ul style="list-style-type: none"> <li>• <b>SmallTalk</b> <ul style="list-style-type: none"> <li>○ Everything is an object</li> <li>○ Blocks</li> <li>○ Metaprogramming</li> </ul> </li> <li>• <b>Perl</b> <ul style="list-style-type: none"> <li>○ Regular Expressions</li> <li>○ Function names</li> </ul> </li> </ul>	<b>Basic Ruby Syntax</b> <pre>puts "Hello World"  a = [1, 2, 3] m = { 'a' =&gt; "Apple",       'b' =&gt; "Bear",       'c' =&gt; "Cat" }  # Prints "1" puts a[0]  # Prints "Apple" puts a['m']</pre>	<b>Basic Ruby Class</b> <pre>class Person   # Constructor   def initialize name # Parameter     # Attribute     @name = name   end   # Getter   def name     return @name   end   # Setter   def name = newName     @name = name   end   # Method   def say_hi     puts "Hi my name is #{@name}"   end end</pre>	<b>Using Metaprogramming for Getters and Setters</b> <pre>class Person   # Replaces getters and setters   # Uses metaprogramming   attr_accessor :name   # Constructor   def initialize name     # Attribute     @name = name   end    # Method   def say_hi     puts "Hi my name is #{@name}"   end end</pre>
<b>Ruby on Rails</b> <ul style="list-style-type: none"> <li>• “Killer” app for Ruby           <ul style="list-style-type: none"> <li>○ <b>Lightweight web framework</b></li> <li>○ “Convention over configuration” – If use standard configuration, very little configuration required.</li> </ul> </li> <li>• Initial framework was PHP, but that was abandoned.</li> </ul>	<b>Keywords</b> <p>@ - Represents an object property</p>	<b>Returning From a Function</b> <ul style="list-style-type: none"> <li>• <b>Every function in Ruby returns a value</b>, even if return is not used.</li> <li>• If no return is specified, a function returns the last used value.</li> </ul>	<b>Using a Class in Ruby</b> <pre>p = Person.new "Joe"  puts "Name is #{p.name}"  p.say_hi  #{...} – Embeds a variable in a Ruby String</pre>

<b>Getters and Setters the Ruby Way</b> <p>Relies on metaprogramming</p> <ul style="list-style-type: none"> <li>• <b>attr_reader</b> – Getter only</li> <li>• <b>attr_writer</b> – Setter only</li> <li>• <b>attr_accessor</b> – Getter and setter</li> </ul>	<b>Inheritance in Ruby</b> <div> <div> <b>Parent Class</b> <pre>class Dog   # Parentheses optional   def initialize(name)     @name = name   end   def speak     puts "#{@name} says bark"   end end</pre> </div> <div> <b>Child Class</b> <pre>class GuardDog &lt; Dog   attr_accessor :breed   def initialize(name, breed)     # Use parent constructor     super(name)     @breed = breed   end   def attack     puts "Grrr"   end end</pre> <p><b>Note:</b> Inheritance is doing <b>using the less than (&lt;)</b> operator.</p> </div> </div>		<b>Mixin</b> <ul style="list-style-type: none"> <li>• Add features to a class</li> <li>• <b>Similar to interfaces in Java</b> with the exception that they <b>can include functionality</b>.</li> <li>• <b>module</b> – Keyword to define a Mixin.</li> <li>• <b>include</b> – Keyword to include a Mixin into a class.</li> </ul>
<b>Reopening a Class in Ruby</b> <ul style="list-style-type: none"> <li>• <b>Class definitions can be changed during runtime</b> in Ruby.</li> <li>• This is known as “reopening the class”</li> </ul>			

<b>Blocks in Ruby</b> <ul style="list-style-type: none"> <li>• <b>Superficially similar to blocks in other languages.</b></li> <li>• Create custom control structures.</li> <li>• Can be represented with <b>curly brackets</b> ({...}) or <b>do/end</b>.</li> </ul>	<b>File IO without Blocks</b> <pre>file = File.open('test.txt', 'r') file.each_line do  line    puts line end file.close</pre> <p><b>Note #1:</b> Similar “boilerplate” code of open and closing the file.</p> <p><b>Note #2:</b> It is possible one <b>may forget to the close the file</b>.</p>	<b>File IO without Blocks</b> <pre>File.open('test.txt', 'r') do  file    file.each_line {  line      puts line   } end</pre> <p><b>Note #1:</b> Eliminates the “boilerplate” code.</p> <p><b>Note #3:</b> When using a block (both <b>do/end</b>, and <b>curly brackets</b>), <b>surround the variable names in pipes ( )</b>.</p>	<b>Example: Mixin</b> <pre># Define the mixin module RevString   def to_rev_s     # Object is implicit     to_s.reverse   end end  # Reopen the Person Class class Person include RevString   def to_s     # Returns the value     @name   end end</pre>
--	---	---	--

<b>Dynamic Code Evaluation (eval)</b> <ul style="list-style-type: none"> <li>Executes source code dynamically <ul style="list-style-type: none"> <li>Code passed as either a string (or a block of code)</li> </ul> </li> <li>Popular feature in JavaScript <ul style="list-style-type: none"> <li>Early usage was to convert JSON strings to variables since not supported by JavaScript.</li> </ul> </li> <li>Source of security concerns.</li> </ul>	<b>Additional Ruby eval Methods</b> <ul style="list-style-type: none"> <li><code>instance_eval</code> – Evaluates code within an object's body. <ul style="list-style-type: none"> <li>Access the internals of an object.</li> </ul> </li> <li><code>class_eval</code> – Evaluates code within a class' body. <ul style="list-style-type: none"> <li>Modifies the class' definition.</li> </ul> </li> <li>Takes either a string or block of code. Block of code is more secure.</li> </ul>	<b>Example: Use <code>instance_eval</code> to Change an Object's Value</b> <pre># Create with the name Bob bob = Person.new "Bob"  # Change his name bob.instance_eval do   @name = "Steve" end  # Prints "Steve" puts bob.name</pre>	<b>Regular Expressions in Ruby</b> <ul style="list-style-type: none"> <li><code>sub</code> – Replaces the first instance of a string match. <ul style="list-style-type: none"> <li>To perform the modification in place, must include an exclamation point (!) after sub.</li> </ul> </li> <li><code>gsub</code> – Replaces all instance of a string match. <ul style="list-style-type: none"> <li>To perform the modification in place, must include an exclamation point (!) after sub.</li> </ul> </li> </ul>
---	--	---	--

<b>Example: <code>class_eval</code> in Ruby</b> <pre># Applies to all classes class Class   # Simulate the "attr_accessor" function   def my_attr_accessor(args)     args.each do  prop        # Create getter       self.class_eval("def #{prop};         return @#{prop};       end")        # Create setter       self.class_eval("def #{prop} = v;         @#{prop} = v;       end")     end   end end  # Use the new attribute class Musician   my_attr_accessor :name, :genre end  m = Musician.new m.name = "Bob Marley" puts m.name # Prints "Bob Marley"</pre>	<b>Example: Using Regular Expressions in Ruby</b> <pre>s = "Hi, I'm Larry; this is my" +   " brother Darryl, and this" +   " is my other brother Darryl." s.sub(/Larry/, 'Laurent')  # Prints s unchanged puts s  # Changes first "Larry" to "Laurent" s.sub!(/Larry/, 'Laurent') puts s  # Prints first "brother" replaced with # "frere". s is unchanged, bt it did # return the modified string. puts s.sub(/brother/, 'frère')  # Same as previous except all where # changed when printing. puts s.gsub(/brother/, 'frère')</pre>
---	--

### Regular Expression Symbols in Ruby

/. / - Any character except a newline	/\w/ - Any word character: [a-zA-Z0-9_]	/\d/ - Any digit character: [0-9]	/\s/ - Any whitespace character: [ \t\r\n\f]
	/\W/ - Any non-word character: [^a-zA-Z0-9_]	/\D/ - Any non-digit character: [^0-9]	/\S/ - Any non-whitespace character: [^ \t\r\n\f]
* - Zero or more times	+ - One or more times	? - Zero or one times (optional)	

### Important Syntax in Ruby

<b>For Each Loop</b> <pre>object.each do  val    ... end</pre>	<b>Create a Mixin</b> <pre>module Name   ... end</pre>	<b>Return from a Block</b> <pre>def block_name   ...   yield x   ... end</pre>	<b>Single Line If Statement</b> <pre>x = 5 # Does nothing x = 3 if (x &gt; 10) puts x # Prints "5"</pre>	<b>Ranges in Ruby</b> <pre># Create array from 1 to 5 x = (1..5)</pre> <p>Note: Uses parentheses.</p>
<b>irb</b> – Command line for Ruby similar to GHCi.	<b>Use a Mixin in a Class</b> <pre>class MyClass   include MixinName   ... end</pre>			

## Lecture #21 – Blocks and Messages

<p><b>Influence of Smalltalk on Ruby</b></p> <ul style="list-style-type: none"><li>Everything is an object</li><li>Blocks</li><li>Message passing</li></ul>	<p><b>Benefits of Blocks in Ruby</b></p> <ul style="list-style-type: none"><li>Create custom control structures</li><li>Eliminate boilerplate code.</li><li>Ruby blocks are closures, but they are different than JavaScript blocks.</li></ul>	<p><b>Example: <code>do_noisy</code> Block</b></p> <pre>def do_noisy   puts "About to call block"   yield # Calls block code   puts "Just called block" end</pre> <p><b>Note:</b> Called with a <code>do/end</code> or with curly brackets.</p>	<p><b>Example: Extend Array Class to Return Lowercase Version of Every Element</b></p> <pre># Reopen the Array class class Array   def each_downcase     self.each do  val        yield val.downcase     end   end end</pre>	<p><b>Example: Using the <code>each_downcase</code> Block</b></p> <pre>arr = ["Alpha", "Beta", "So On"] arr.each_downcase do  val    puts val end</pre>		
<p><b>Example: Probabilistic Run Block</b></p> <pre># Probabilistic Run Block def with_prob(prob)   yield if (Random.rand &lt; prob) end  with_prob 0.42 do   puts "Prints 42% of time." end</pre>		<p><b>Example: Passing Code to a Block</b></p> <pre>def with_prob2(prob, &amp;blk)   blk.call if (Random.rand &lt; prob) end</pre> <p><code>blk</code> – Block of code passed to the function.</p> <p><b>Note #1:</b> Argument name has an ampersand (&amp;) before it.</p> <p><b>Note #2:</b> No ampersand is used when calling the block.</p>	<p><b>Example: Sharing Code Between Blocks</b></p> <pre>def half_the_time(prob, &amp;blk)   with_prob2(0.5, &amp;blk) end</pre> <p><b>Note:</b> Need to pass argument to the function with the ampersand (&amp;).</p>			
<p><b>Example: <code>with_prob</code> in JavaScript</b></p> <pre>function with_prob(prob, f){   if(Math.random() &lt; prob){     return f();   } }</pre> <p><b>Note:</b> The JavaScript implementation relies on <b>callbacks</b>.</p>		<p><b>Example: Difference Between Ruby and JavaScript Blocks</b></p> <table><tr><td><p><b>Ruby</b></p><pre>def coin_flip   with_prob 0.5 do     return "Heads"   end   return "Tails" end</pre><p><b>Note:</b> This returns “Heads” half the time and “Tails” half the time.</p><ul style="list-style-type: none"><li>This is because a <b>return in a Ruby block returns for the entire function</b>.</li></ul></td><td><p><b>JavaScript</b></p><pre>function coin_flip(){   with_prob(0.5, function(){     return "Heads";   })   return "Tails"; }</pre><p><b>Note:</b> This always returns “Tails”</p><ul style="list-style-type: none"><li>This is because even if “with_prob” runs, the return only occurs within the anonymous function.</li></ul></td></tr></table>			<p><b>Ruby</b></p> <pre>def coin_flip   with_prob 0.5 do     return "Heads"   end   return "Tails" end</pre> <p><b>Note:</b> This returns “Heads” half the time and “Tails” half the time.</p> <ul style="list-style-type: none"><li>This is because a <b>return in a Ruby block returns for the entire function</b>.</li></ul>	<p><b>JavaScript</b></p> <pre>function coin_flip(){   with_prob(0.5, function(){     return "Heads";   })   return "Tails"; }</pre> <p><b>Note:</b> This always returns “Tails”</p> <ul style="list-style-type: none"><li>This is because even if “with_prob” runs, the return only occurs within the anonymous function.</li></ul>
<p><b>Ruby</b></p> <pre>def coin_flip   with_prob 0.5 do     return "Heads"   end   return "Tails" end</pre> <p><b>Note:</b> This returns “Heads” half the time and “Tails” half the time.</p> <ul style="list-style-type: none"><li>This is because a <b>return in a Ruby block returns for the entire function</b>.</li></ul>	<p><b>JavaScript</b></p> <pre>function coin_flip(){   with_prob(0.5, function(){     return "Heads";   })   return "Tails"; }</pre> <p><b>Note:</b> This always returns “Tails”</p> <ul style="list-style-type: none"><li>This is because even if “with_prob” runs, the return only occurs within the anonymous function.</li></ul>					
<p><b>Singleton Classes</b></p> <ul style="list-style-type: none"><li>In Ruby, every object has its own singleton class.</li><li>This class holds <b>methods and fields unique to that object</b>.</li><li>This is different from Singleton Objects in design patterns.</li></ul>	<p><b>Example: Adding a Property to a <i>Variable</i> in JavaScript</b></p> <pre>function Employee(name, salary){   this.name = name;   this.salary = salary; } var a = new Employee("Alice", 500); var b = new Employee("Bob", 1000);  // Add a signing bonus to "Alice" a.signingBonus = 2000;</pre>		<p><b>Example: Adding a Property to an <i>Object</i> in Ruby</b></p> <pre>class Employee   attr_accessor :name, :salary   def initialize(name, salary)     @name = name     @salary = salary   end   def to_s     @name # No return required   end end  # Create the Objects a = Employee.new("Alice", 500) b = Employee.new("Bob", 1000)  # Access the singleton class of "a" class &lt;&lt; a   def signing_bonus     2000   end end</pre>			
	<p><b>Accessing Singleton Classes in Ruby</b></p> <ul style="list-style-type: none"><li>To open an object’s singleton class, use <b>double less than symbols</b> (“&lt;&lt;”).</li><li>Code only added to the specific object being reference.</li></ul>					

<p><b>Example:</b> Using a Singleton Class to Create Static Methods</p> <pre># Add Static Methods to Employee Class class Employee   class &lt;&lt; self     def get_employee_by_name(name)       @employee[name] # No return needed     end     # Called in constructor     def add(emp)       puts "Adding #{emp}"       # Create map if not exist       @employee = Hash.new unless @employee       @employee[emp.name] = emp     end   end end</pre>	<p><b>Message Passing</b></p> <ul style="list-style-type: none"> <li>Represents <b>inter-object interaction</b>.</li> <li>Sender Sends:           <ul style="list-style-type: none"> <li>Method name</li> <li>Data: <b>Method parameters</b> (if any)</li> </ul> </li> <li>Receiver:           <ul style="list-style-type: none"> <li>Processes the message</li> <li>(<i>Optionally</i>) returns data</li> </ul> </li> <li><b>Receiver may not understand the message.</b></li> </ul>	<p><b>Example:</b> <code>missing_method</code> in Ruby</p> <pre>class Person   attr_accessor :name   def initialize(name)     @name = name   end   # Called when method unknown   def method_missing(m)     puts "Didn't understand #{m}"   end end</pre>
	<p><b>method_missing</b></p> <ul style="list-style-type: none"> <li>Method that is part of every class. Can be overridden.           <ul style="list-style-type: none"> <li><b>Smalltalk Name:</b> <code>doesNotUnderstand</code></li> <li><b>Ruby Name:</b> <code>method_missing</code></li> </ul> </li> <li>Invoked whenever an unknown method is called.</li> </ul>	<p><b>Active Record and Message Passing</b></p> <ul style="list-style-type: none"> <li>Relational database tool in Ruby.</li> <li>Specify fields in the database to be extracted based off method names. <b>Example:</b></li> </ul> <pre>Person.find_by_first_name "John"</pre>

## Lecture #22 – Virtual Machines and Just-In Time Compilation

<b>Virtual Machine Overview</b> <ul style="list-style-type: none"> <li>Code is compiled to bytecode <ul style="list-style-type: none"> <li>Byte code is low level</li> <li>Platform independent</li> </ul> </li> <li>The VM interprets the bytecode</li> </ul>	<b>Supported VM Operations</b> <ul style="list-style-type: none"> <li><b>PUSH</b> – Adds an argument to the stack</li> <li><b>PRINT</b> – Pops an argument off the stack and prints it.</li> <li><b>ADD</b> – Pops two elements off the stack, adds them, and places result on the stack.</li> <li><b>SUB</b> – Similar to add but for subtraction. If “A” is on the top of the stack and “B” is below it, the result is B – A</li> <li><b>MUL</b> – Similar to add but with multiplication.</li> </ul>	<b>Compilers vs. Interpreters vs. JIT</b> <ul style="list-style-type: none"> <li><b>Compiler</b> <ul style="list-style-type: none"> <li>Efficient execution</li> </ul> </li> <li><b>Interpreter</b> <ul style="list-style-type: none"> <li>Runtime flexibility</li> </ul> </li> <li><b>JIT</b> <ul style="list-style-type: none"> <li>Efficient execution with runtime flexibility.</li> </ul> </li> </ul>	<b>Just-In-Time Compilers</b> <ul style="list-style-type: none"> <li>Interpret code</li> <li>“Hot” (i.e., heavily-used) sections are compiled at runtime.</li> <li><b>Advantages</b> <ul style="list-style-type: none"> <li>Speed of compiled code</li> <li>Flexibility of interpreter</li> </ul> </li> <li><b>Disadvantages</b> <ul style="list-style-type: none"> <li>Overhead of compiler and interpreter</li> <li>Complex implementation</li> </ul> </li> </ul>
<b>Scheme</b> <ul style="list-style-type: none"> <li>Similar to an AST. Uses parentheses.</li> <li>Relies on a stack.</li> </ul>			
<b>Dynamic Recompilation</b> <ul style="list-style-type: none"> <li>JIT pursues aggressive optimizations <ul style="list-style-type: none"> <li>Makes assumptions about the code</li> <li>Guard conditions verify assumptions</li> </ul> </li> <li>Unexpected cases are interpreted (i.e., not compiled)</li> <li>Can in some corner cases outperform static compilation.</li> </ul>	<b>Types of JITs</b> <ul style="list-style-type: none"> <li><b>Method Based</b> – Compile Methods</li> <li><b>Trace Based</b> – Compile loops</li> </ul>	<b>How to Support JITs for a Language</b> <ul style="list-style-type: none"> <li><b>Option #1:</b> Build your own JIT. <ul style="list-style-type: none"> <li>Study the latest techniques</li> <li>Build large code bases to test.</li> <li>Profile the code execution</li> </ul> </li> <li><b>Option #2:</b> Use someone else’s Just-In-Time VM.</li> </ul>	

# Final Exam Review Notes

<p><b>Parsec</b></p> <ul style="list-style-type: none"> <li>Parser combinator in Haskell.</li> <li><b>Example Question:</b> Write a grammar at the level of the CSV parser.</li> </ul>	<p><b>JavaScript</b></p> <ul style="list-style-type: none"> <li>Prototype based language. <ul style="list-style-type: none"> <li><b>Inherit from an object not from a class.</b></li> <li>Add properties and methods on the fly.</li> </ul> </li> <li><b>Closure</b> – By wrapping an inner function with an outer function, the inner function can encapsulate variables. <ul style="list-style-type: none"> <li><b>Have a scope chain.</b></li> </ul> </li> </ul>	<p><b>Example: Write a Function that Toggles a Variable Each Time the Function is Called</b></p> <p><b>Example #1: Use a Global</b></p> <pre>var b = true; function flip() {   if (b) b = false;   else b = true;   return b; }</pre>	<p><b>Example #2: Use a Closure</b></p> <pre>var flip = function() {   var b = true;   return function() {     if (b) b = false;     else b = true;     return b;   }; }(); // Needed to call out function</pre>
<p><b>JavaScript – Multiparadigm Language</b></p> <ul style="list-style-type: none"> <li><b>Imperative</b></li> <li><b>Object Oriented</b></li> <li><b>Functional</b> <ul style="list-style-type: none"> <li>Supports higher order functions but <b>not purely functional</b> since not immutable.</li> </ul> </li> </ul>	<p><b>JavaScript Scoping</b></p> <ul style="list-style-type: none"> <li><b>No block scope</b></li> <li><b>Variable declaration hoisting to the top of the function.</b></li> </ul>	<p><b>this in JavaScript</b></p> <ul style="list-style-type: none"> <li>In a <b>method</b>, this refers to the associated object.</li> <li>In a <b>non-method</b>, this refers to the global scope.</li> <li><b>Constructor (with new)</b> – Refers to the object being created.</li> <li><b>DOM for Event Listeners</b> – Refers to the DOM element.</li> <li><b>apply, call, bind</b> – User can define what “this” refers to.</li> </ul>	<p><b>Scope Precedence in JavaScript</b></p> <p><b>From highest to lowest precedence</b></p> <ol style="list-style-type: none"> <li><b>Variable object</b> (i.e., local variables)</li> <li><b>Scope Chain</b></li> <li><b>Global object</b> (i.e., <b>this</b>)</li> </ol>
<p><b>Example: Scope Chain</b></p> <pre>function my_hello() {   var x = 5;   // Scope Chain   function print_hello() {     console.log("Hello " + x);   }   print_hello(); }  my_hello();</pre>	<p><b>Quirks of JavaScript</b></p> <ul style="list-style-type: none"> <li>Semicolon insertion</li> <li><b>typeof</b> <ul style="list-style-type: none"> <li><b>typeof NaN</b> – Number</li> <li><b>typeof null</b> – Object</li> </ul> </li> <li><b>== Not Transitive</b> <ul style="list-style-type: none"> <li>If <b>a == b</b> and <b>b == c</b>, it is not guaranteed that <b>a == c</b></li> </ul> </li> </ul>	<p><b>JSLint</b></p> <ul style="list-style-type: none"> <li>Designed to catch common JavaScript errors.</li> <li>Based off lint for the C language.</li> <li>Performs <b>static code analysis</b></li> </ul>	<p><b>TypeScript</b></p> <ul style="list-style-type: none"> <li>Developed by Microsoft</li> <li><b>Source-to-source compiler</b> (i.e., transpiler)</li> <li><b>Compiles to JavaScript</b></li> <li><b>Provides type annotation and classes.</b></li> </ul>
<p><b>Event Based Programming</b></p> <ul style="list-style-type: none"> <li>Relies on <b>listeners</b></li> <li><b>Events are placed into an event queue.</b></li> <li><b>No concurrency.</b></li> <li><b>emit</b> – Invokes an event</li> <li><b>on</b> – Registers a listener</li> <li><b>Client-Based Programming</b> – Often used in GUIs.</li> </ul>	<p><b>Metaprogramming</b></p> <ul style="list-style-type: none"> <li><b>Reflection</b> – Two primary categories. <b>Both occur at runtime.</b> <ul style="list-style-type: none"> <li><b>Introspection</b> – Examine program’s execution at runtime.</li> <li><b>Self-modification</b> – Modify a program execution at runtime.</li> </ul> </li> <li><b>Intercession</b> – Trigger or control interaction at runtime.</li> <li>Reflection is more common, but intercession is more powerful.</li> </ul>	<p><b>Aspect-Oriented Programming</b></p> <p>Designed to address cross-cutting code (i.e., code that is interspersed everywhere in a program).</p>	<p><b>Metaobject Protocols</b></p> <ul style="list-style-type: none"> <li><b>Metaobject</b> – Any object that can reason about the behavior of other objects. <ul style="list-style-type: none"> <li><b>Example:</b> Proxies</li> </ul> </li> <li><b>Handler</b> <ul style="list-style-type: none"> <li>Type of Meta Object.</li> <li><b>Defines traps.</b></li> </ul> </li> <li><b>Trap</b> – Methods that intercept an operation.</li> </ul>
<p><b>Macros Sweet.js</b></p> <ul style="list-style-type: none"> <li>Source-to-source compiler (<b>transpiler</b>)</li> <li><b>Hygiene</b> – <b>No inadvertent variable capture.</b></li> <li><b>Text Substitution Macro</b> – Works at the lexeme or text level.</li> <li><b>Syntactic Macro</b> – Works at the Abstract Syntax Tree (AST) level.</li> </ul>	<p><b>Lambda Calculus</b></p> <ul style="list-style-type: none"> <li><b>Simple, Turing complete language.</b></li> <li>Based off anonymous functions (lambda)</li> <li><b>Expressions:</b> <math display="block">x \mid \lambda x. e \mid e e</math> </li> <li><b>Values:</b> <math display="block">\lambda x. e</math> </li> </ul>	<p><b>Evaluation Strategies</b></p> <ul style="list-style-type: none"> <li><b>Strict Evaluation Strategies</b> <ul style="list-style-type: none"> <li><b>Call by Value</b> – Pass a <b>copy</b> of the parameter.</li> <li><b>Call by Reference</b> – <b>Implicit reference</b> (e.g., a pointer to the parameter is passed).</li> </ul> </li> <li><b>Lazy Evaluation Strategies</b> <ul style="list-style-type: none"> <li><b>Call by Name</b> – Re-evaluate the expression each time it is needed.</li> <li><b>Call by Need</b> – Evaluate when needed and memo-ize the result.</li> </ul> </li> </ul>	<p><b>Advantages and Disadvantages of Type Systems</b></p> <p><b>Benefits:</b></p> <ul style="list-style-type: none"> <li>Enforced documentation</li> <li>Tips for IDEs and Developers</li> <li>Prevent code with errors from running.</li> </ul> <p><b>Disadvantage:</b></p> <ul style="list-style-type: none"> <li>May prevent valid code from running.</li> </ul>
<p><b>Simply-Type Lambda Calculus</b></p> <ul style="list-style-type: none"> <li>Relies on a <b>typing environment (Γ)</b></li> <li><b>Not Turing complete.</b></li> </ul> <p><b>Type Safety – Two Components</b></p> <ul style="list-style-type: none"> <li><b>Progress</b> – Valid input continues to evaluation or reaches a value.</li> <li><b>Preservation</b> – Evaluation does not change the type of an object.</li> </ul>	<p><b>Influences of Ruby</b></p> <ul style="list-style-type: none"> <li><b>SmallTalk</b> <ul style="list-style-type: none"> <li>Everything is an object</li> <li>Blocks</li> <li><b>Metaprogramming</b> – <b>method_missing</b> and message passing.</li> </ul> </li> <li><b>Perl</b> <ul style="list-style-type: none"> <li>Regular expressions</li> <li>Names of functions.</li> </ul> </li> </ul>	<p><b>Goals of Ruby</b></p> <ul style="list-style-type: none"> <li>Object oriented scripting language.</li> <li>Dynamically typed</li> <li>Interpreted.</li> </ul>	<p><b>Ruby Features</b></p> <ul style="list-style-type: none"> <li><b>eval</b> – Execute a string as code.</li> <li><b>Singleton Classes</b> <ul style="list-style-type: none"> <li>No relation to singleton objects</li> <li><b>Class for a single object.</b></li> <li>Can be used to create static methods in a class.</li> </ul> </li> </ul>



<p><b>Virtual Machine</b></p> <ul style="list-style-type: none"><li>• Source compiled to <b>byte code</b>.</li><li>• <b>Byte code executed by an interpreter.</b></li></ul>	<p><b>Interpreters vs. Compilers</b></p> <ul style="list-style-type: none"><li>• <b>Interpreter</b> – Runtime flexibility</li><li>• <b>Compiler</b> – Efficient code</li><li>• <b>JIT</b> – <b>Benefits</b> <i>and</i> <b>overhead</b> of both an interpreter and compiler.</li></ul>	<p><b>Just-In-Time Compiler</b></p> <ul style="list-style-type: none"><li>• Identify <b>“hot”</b> (i.e. frequently run) code.</li><li>• <b>Optimize for most common cases and skip corner cases.</b><ul style="list-style-type: none"><li>◦ Guards protect for corner cases which are interpreted.</li></ul></li><li>• <b>May outperform statically compiled code.</b></li></ul>
---	---	--

```
var b = true;
function flip(){
  if(b) b = false;
  else b= true;
  return b;
}

var flip = function(){
  var b = true;
  return function(){
    if(b) b = false;
    else b = true;
    return b;
  };
};
```