

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"> 1. Different domains (e.g. web, security, bioinformatics) 2. Legacy code and libraries 3. Personal preference 		<ol style="list-style-type: none"> 1. Flexibility 2. Type safety 3. Performance 4. Build Time 5. Concurrency 		<ol style="list-style-type: none"> 1. Simplicity 2. Readability 3. Learnability 4. Safety (e.g. security and can errors be caught at compile time) 5. Machine independence 6. Efficiency 	
				Goals almost always conflict	
Conflict: Type Systems <ul style="list-style-type: none"> • Advantage: Prevents bad programs. • Disadvantage: Reduces programmer flexibility. 		Blub Paradox: 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.		Current Programming Language Issues <ul style="list-style-type: none"> • Multi-core “explosion” • Big Data • Mobile Devices 	
				Advantages of Web and Scripting Languages <ul style="list-style-type: none"> • Examples: Perl, Python, Ruby, PHP, JavaScript • Highly flexible • Dynamic typing • Easy to get started • Minimal typing (i.e. type systems) 	
Major Programming Language Research Contributions <ul style="list-style-type: none"> • Garbage collection • Sound type systems • Concurrency tools • Closures 		Programs that Manipulate Other Programs <ul style="list-style-type: none"> • Compilers & interpreters • JavaScript rewriting • Instrumentation • Program Analyzers • IDEs 		Formal Semantics <ul style="list-style-type: none"> • Used to share information unambiguously • Can formally prove a language supports a given property • Crisply define how a language works 	
				Types of Formal Semantics <ul style="list-style-type: none"> • Operational <ul style="list-style-type: none"> ◦ Big Step “natural” ◦ Small Step “structural” • Axiomatic • Denotational 	

Haskell

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

Lecture #02 – Introduction to Haskell

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

Primitive Classes in Haskell <ol style="list-style-type: none"> 1. Int – Bounded Integers 2. Integer – Unbounded 3. Float 4. Double 5. Bool 6. Char 	Lists <ul style="list-style-type: none"> • Base 0 • Comma separated in square brackets • Operators <ul style="list-style-type: none"> ◦ : Prepend ◦ ++ Concatenate ◦ !! Get element a specific index ◦ head First element in list ◦ tail All elements after head 		Ranges <ul style="list-style-type: none"> • Can be infinite or bounded • Use the “..” notation. Examples: <pre>> [1..4] [1, 2, 3, 4] > [1,2..6] [1, 2, 3, 4, 5, 6] > [1,3..10] [1, 3, 5, 7, 9] > [5, 4..1] [5, 4, 3, 2, 1]</pre> 	
	List Examples <pre>> putStrLn \$ "Hello " ++ "World" "Hello World" > let s = bra in s !! 2 : s ++ 'c' : last s : 'd' : s "abracadabra"</pre>		Infinite List Example <pre>> let even = [2,4..] > take 5 even [2, 4, 6, 8, 10]</pre>	

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

<p>Recursion</p> <ul style="list-style-type: none"> Base Case – Says when recursion should stop. Recursive Step – Calls the function with a smaller version of the problem <p>Example:</p> <pre>addNum :: [Int] -> Int addNum [] = 0 addNum (x:xs) = x + addNum xs</pre>	<p>Lab #01 – Max Number</p> <pre>> maxNum :: [Int] -> Int > maxNum [] = error "Invalid Input" > maxNum [x] = x > maxNum (x:xs) = if x > max xs then x else max xs > where max xs = maxNum xs</pre>	<p>Reasons for a Large Number of Programming Languages</p> <ul style="list-style-type: none"> Different domains Different design choices
--	--	---

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

Lecture #03 – Operational Semantics

<div>Formal Semantics</div> <div>Crisply define how the language features work.</div>	<div>Formal Semantic Styles</div> <ul style="list-style-type: none">Operational – Specify how expressions should be evaluated.<ul style="list-style-type: none">Big-Step (“Natural”)Small-Step (“structural”)AxiomaticDenotational	<div>A Review of Compilers</div> <pre>graph LR; SC[source code] --> LT[Lexer/Tokenizer]; LT -- tokens --> P[Parser]; P --> AST[Abstract Syntax Tree (AST)]; AST --> C[Compiler]; AST --> I[Interpreter]; C --> MC[Machine code]; I --> Com[Commands];</pre> <div>We don't care about lexing or parsing.</div> <div>We don't care if we have a compiler or interpreter</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>Big Step Operational Semantics</div> <ul style="list-style-type: none">Evaluates every expression to a value <div>\Downarrow : “Evaluates to” symbol in Big-Step operational semantics.</div> <div>Example Formatting:</div> <div>$e \Downarrow v$</div> <ul style="list-style-type: none">Read as: “Expression e evaluates to the value v”					
<div>Bool * Language</div> <table><tr><td><div>$e ::=$</div><div>true false if e then e else e</div></td><td><div>Expressions:</div><div>constant true constant false conditional</div></td></tr><tr><td><div>$v ::=$</div><div>true false</div></td><td><div>Values:</div><div>constant true constant false</div></td></tr></table>	<div>$e ::=$</div> <div>true false if e then e else e</div>	<div>Expressions:</div> <div>constant true constant false conditional</div>	<div>$v ::=$</div> <div>true false</div>	<div>Values:</div> <div>constant true constant false</div>		
<div>$e ::=$</div> <div>true false if e then e else e</div>	<div>Expressions:</div> <div>constant true constant false conditional</div>					
<div>$v ::=$</div> <div>true false</div>	<div>Values:</div> <div>constant true constant false</div>					

<p>Small-Step Operational Semantics</p> <ul style="list-style-type: none"> Evaluate an expression until it is in normal form Normal Form – Any form that cannot be evaluated further. → : "Evaluates to" symbol in small step operational semantics. Example: $e \rightarrow e' \rightarrow e'' \rightarrow v$ →* : Many evaluation steps required. Example: $e \rightarrow^* v$ 	<p>Bool* Small-Step Operational Semantics Rules</p> <p>E-IfTrue:</p> $\frac{}{\text{if true then } e_2 \text{ else } e_3 \rightarrow e_2}$ <p>E-IfFalse:</p> $\frac{}{\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3}$ <p>E-If:</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>Bool* Extension: Numbers</p> <ul style="list-style-type: none"> • 0 : The Number "0" • succ 0 : Represents "1" • succ succ 0 : Represents "2" • pred n : Gets the predecessor of "n" 	<p>Extended Bool * Language</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>Literate Haskell</p> <ul style="list-style-type: none"> • File Extension: ".lhs" • Code lines begin with ">" • All other lines are comments. • "Essentially swaps code with comments." 	<p>Case Statement in Haskell</p> <ul style="list-style-type: none"> • Keywords: case, of, otherwise • Operator: -> <p>Example:</p> <pre> case x of val1 -> "Value 1" val2 -> "Value 2" otherwise -> "Everything else." </pre>
--	--	---	---

Lab #02 Review

<p>Bool Expression Type</p> <pre> > data BoolExp = BTrue BFalse Bif BoolExp BoolExp BoolExp B0 Bsucc BoolExp Bpred BoolExp deriving Show </pre>	<p>BoolVal Type</p> <pre> > data BoolVal = BVTrue BVFalse BVNum BVInt deriving Show > data BVInt = BV0 BVSucc BVInt deriving Show </pre>	<p>Type Constructors: BoolExp, BoolVal, BVInt</p> <p>Non-nullary Value Constructors: Blf, Bsucc, Bpred, BVSucc, BVNum</p> <p>Note: 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>Lambda</p> <ul style="list-style-type: none"> • Analogous to anonymous classes in Java. • Based off Lambda calculus • Example: <pre> > (\x -> x + 1) 1 2 > (\x y -> x + y) 2 3 5 </pre>	<p>Function Composition</p> <ul style="list-style-type: none"> • Uses the period (.) • f(g(x)) can be rewritten (f . g) x 	<p>Point-Free Style</p> <ul style="list-style-type: none"> • Pass no arguments to a function • Example: <pre> > let inc = (+1) -- No args > inc 3 4 </pre>	<p>Example: Lambda with Function Composition</p> <pre> > let f = (\x -> x - 5) . (\y -> y * 2) > f 7 9 > let f = (\x y -> x - y) . (\z -> z * (-1)) > f 3 4 -7 </pre>
--	---	--	---

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

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

Higher Order Functions

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

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

Lecture #05 – Small-Step Operational Semantics

<p>WHILE Language</p> <ul style="list-style-type: none"> Unlike the Bool* language, WHILE supports mutable references. 	<p>Small Step Semantics with State</p> <ul style="list-style-type: none"> Since the WHILE language supports mutable references, the grammar must be updated to support it. <p>While Relation:</p> $e, \sigma \rightarrow e', \sigma'$ <ul style="list-style-type: none"> σ – Store. Maps references to values. <p>Example Operations:</p> <ul style="list-style-type: none"> $\sigma(a)$ – Retrieves the value at address "a" $\sigma[a := v]$ – Identical to the original store with the exception that it now stores the value v at address "a" 	<p>Evaluation Order Rules</p> <ul style="list-style-type: none"> Tend to be repetitive and clutter the semantics. Context based rules tend to represent the same information as evaluation order rules but more concisely. <p>Reduction Rule</p> <p>Rewrites the expression. Example:</p> <p>E-IfFalse:</p> <pre>if false then e2 else e3 → e3</pre> <p>Context Rule</p> <p>Specify the order for evaluating expressions. Example:</p> <p>E-If:</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>Reducible Expression (Redex) – Any expression that can be transformed (reduced) in one step.</p>	<p>Example: Redex</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>Example: Not a Redex</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>Evaluation Contexts</p> <ul style="list-style-type: none"> Alternative to evaluation order rules. Marker (•) / hole indicate the next place for evaluation (i.e. where we will do the work). <p>Example:</p> <pre>C[r]</pre> <pre>= if (if true then false else false) then true else true</pre> <p>r = if true then false else false</p> <p>C = if • then true else true</p> <p>C[r] is the original expression.</p>	<p>Rewriting Evaluation Order Rules</p> <p>Context based rules only apply to reducible expressions (redexs). Example:</p> <p>EC-IfFalse:</p> <pre>C[if false then e2 else e3] → C[e3]</pre> <p>Context Syntax</p> <pre>C ::= • if C then e else e C op e v op C ...</pre>	<p>Data.Map</p> <ul style="list-style-type: none"> Library: import Data.Map as Map Immutable Example Methods: <ul style="list-style-type: none"> Map.empty – Creates and returns an empty map Map.insert k v m – Inserts a value "v" at key "k" into map "m". Returns a new, updated map. Map.lookup k m – Returns the value at key "k" in map "m". Wrapped in a Maybe. Map.member k m – Returns true if k is in map "m" and false otherwise.
---	---	---

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

Lecture #06 – LaTeX

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

Create Section with Label <code>\section{Section #1}</code> <code>\label{sec:one}</code> Create Subsection with Label <code>\subsection{<SubsectionName>}</code> <code>\label{sec:<refName>}</code> Use of Tilde (~) Creates an undividable space so the text "Section~\ref{sec:one}" will appear on one line	BibTeX <ul style="list-style-type: none"> References are tedious to reformat and renumber. Reference details shorted in a "*.bib" file. Create a Bibliography <code>\bibliography{biblio}</code> BibTeX filename for the example would be "biblio.bib" Define Bibliography Style <code>\bibliographystyle{plainurl}</code>	BibTeX Article Reference Example <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

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

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

Kinds		Typeclasses	
<ul style="list-style-type: none"> • “The type of types”. • Concrete types have a kind of “*” • Keyword :k, :kind • Example: <pre>> :k Tree Tree :: * -> *</pre> <p>Explanation: A Tree requires one type parameter (e.g. Int) to be made a concrete type.</p>	<p>String Kind</p> <pre>> :kind String String :: *</pre> <p>Map Kind</p> <pre>> :k Map Map :: * -> * -> *</pre> <p>Maybe Kind</p> <pre>> :k Maybe Map :: * -> *</pre> <p>Map String Kind</p> <pre>> :kind (Map String) (Map String) :: * -> *</pre> <p>Explanation: Map String is has one of the two type parameters filled so it has one less asterisk.</p>	<ul style="list-style-type: none"> • Similar to interfaces in Java. <ul style="list-style-type: none"> ◦ Like a contract. ◦ Implementation details can be included in typeclass definition. • No relation to classes in object-oriented programming. <ul style="list-style-type: none"> ◦ Example: Do not have any data associated with them. • Simplify polymorphism. <p>Example: Eq Typeclass</p> <pre>class Eq a where (==) :: a -> a -> Bool (/=) :: a -> a -> 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>Example: Make Maybe an Instance of Eq</p> <pre>instance (Eq a) => 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 class constraint.</p> <p>Class Constraint</p> <ul style="list-style-type: none"> • Operator: => • Ensures that a type parameter satisfies some typeclass requirement. <p>Kind of Typeclasses</p> <pre>> :k Eq Eq :: * -> Constraint</pre> <pre>> :k Num Num :: * -> Constraint</pre> <p>Note: Typeclasses are a class constraint (not a type) so their kind is different.</p>

Lecture #08 – Functors

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

IO in Haskell

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

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

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

Functor Laws

<p>Functor Law #1: 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>Examples:</p> <pre>> fmap id (Just 3) Just 3 > fmap id Nothing Nothing > fmap id [1, 2, 3] [1, 2, 3]</pre>	<p>Functor Law #2: 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>Law #2 Written Formally</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>Functor – Something that can be mapped over. Allow you to map functions over different data types. Examples:</p> <ul style="list-style-type: none"> • Maybe • Either • IO • Lists • <*> <p>Functors return boxed up values.</p>	<p>Functor Example</p> <pre>> fmap (+1) [1, 2, 3] [2, 3, 4] > let x = fmap (+) [1, 2, 3]</pre> <p>Explanation: In this case x is: [(1+), (2+), (3+)]</p>	<p>Applicative Functor</p> <ul style="list-style-type: none"> • Requires the importing of a special library as shown below: <pre>import Control.Applicative</pre> <p>Functions in Applicative Typeclass:</p> <ul style="list-style-type: none"> • pure – Wraps/boxes a value • <*> - Infix version of fmap. Is itself a Functor. 	<p>Example Uses of pure</p> <pre>> pure 7 7 > pure 7 :: Maybe Int Just 7 > pure 7 :: [Int] [7]</pre>
---	---	---	--

<p>Type Class Definition of Applicative</p> <pre>class (Functor f) => Applicative f where pure :: a -> f a <*> :: f (a -> b) -> f a -> f b</pre> <p>Only difference between <*> and fmap is that the function in <*> is boxed while it is not in fmap (see the green f).</p>	<p>Make Maybe an Instance of Applicative</p> <pre>instance Applicative Maybe where pure = Just Nothing <*> _ = Nothing (Just f) <*> x = fmap f x</pre> <p>Explanation: 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>Examples of Applicative Maybe</p> <pre>> Just (+3) <*> Just 4 Just 7 > pure (+3) <*> Just 4 Just 7 > pure (+) <*> Just 3 <*> Just 4 Just 7 > (+) <\$> Just 3 <*> Just 4 Just 7</pre> <p>Explanation: x <\$> is fmap as an infix operator. It is NOT necessarily the same as pure x <*>. It should be based off Applicative Functor Law #1.</p>
---	---	--

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

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

<p>Example of liftA2</p> <pre>> (:) <\$> Just 3 <*> Just [4] Just [3, 4] > liftA2 (:) (Just 3) (Just [4]) Just [3, 4]</pre>	<p>Applicative Functor Definition</p> <p>A functor you can apply to other Functors.</p>
--	---

Applicative Functor Laws

<p>Law 1:</p> <pre>pure f <*> x = fmap f x</pre>	<p>Law 2:</p> <pre>pure id <*> v = v</pre>	<p>Law 3:</p> <pre>pure (.) <*> u <*> v <*> w = u <*> (v <*> w)</pre>
<p>Law 4:</p> <pre>pure f <*> pure x = pure (f x)</pre>	<p>Law 5:</p> <pre>u <*> pure y = pure (\$y) <*> 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>Monoid: An associative binary function and a value that acts as an identity with respect to that function.</p> <p>Examples</p> <ul style="list-style-type: none"> $x * 1$ Identity of Multiplication <code>lst ++ []</code> Identity of Concatenation $x + 0$ Identity of Addition 	<p>Definition of Monoid Typeclass</p> <pre>class Monoid m where mempty :: m mappend :: m -> m -> m mconcat :: [m] -> m mconcat = foldr mappend mempty</pre>	
---	---	--

Monoid Rules

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

Lecture #10 – Monads

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

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

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

<p>Example: Implement <code>applyMaybe</code> that applies a function to a <code>Maybe</code></p> <pre> applyMaybe :: Maybe a -> (a -> Maybe b) -> (Maybe b) applyMaybe Nothing _ = Nothing applyMaybe (Just x) f = Just (f x) </pre>	<p>Chaining <code>applyMaybe</code></p> <pre> > (Just 3) \applyMaybe (\x -> Just (x*2)) \applyMaybe (\y -> Just (y-1)) Just 5 > (Just 3) \applyMaybe (_ -> Nothing) \applyMaybe (\y -> Just (y-1)) Nothing </pre>	<p>Additional Names for Monoids</p> <ul style="list-style-type: none"> • “Programmable Semicolons” • “Applicative Functors you can chain.”
---	--	---

<p>Monad Typeclass Definition</p> <pre> class Monad m where return :: a -> m a (>=) :: m a -> (a -> m b) -> m b (>>) :: m a -> m b -> m b x >> y = x >= (_ -> y) --Lamda fail :: String -> m a fail msg = error msg </pre>	<p>Example a Robot Moving Towards a Goal (Not Failure)</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) > start -: up -: right (1, 1) > start -: up -: left -: left -: right -: down (-1, 0) </pre>
--	--

<p>Maybe as an Instance of the Monad Typeclass</p> <pre> instance Monad Maybe where return = Just (>=) Nothing _ = Nothing (>=) (Just x) f = f x fail _ = Nothing </pre>	<p>Example a Robot Moving Towards a Goal (with Failure)</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 -> 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) > return start >= up >= left >= left >= right >= down Just (-1, 0) > return start >= left >= left >= up >= up >= right >= up >= right >= right >= down Nothing Explanation: Reached one of the goals (-1, 3) at the red up </pre>
--	---	---

Integer Division Using Monads

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

List Monad

<p>Making List an Instance of Monad</p> <pre> instance Monad [] where return x = [x] (>=) xs f = concat(map f xs) fail _ = [] </pre> <p>Explnation: <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>Example Use of List as a Monad</p> <pre> listOfTuples :: [(Int, Char)] listOfTuples = do n <- [1, 2] ch <- ['a', 'b'] return (n, ch) > listOfTuples [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')] </pre>	<p>Combining a Maybe and a List Monad</p> <pre> > Just [2,3] >= (\x -> Just(fmap (+1) x)) [3, 4] </pre>
---	---	--

Lecture #11 – Parsing Combinators

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

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

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

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

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

Practice Midterm and Review Notes

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

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

CSV Parser Example

Verbose Approach

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

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

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

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

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

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

main = do
    args <- getArgs
    p <- parseFromFile csvFile "example 1" (head args)
    case p of
        Left msg -> error msg
        Right csv -> print csv
```

Concise Approach

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

csvFile = lines `sepBy` eol
line = cells `sepBy` string ","
cells = many (noneOf "\n")
eol = try (string "\n")
    <|> string "\n\r"
    <?> "end of line"

main = do
    args <- getArgs
    p <- parseFromFile csvFile "example 1" (head args)
    case p of
        Left msg -> error msg
        Right csv -> print csv
```

Miscellaneous

<p>Kind of Show and show</p> <pre>> :k Show Show :: * -> Constraint</pre> <p>Type and Kind of show</p> <pre>> :k show Error (A function not a type) > :t show show :: (Show a) => a -> String</pre>	<p>Lambda and ADT Combined</p> <pre>> (\x -> Just (x+1)) 1 Just 2</pre> <p>Creating Type Alias</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>Example: <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) -> (a -> Maybe b) -> (Maybe b) applyMaybe Nothing _ = Nothing applyMaybe (Just x) f = f x</pre> <p>Explanation: 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>Applying return to Items</p> <pre>> return 7 7 > return 7 :: Maybe Int Just 7 > return 7 :: [Int] [7] -- Need Int or get an error</pre> <p>Conclusion: Behavior for <code>return</code> is the same as <code>pure</code>. Both put the object in the minimum default context that still yields that value.</p>	<p>List comprehension is syntactic sugar for using lists as monads.</p>	
--	--	--

<p>Monads and Lambda</p> <p>When trying to chain multiple functions together in a <code>Monad</code>, remember the <code>Monad</code> must return a boxed value. Hence, <code>Lambda</code> often work well as they simplifying boxing.</p>	<p>Applicative Typeclass – Allows you to use normal functions on values that have a context (i.e. are inside a <code>Functor</code>).</p>	<p>return – <code>Monad</code> equivalent of “pure” for <code>Applicative Functors</code>.</p> <p>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.</p>
	<p>Monad: 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>(>>=) :: (Monad m) => m a -> (a -> m b) -> m b</pre> <p>Monads are just applicative functors that support <code>bind (>>=)</code>.</p> <p>Key Difference: <code>Applicative</code> functors support normal functions that take and return unboxed values while <code>Monads</code> return boxed values.</p>	

Functor Definitions

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

Applicative Functor Definitions

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

Monad Definitions

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

Lecture #12 – Introduction to JavaScript

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

Example: Functional JavaScript <pre>function addList(list){ if(list.length == 0){ return 0; } return list[0] + addList(list.slice(1)); }</pre> <p>slice(begin[, end]) – 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>	Example: Object-Oriented JavaScript <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>Adder – Name of a new constructor. Convention is to start constructors with a capital letter.</p> <p>this – Not optional in JavaScript.</p>	Example: Functional JavaScript <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>
--	--	---

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

Reading from a File with Callbacks in Node.js <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. require – Includes the JavaScript package “fs”</p>	Synchronous File IO in Node <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>	Creating a JavaScript Object <pre>var myDog = {age : 3, weight: 100}</pre> <p>Every object is a map.</p> <p>Adding a Field to a JavaScript Object</p> <pre>myDog['height'] = 45 // Add a new height field // Note the single quotes</pre> <p>Adding a Function to a JavaScript Object’s Prototype</p> <pre>myDog.speak = function(){ console.log(“Grr”); }</pre> <p>Delete a Function from a JavaScript Object’s Prototype</p> <pre>delete myDog.speak</pre>
	Undeclared Object Fields <p>Any undeclared object fields or uninstantiated variables are undefined.</p> <pre>var y; // Uninstantiated // Both print ‘undefined’ console.log(y) console.log(myDog.name)</pre>	

Prototypes

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

<p>Add a Special "speak" Method to Rex</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>Effect of the "new" Keyword</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>Unspecified Function Arguments: In JavaScript, any unspecified function argument defaults to "undefined".</p>	<p>No "return" in a Function</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>Top Prototypes</p> <p>Object.prototype – Top of all object prototypes</p> <p>Function.prototype – Top of all function prototypes.</p>	<p>Iterating Using "forEach"</p> <pre> var arr = [1, 2, 3]; // Print each element in array arr.forEach(function(val) { console.log(val); }); </pre>	<p>require</p> <ul style="list-style-type: none"> Used to import an external module in Node.js Can be stored in a variable. Example: <pre>var net = require('net');</pre>	<p>Running from the Command Line</p> <ul style="list-style-type: none"> Use the keyword "node" for Node.js. <p>Example:</p> <pre>\$ node my_program.js</pre>
---	--	--	--

<p>Example: Create an Object with a Constructor</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>Example: Currying in JavaScript</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	
$e ::= x$ (Variables, immutable) $\mid \lambda x. e$ (Lambda abstraction) $\mid e e$ (Function application) Note: Lambda (λ) is simply a function. $v ::= (\lambda x. e)$ (Lambda abstraction)	Given a function where λ is a complex expression : $\lambda(x. E)v$ Then: $\lambda(x. E)v \rightarrow E[x \mapsto v]$ Hence, “ v ” replaces “ x ” in “ E ”.	

Small-Step Evaluation Order Rules for Lambda Calculus

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

Evaluation Strategies

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

Lecture #14 – JavaScript Scoping

<p>Example: First Class Function</p> <pre>function makeAdder(x) { return function(y) { return x + y; }; } var addOne = makeAdder(1); // Prints "11" console.log(addOne(10));</pre>	<p>Example: Function Application</p> <pre>function makeAdderList(arr) { var i; var output = []; for(i = 0; i < 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>Block Scope – 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"> Unlike most languages, JavaScript does not have block scope. To create a new scope, use an anonymous function. <p>Variable Hoisting – All variable declarations (i.e., use of “var”) are treated as if they are at the beginning of the function.</p>	<p>“this” in JavaScript</p> <p>this – Refers to the scope where the function is called.</p> <ul style="list-style-type: none"> In Normal Function Calls – this refers to the global “this” Object Methods – The object itself. Constructor (using “new”) – The newly created object. Exceptions: apply, call, and bind. Inline event handles on DOM elements <p>Any time a new function is created, the other “this” is no longer in scope</p>
---	--	--	--

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

<p>apply, bind, call Example</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"> apply – Calls a function with the arguments passed as an array. call – Calls a function with the arguments passed in comma separated. bind – Used to create a new function with a custom context. 	
--	--	--

Lecture #14.5 – JSLint and TypeScript

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

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

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

TypeScript Class <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>	
---	--

Lecture #15 – Event-Based Programming and Cryptocurrencies

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

<p>Removing an Event Listener</p> <ul style="list-style-type: none"> Event listeners can be removed by function name. <p>Example:</p> <pre>btn.removeEventListener('click', sayGroovy);</pre>	<p>Event Emitter</p> <ul style="list-style-type: none"> Import the "events" module using the syntax <pre>var ee = require('events').EventEmitter;</pre> <ul style="list-style-type: none"> Used to create event via the keyword "on". <p>Example:</p> <pre>ee.on('die', function(){ console.log("Died"); });</pre> <ul style="list-style-type: none"> Invoking (emitting) an event using the keyword "emit" Example: <pre>setTimeout(function(){ ee.emit('die'); }, 100); // in ms</pre>	<p>Create a TCP Server in Node.js Using Event Listeners</p> <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>
<p>Events in JavaScript</p> <ul style="list-style-type: none"> JavaScript is single threaded. <p>An event must be run to completion before the next event handler can run.</p>		

Cryptocurrencies

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

<p>Example: JavaScript Signer Example</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>Double Spending – Spend the same funds in multiple places.</p> <p>Solutions to Prevent Double Spending:</p> <ul style="list-style-type: none"> Centralized Authority – Disadvantages include that the central authority would charge a fee and not everyone trusts central authorities. Decentralized Authority – Broadcast transactions to everyone. <p>Ledger – Used to keep a history of all transactions 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

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

Lecture #17 – Macros and Sweet.js

<p>Macros</p> <ul style="list-style-type: none"> Short for “<i>macroinstruction</i>” Rule specifies how an input sequence maps to a replacement sequence. 	<p>Example: C Preprocessor Example</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>Basic Compiler Structure with C-Style Macros</p> <pre> graph LR SC[Source Code] --> PP[Pre-Processor] PP --> EC[Expanded Code] EC --> LT[Lexer/Tokenizer] LT --> T[Tokens] T --> P[Parser] P --> AST[Abstract Syntax Tree] AST --> C[Compiler] C --> MC[Machine Code] AST --> I[Interpreter] I --> IP[Interpreter] </pre>
<p>Macros in C</p> <ul style="list-style-type: none"> Performed by a <i>preprocessor</i> Rely on text substitution. Embedded languages like PHP, Ruby, etc. use a similar approach. 	<p>C Preprocessor Output</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>Problem with C Macros (Input)</p> <pre>// Macro should be on one line #define SWAP(a,b) {int tmp=a; a=b; b=tmp;}</pre> <pre>int main(void){ int x = 4, tmp = 5; SWAP(x,tmp) }</pre>	<p>Hygienic Macro – Any macro whose expansion is guaranteed not to cause the accidental capture of identifiers.</p>	<p>Macros in JavaScript</p> <ul style="list-style-type: none"> No standard macro system for JavaScript Sweet.js has been gaining interest. Recently redesigned.
<p>Problem with C Macros (Output)</p> <pre>int main(void){ int x = 4, tmp = 5; { int tmp = x; a = tmp; tmp = tmp; } }</pre> <p>Hence, a variable name collision between the two variables named “<i>tmp</i>”. This is known as “<i>inadvertent variable capture</i>”</p>	<p>Syntactic Macros</p> <ul style="list-style-type: none"> Derive from Lisp since Lisp programs are essentially one big AST. Work at the level of abstract syntax trees. Powerful by expensive. Hygiene easier to address at the AST level. Essentially a source-to-source compiler. 	<p>Sweet.js</p> <ul style="list-style-type: none"> Borrows concepts from Racket. Source-to-source compiler (i.e., transpiler) for JavaScript. Examples of other JavaScript transpilers: <ul style="list-style-type: none"> TypeScript CoffeeScript Dart (includes its own VM) Project backed by Mozilla
	<p>Basic Compiler Structure with Syntactic Macros</p> <pre> graph LR AST1[Abstract Syntax Tree] --> ME[Macro Expander] ME --> AST2[Abstract Syntax Tree] </pre>	<p>Invoking Sweet.js</p> <ul style="list-style-type: none"> From command line: \$ sjs myfile.js -d out/ Compiled files run normally (as shown below for Node): \$ node out/myfile.js

<p>Writing a Swap Function in Sweet.js</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>Note #1: The returned string is preceded by a pound (#) sign and is enclosed in backticks (`).</p> <p>Note #2: Sweet.js variables are declared with “<i>let</i>”.</p>	<p>Concatenating Multiple Result Strings</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>Note #1: Use “<i>.concat</i>” to concatenate multiple result strings.</p> <p>Note #1: If a token is not present, “<i>.next()</i>” does not cause an error.</p>	<p>Keywords in Sweet.js</p> <ul style="list-style-type: none"> let – Create a Sweet.js variable. ctx.next().value() – Get the next value from the context. #`...` – Used to define a result string. concat – Used to combine two result strings. let xxx of yyy – Iterate over a list of tokens. isIdentifier – Used to check if a Sweet.js variable matches some string.
---	--	--

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

Lecture #19 – Metaprogramming and JS Proxies

<p>Metaprogramming: Writing programs that manipulate other programs.</p> <ul style="list-style-type: none">Proposed in ECMAScript 6 for JavaScript. <p>Terminology in Reflection</p> <ul style="list-style-type: none">Introspection: Ability to examine (but not modify) the structure of a program.Self-modification: Ability to modify the structure of a program.	<p>Introspection</p> <p>Ability to examine (but not modify) the structure of a program.</p> <p>JavaScript Examples</p> <p>Property Lookup</p> <pre>"x" in o; //o is an object</pre> <p>Iterate Over All Properties of an Object<pre>for(prop in o){ // Do something ... }</pre></p>	<p>Self-modification</p> <p>Ability to modify the structure of a program.</p> <p>JavaScript Examples</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>	<p>Proxies in JavaScript</p> <ul style="list-style-type: none">Metacircular Interpretation – The language is able to understand its own language.Until recently, JavaScript did not support intercession.<ul style="list-style-type: none">Javascript proxies are intended to fix that.Node.js’ implementation of proxies lags behind the standard.Proxies only exist for objects and functions.<ul style="list-style-type: none">Proxies do not exist for primitives.
<p>Proxies and Common Lisp</p> <ul style="list-style-type: none">Developed before object oriented languages were popular.Many libraries were create with non-standard OO systems.Common Lisp Object System (CLOS) – Standard object oriented system for Lisp.	<p>Achieving Lisp Object Backwards Compatibility</p> <p>Option #1: Rewrite all libraries using CLOS. Disadvantage</p> <ul style="list-style-type: none">Huge number of libraries.Not feasible to rewrite them all. <p>Option #2: Make a complex API</p> <ul style="list-style-type: none">API difficult to understand.Systems had conflicting features. <p>Option #3: Keep API simple and modify object behavior to fit different systems.</p> <ul style="list-style-type: none">This approach relies on metaobject protocols.	<p>Proxies and Handlers</p> <ul style="list-style-type: none">The behavior of a proxy is determined by traps specified in its handler (i.e., the metaobject).Trap – Methods that intercept an operation.Handler – The metaobject that specifies the details of the trap. The handler itself is usually a normal object.Using proxies in node requires a special flag “--harmony-proxies”. Example: \$ node --harmony-proxies prog.js	<p>Kinds of JavaScript Proxies</p> <ul style="list-style-type: none">Object Proxies – Defined with: Proxy.create(handler, proto)Functions (with extra traps) - Defined with: Proxy.createFunction(handler, callTrap, constructTrap)Proxies do not exist for primitives.
<p>A Simple Proxy</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>A Noop Proxy – All Operations Passed through Unchanged</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.prototype .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>Aspect Oriented Programming</p> <ul style="list-style-type: none">Some code not well organized into objects. Example:<ul style="list-style-type: none">Cross-cutting concern where code is spread throughout a program.Canonical Example: Logging Statements<ul style="list-style-type: none">Littered throughout the codeSwapping out a logger requires massive code changes.Solution: Use a proxy	

Lecture #20 – Introduction to Ruby

<h3>Influences of Ruby</h3> <ul style="list-style-type: none">• SmallTalk<ul style="list-style-type: none">○ Everything is an object○ Blocks○ Metaprogramming• Perl<ul style="list-style-type: none">○ Regular Expressions○ Function names	<h3>Basic Ruby Syntax</h3> <pre>puts "Hello World" a = [1, 2, 3] m = { 'a' => "Apple", 'b' => "Bear", 'c' => "Cat" } # Prints "1" puts a[0] # Prints "Apple" puts a['m']</pre>	<h3>Basic Ruby Class</h3> <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>	<h3>Using Metaprogramming for Getters and Setters</h3> <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>
<h3>Ruby on Rails</h3> <ul style="list-style-type: none">• “Killer” app for Ruby<ul style="list-style-type: none">○ Lightweight web framework○ “Convention over configuration” – If use standard configuration, very little configuration required.• Initial framework was PHP, but that was abandoned.	<h3>Keywords</h3> <p>@ - Represents an object property</p> <h3>Returning From a Function</h3> <ul style="list-style-type: none">• Every function in Ruby returns a value, even if return is not used.• If no return is specified, a function returns the last used value.		<h3>Using a Class in Ruby</h3> <pre>p = Person.new "Joe" puts "Name is #{p.name}" p.say_hi #{...} – Embeds a variable in a Ruby String</pre>

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

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

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

Example: <code>class_eval</code> in Ruby <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>	Example: Using Regular Expressions in Ruby <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

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

Lecture #21 – Blocks and Messages

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

<p>Example: Using a Singleton Class to Create Static Methods</p> <pre> # Add Static Methods to Employee Class class Employee class << 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 end </pre>	<p>Message Passing</p> <ul style="list-style-type: none"> Represents inter-object interaction. Sender Sends: <ul style="list-style-type: none"> Method name Data: Method parameters (if any) Receiver: <ul style="list-style-type: none"> Processes the message (<i>Optionally</i>) returns data Receiver may not understand the message. 	<p>Example: <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>method_missing</p> <ul style="list-style-type: none"> Method that is part of every class. Can be overridden. <ul style="list-style-type: none"> Smalltalk Name: <code>doesNotUnderstand</code> Ruby Name: <code>method_missing</code> Invoked whenever an unknown method is called. 	<p>Active Record and Message Passing</p> <ul style="list-style-type: none"> Relational database tool in Ruby. Specify fields in the database to be extracted based off method names. Example: <pre> Person.find_by_first_name "John" </pre>

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

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