

CS252 – Midterm Exam Study Guide

By: Zayd Hammoudeh

Lecture #01 – General Introduction

Reasons for Different Programming Languages		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 	Programming Language Design Choices <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 	<ol style="list-style-type: none"> 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-code “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"> ◦ An expression 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: <code>> let f x = x + 1</code> <code>> f 3</code> <code>4</code>	Run Haskell from Command Line Use <code>runhaskell</code> keyword. Example: <code>> runhaskell <FileName>.hs</code>	Hello World in Haskell <pre>main :: IO () main = do putStrLn "Hello World"</pre>
--	--	---	--

Primitive Classes in Haskell	Lists		Ranges
	<ul style="list-style-type: none"> • Base 0 • Comma separated in square brackets • Operators <ul style="list-style-type: none"> ◦ <code>:</code> Prepend ◦ <code>++</code> Concatenate ◦ <code>!!</code> Get element a specific index ◦ <code>head</code> First element in list ◦ <code>tail</code> All elements after head 	<ul style="list-style-type: none"> ◦ <code>last</code> Last element in the list ◦ <code>init</code> All elements in the list except the last one ◦ <code>take n</code> Take first n elements from a list ◦ <code>replicate l m</code> Create a list of length l containing only m ◦ <code>repeat m</code> Create an infinite list containing only m 	<ul style="list-style-type: none"> • Can be infinite or bounded • Use the “<code>..</code>” notation. Examples: <code>> [1..4]</code> <code>[1, 2, 3, 4]</code> <code>> [1,2..6]</code> <code>[1, 2, 3, 4, 5, 6]</code> <code>> [1,3..10]</code> <code>[1, 3, 5, 7, 9]</code> <code>> [5, 4..1]</code> <code>[5, 4, 3, 2, 1]</code>
Hello World in Haskell <pre>main :: IO () main = do putStrLn "Hello World"</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> <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>A Simple Function</p> <pre>> let inc x = x + 1 > inc 3 4 > inc 4.5 5.5 > inc (-5) -- Negative -4</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>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>
--	---	---

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

Lecture #03 – Operational Semantics

<p>Formal Semantics</p> <p>Crisply define how the language features work.</p> <p>Abstract Syntax Tree</p> <p>Tree representation of the abstract syntactic structure of a program's source code. Example is Bool* language below.</p> <p>Bool* Language</p> <div> <pre>e ::= true false if e then e else e</pre> <p>Expressions:</p> <p>constant true</p> <p>constant false</p> <p>conditional</p> </div> <div> <pre>v ::= true false</pre> <p>Values:</p> <p>constant true</p> <p>constant false</p> </div>	<p>Formal Semantic Styles</p> <ul style="list-style-type: none"> Operational <ul style="list-style-type: none"> Big-Step ("Natural") Small-Step ("structural") Axiomatic Denotational <p>Big Step Operational Semantics</p> <ul style="list-style-type: none"> Evaluates every expression to a value <p>↓ : "Evaluates to" symbol in Big-Step operational semantics.</p> <p>Example Formatting:</p> $e \Downarrow v$ <ul style="list-style-type: none"> Read as: "Expression e evaluates to the value v" 	<p>A Review of Compilers</p>
---	--	-------------------------------------

<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: $e \rightarrow e' \rightarrow e'' \rightarrow v$ <ul style="list-style-type: none"> →* : 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</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

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

Lambda	Function Composition	Point-Free Style	Example: Lambda with Function Composition
<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>	<ul style="list-style-type: none"> Uses the period (.) $f(g(x))$ can be rewritten $(f \cdot g) x$ 	<ul style="list-style-type: none"> Pass function arguments no arguments. Example: <pre>> let inc = (+1) -- No args > inc 3 4</pre>	<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>

Iterative vs. Recursive	Not Tail Recursive	Tail Recursive Factorial
<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>	<pre>public int factorial(int n) { if (n==1) return 1; else { return n * factorial(n-1); } }</pre> <p>Last step is the multiplication so not tail recursive.</p>	<pre>public int factorialAcc(int n, int acc) { if (n==1) return acc; else { return factorialAcc(n-1, n*acc); } }</pre> <p>Tail recursive code often uses the accumulator pattern like above.</p>

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

Higher Order Functions

Functions in Functional Programming	Qualities of Functional Programming	Higher Order Function	
<ul style="list-style-type: none"> Functional languages treat programs as mathematical functions. Mathematical Definition of a Function: A function f is a rule that associates to each x from some set X of values a unique y from a set of Y 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 	<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>Any function that takes a function as a parameter or returns a function as a result.</p> <p>Function Currying 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. 	

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> $\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3$ <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> <p>if true then (if true then false else false) else true</p> <p>This reduces to "if true then false else false"</p>	<p>Example: Not a Redex</p> <p>if (if true then false else false) then true else true</p> <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> $C[r] = \text{if (if true then false else false) then true else true}$ $r = \text{if true then false else false}$ $C = \text{if } \bullet \text{ then true else true}$ <p>C[r] is the original expression.</p>	<p>Rewriting Evaluation Order Rules</p> <p>Context based rules only apply to reducible expressions (redexes). Example:</p> <p>EC-IfFalse:</p> $C[\text{if false then } e_2 \text{ else } e_3] \rightarrow C[e_3]$ <p>Context Syntax</p> $C ::= \bullet$ $ \text{if } C \text{ then } e \text{ else } e$ $ C \text{ op } e$ $ v \text{ op } C$ $ \dots$	<p>Data.Map</p> <ul style="list-style-type: none"> Library: Data.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.
---	--	--

<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

<p>TeX</p> <ul style="list-style-type: none"> Created by Donald Knuth Precisely controls the interface of content. Type of Literate Programming – Logic is in natural language and code is interspersed. 	<p>LaTeX</p> <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 	<p>Specify Document Type</p> <pre>\documentclass{article}</pre> <p>Specify Title Block Content</p> <pre>\title{Hello World!}</pre> <p>Start Document</p> <pre>\begin{document}</pre> <p>Generate Title from Title Information</p> <pre>\title{Hello World!}</pre> <p>Close the Document</p> <pre>\end{document}</pre>	<p>Cross-Reference</p> <pre>\ref{<referenceName>}</pre> <p>Reference a Bibliography Citation</p> <pre>\cite{<citationName>}</pre> <p>Create a Reference</p> <pre>\label{<referenceName>}</pre> <p>Create a Bibliography</p> <pre>\bibliography{<bibFileName>}</pre> <p>Create a List</p> <pre>\begin{itemize} \item Text for #1 \item Text for #2 \end{itemize}</pre>
---	--	--	--

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

Lecture #07 – Types and Typeclasses

<p>Maybe Type</p> <ul style="list-style-type: none"> Example of an algebraic data type Enables behavior similar to null in Java 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>	<p>Maybe "Divide" Example</p> <pre>divide :: Int -> Int -> Maybe Int divide _ 0 = Nothing divide x y = Just \$ x `div` y</pre> <pre>> divide 5 2 2 > divide 4 0 Nothing</pre> <p>DO NOT FORGET THE Just IN CORRECT SOLUTION</p>	<p>Maybe Map Example</p> <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>
--	---	--

<p>Algebraic Data Type</p> <ul style="list-style-type: none"> A composite data type (i.e. a type made from other types). Created via the Keyword: data Examples: Either, Maybe, Tree 	<p>Example Algebraic Data Type</p> <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>
	<p>Partially Applying a Value Constructor</p> <ul style="list-style-type: none"> Value constructors can be partially applied similar to functions. <p>Example:</p> <pre>> let leaf = Node EmptyTree EmptyTree > Node (leaf 3) (leaf 7) 5</pre> <p>This creates a three node tree with value 5 at the root and values 3 and 7 at the leaves.</p>	<p>Type of the "+" Operator</p> <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> <p>Type of a Number</p> <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>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 = Just (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: Lex/Flex (C/C++). ANTLR & JavaCC (Java). Parsec (Haskell)</p>
--	---	---

<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> • 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` eof
line = cells `sepBy` string ","
cells = many (noneOf "\n")
eof = 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 > 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>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>	<p><code>return</code> – <code>Monad</code> equivalent of “pure” for <code>Applicative</code> <code>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>