CS252 – Final Exam Study Guide

By: Zayd Hammoudeh

Lecture #01 – General Introduction

Reasons for Different Programming Languages
1. Different domains (e.g. web, security, bioinformatics)
2. Legacy code and libraries
3. Personal preference

Programming Language Design Choices
1. Flexibility
2. Type safety
3. Performance
4. Build Time
5. Concurrency

Features of Good Programming Languages
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
- 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
- Multi-core "explosion"
- Big Data
- Mobile Devices

Advantages of Web and Scripting Languages
- 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
- Garbage collection
- *Sound* type systems
- Concurrency tools
- Closures

Programs that Manipulate Other Programs
- Compilers & interpreters
- JavaScript rewriting
- Instrumentation
- Program Analyzers
- IDEs

Formal Semantics
- Used to share information *unambiguously*
- Can formally prove a language supports a given property
- ***Crisply define*** *how a language works*

Types of Formal Semantics
- Operational
  - Big Step "*natural*"
  - Small Step "*structural*"
- Axiomatic
- Denotational

Haskell

- Purely functional – Define "*what stuff is*"
- No side effects
- Referential transparency – A function with the same input parameters will always have the same result.
  - 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.
- More flexible but less safe.
- Supported by Haskell
- Common in scripting languages (e.g. Python, Ruby)

Side Effects in Haskell
- Generally not supported.
- Example of Support Side Effects: File IO
- Functions that do have side effects must be separated from other functions.

Lazy Evaluation
- 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
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:

```
> let f x = x + 1
> f 3
4
```

Run Haskell from Command Line

Use `runhaskell` keyword.

Example:

```
> runhaskell <FileName>.hs
```

Hello World in Haskell

```
main :: IO ()
main = do
  putStrLn "Hello World"
```

Primitive Classes in Haskell
1. `Int` – Bounded `Integers`
2. `Integer` – Unbounded
3. `Float`
4. `Double`
5. `Bool`
6. `Char`

Lists
- Base 0
- Comma separated in square brackets
- Operators
  - `:` Prepend
  - `++` Concatenate
  - `!!` Get element a specific index
  - `head` First element in list
  - `tail` All elements after head
  - `last` Last element in the list
  - `init` All elements in the list except the last one
  - `take n` Take first n elements from a list
  - `replicate l m` Create a list of length `l` containing only `m`
  - `repeat m` Create an infinite list containing only `m`

Ranges
- Can be infinite or bounded
- Use the "`..`" notation. Examples:

```
> [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]
```

Hello World in Haskell

```
main :: IO ()
main = do
  putStrLn "Hello World"
```

List Examples

```
> putStrLn $ "Hello " ++ "World"
"Hello World"

> let s = bra in s !! 2 : s ++ 'c' : last s : 'd' : s
"abracadabra"
```

Infinite List Example

```
> let even = [2,4..]
> take 5 even
    [2, 4, 6, 8, 10]
```

List Comprehension

- 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:

```
> [ 2*x | x <- [1..5]]
[2, 4, 6, 8, 10]
```

```
> [(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)]
```

A Simple Function

```
> let inc x = x + 1
> inc 3
4

> inc 4.5
5.5

> inc (-5) -- Negative
-4
```

Type Signature

- Uses symbols "::" and "->"
- Example:

```
inc :: Int -> Int
inc x = x + 1
```

Pattern Matching

- Used to handle different input data
- Guard uses the pipe (|) operator
- Example:

```
inc :: Int -> Int
inc x
  | x < 0 = error "invalid x"
inc x = x + 1
```

Recursion

- Base Case – Says when recursion should stop.
- Recursive Step – Calls the function with a *smaller version* of the problem

Example:

```
addNum :: [Int] -> Int
addNum [] = 0
addNum (x:xs) = x + addNum xs
```

Lab #01 – Max Number

```
> maxNum :: [Int] -> Int
> maxNum [] = error "Invalid Input"
> maxNum [x] = x
> maxNum (x:xs) = if x > maxXs then x else maxXs
>   where maxXs = maxNum xs
```

Reasons for a Large Number of Programming Languages

- Different domains
- Different design choices

Recursion

- `:t` or `:type` – Gets the type of a variable or function.

Example:

```
> :type 'A'
'A' :: Char
> :t "Hello"
"Hello" :: [Char]
```

Haskell's Base Typeclasses

- `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

Formal Semantics

Crisply define how the language features work.

Abstract Syntax Tree

Tree representation of the abstract syntactic structure of a program's source code. Example is Bool* language below.

Bool * Language

`e ::=`
`    true`
`  | false`
`  | if e`
`     then e`
`     else e`
	Expressions:
	constant true
	constant false
	conditional

`v ::=`
`    true`
`  | false`
	Values:
	constant true
	constant false

Formal Semantic Styles

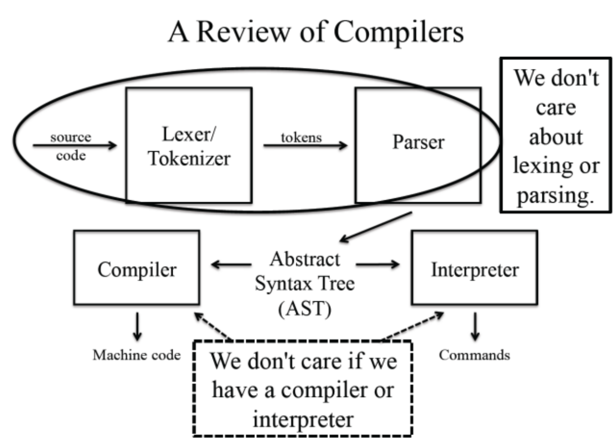
- Operational – Specify how expressions should be evaluated.
  - Big-Step ("Natural")
  - Small-Step ("structural")
- Axiomatic
- Denotational

Big Step Operational Semantics

- Evaluates every expression to a value
- $\Downarrow$: "*Evaluates to*" symbol in Big-Step operational semantics.
- Example Formatting:

$$e \Downarrow v$$

- Read as: "Expression e *evaluates* to the value v"



Small-Step Operational Semantics

- Evaluate an expression until it is in ***normal form***
- Normal Form – Any form that cannot be evaluated further.
- $\rightarrow$: "*Evaluates to*" symbol in small step operational semantics. Example:

$$e \rightarrow e' \rightarrow e'' \rightarrow v$$

- $\rightarrow^*$: Many evaluation steps required. Example:

$$e \rightarrow^* v$$

Bool* Small-Step Operational Semantics Rules

E-IfTrue:

$$\frac{}{\text{if true then } e_2 \text{ else } e_3 \rightarrow e_2}$$

E-IfFalse:

$$\frac{}{\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3}$$

E-If:

$$\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$

Example: Reduce the expression
if (if true then false else true) then true else false

Step #1: Use rule "E-IfTrue" with "E-If"

if false then true else false

Step #2: Use rule "E-IfFalse" (Now in normal form)

false

Bool* Extension: Numbers

- 0 : The Number "0"
- succ 0 : Represents "1"
- succ succ 0 : Represents "2"
- pred n : Gets the predecessor of "n"

Extended Bool * Language

```
e ::=
    true
  | false
  | if e then e else e
  | 0
  | succ e
  | pred e
v ::= true | false
    | IntV

IntV ::= 0 | succ IntV
```

Literate Haskell

- File Extension: ".lhs"
- Code lines begin with ">"
- All other lines are comments.
- "Essentially swaps code with comments."

Case Statement in Haskell

- Keywords: case, of, otherwise
- Operator: ->

Example:

```
case x of
    val1 -> "Value 1"
    val2 -> "Value 2"
    otherwise -> "Everything else."
```

Lab #02 Review

Bool Expression Type

```
> data BoolExp = BTrue
>       | BFalse
>       | Bif BoolExp BoolExp BoolExp
>       | B0
>       | Bsucc BoolExp
>       | Bpred BoolExp
>   deriving Show
```

BoolVal Type

```
> data BoolVal = BVTrue
>              | BVFalse
>              | BVNum BVInt
>   deriving Show

> data BVInt = BV0
>            | BVSucc BVInt
>   deriving Show
```

Type Constructors: BoolExp, BoolVal, BVInt

***Non-nullary* Value Constructors:** BIf, Bsucc, Bpred, BVSucc, BVNum

Note: Even constants like B0, BTrue, BFalse, BVTrue, and BVFalse are nullary value constructors (since they take no arguments)

Lecture #04 – Higher Order Functions

Lambda

- Analogous to anonymous classes in Java.
- Based off Lambda calculus
- Example:

```
> (\x -> x + 1) 1
2
>(\x y -> x + y) 2 3
5
```

Function Composition

- Uses the period (.)
- `f(g(x))` can be rewritten `(f . g) x`

Point-Free Style

- Pass no arguments to a function
- Example:

```
> let inc = (+1) – No args
> inc 3
4
```

Example: Lambda with Function Composition

```
> let f = (\x -> x – 5)
        . (\y -> y * 2)
> f 7
9

> let f = (\x y -> x – y)
        . (\z -> z * (-1))
> f 3 4
-7
```

Iterative vs. Recursive

- Iterative tends to be more efficient than recursive.
- Compiler can optimize tail recursive function.

Tail Recursive Function – The recursive call is the last step performed before returning a value.

Not Tail Recursive

```
public int factorial(int n) {
  if (n==1) return 1;
  else {
    return n * factorial(n-1);
  }
}
```

Last step is the multiplication so not tail recursive.

Tail Recursive Factorial

```
public int factorialAcc(int n, int acc)
{
  if (n==1) return acc;
  else {
    return factorialAcc(n-1, n*acc);
  }
}
```

Tail recursive code often uses the accumulator pattern like above.

Tail Recursion in Haskell

```
fact' :: Int -> Int -> Int
fact' 0 acc = acc
fact' n acc = fact' (n - 1) (n * acc)
```

Higher Order Functions

Functions in Functional Programming

- 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.

$$(x \in X \wedge y \in Y) \rightarrow y = f(x)$$

- f – Name of the function
- x – Independent variable
- y – Dependent variable
- X – Domain
- Y – Range

Qualities of Functional Programming

- Functions clearly distinguish:
  - 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:
  - Passed as arguments to a function
  - Be returned from a function
  - Construct new functions dynamically

Higher Order Function

Any function that takes a function as a parameter *or* returns a function as a result.

Function Currying

Transform a function with multiple arguments into multiple functions that each take exactly one argument.

Named after Haskell Brooks Curry.

Currying Example

```
addNums :: Num a => a -> a -> a
```

addNums is a function that takes in a number and returns a function that takes in another number.

map

- Built in Haskell higher order function
- Applies a function to all elements of a list.

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

> map (+1) [1, 2, 3]
[2, 3, 4]
```

filter

- Built in Haskell higher order function
- Removes all elements from a list that do not satisfy (i.e. make true) some predicate.

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

> filter (>2) [1, 2, 3, 4]
[3, 4]
```

foldl

- Built in higher order function
- Does not support infinite lists.
- Should only be used for special cases.

```
foldl :: (b -> a -> b) -> b -> a -> b
```

Example:

```
> foldl (\x y -> x - y) 0 [1, 2, 3, 4]
-10 -- (((0-1) - 2) - 3) - 4
```

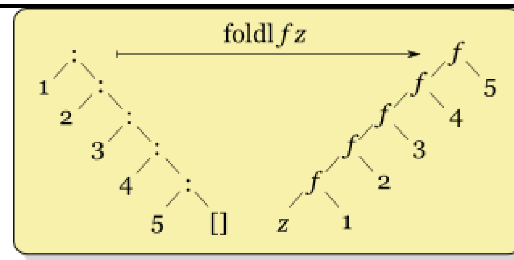
foldr

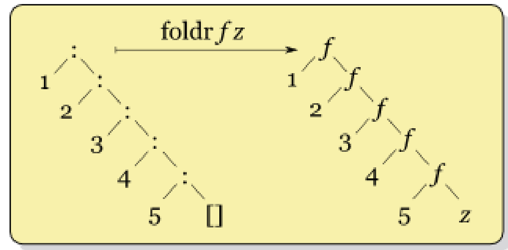
- Built in higher order function
- Supports infinite lists.
- "*Usually the right fold to use*"

```
foldr :: (b -> a -> a) -> a -> b -> a
```

Example:

```
> foldr (\x y -> x + y) 0 [1, 2, 3, 4]
-2 -- 1 – (2 – (3 – (4 – 0)))
```





Thunk – A delayed computation

Due to lazy evaluation, foldl and foldr build thunks rather than calculate the results as they go.

foldl'

- `Data.List.foldl'` evaluates its results eagerly (i.e. does not use thunks)
- Good for large, but finite lists.

`foldl` in terms of `foldr`

```
myFoldl' f acc x = foldr (flip f) acc (reverse x)
```

Lecture #05 – Small-Step Operational Semantics

WHILE Language

- Unlike the Bool* language, WHILE supports mutable references.

`e ::= a`	Variable/addresses
`  | v`	Values
`  | a:=e`	Assignment
`  | e;e`	Sequence
`  | e op e`	Binary Operations
`  | if e then e else e`	Conditional
`  | while (e) e`	While Loops
`v ::= i`	Integers
`  | b`	Boolean
`op ::= + | - | * | / | >= | > | <= | <`	

Small Step Semantics with State

- Since the WHILE language supports mutable references, the grammar must be updated to support it.

While Relation:

$$e, \sigma \rightarrow e', \sigma'$$

- σ – Store. Maps *references* to values.

Example Operations:

- $\sigma(a)$ – 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"

Evaluation Order Rules

- Tend to be repetitive and clutter the semantics.
- Context based rules tend to represent the same information as evaluation order rules but more concisely.

Reduction Rule

Rewrites the expression. Example:

E-IfFalse:

`if false then e2 else e3` → `e3`

Context Rule

Specify the order for evaluating expressions. Example:

E-If:

$$\frac{e_1 \rightarrow e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

Reducible Expression (**Redex**) – Any expression that can be transformed (reduced) in one step.

Example: Redex

if true **then** (**if** true **then** false **else** false) **else** true

This reduces to "**if** true **then** false **else** false"

Example: Not a Redex

if (**if** true **then** false **else** false) **then** true **else** true

Not a redex as expression "**if** true **then** false **else** false" must be evaluated first.

Evaluation Contexts

- Alternative to evaluation order rules.
- Marker (•) / hole indicate the next place for evaluation (i.e. where we will do the work).

Example:

C[r]
= **if** (**if** true **then** false **else** false) **then** true **else** true

r = **if** true **then** false **else** false

C = **if** • **then** true **else** true

C[r] is the original expression.

Rewriting Evaluation Order Rules

Context based rules only apply to reducible expressions (redexes). Example:

EC-IfFalse:

C[if false then e_2 else e_3] → C[e_3]

Context Syntax

```
C ::= •
  | if C then e else e
  | C op e
  | v op C
  | ...
```

Data.Map

- **Library:** import Data.Map as `Map`
- Immutable
- Example Methods:
  - `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.

Precondition – Text above the line in a rule.

Context Rule for Binary Op:

$$\frac{v_3 = v_1 \text{ op } v_2}{C[v_1 \text{ op } v_2] \rightarrow C[v_3]}$$

How to Read a Small Step Semantic Rule: "Given <*Precondition*>, then <*LeftSideArrow*> evaluates to <*RightSideArrow*>."

Lecture #06 – LaTeX

TeX
- 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.** "***Mark code instead of marking comments.***"

LaTeX
- 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**.
  - **Example:** LaTeX separates formatting from content.
- **Literate Programming**

Specify Document Type
```
\documentclass{article}
```
Specify Title Block Content
```
\title{Hello World!}
```
Start Document
```
\begin{document}
```
Generate Title from Title Information
```
\title{Hello World!}
```
Close the Document
```
\end{document}
```

Cross-Reference
```
\ref{<referenceName>}
```
Reference a Bibliography Citation
```
\cite{<citationName>}
```
Create a Reference
```
\label{<referenceName>}
```
Create a Bibliography
```
\bibliography{<bibFileName>}
```
Create a List
```
\begin{itemize}
\item Text for #1
\item Text for #2
\end{itemize}
```

Create Section with Label
```
\section{Section #1}
\label{sec:one}
```
Create Subsection with Label
```
\subsection{<SubsectionName>}
\label{sec:<refName>}
```
Use of Tilde (~)

Creates an undividable space so the text "`Section~\ref{sec:one}`" will appear on one line

BibTeX
- **References are tedious to reformat and renumber.**
- Reference details shorted in a "***.bib**" file.

Create a Bibliography
```
\bibliography{biblio}
```
BibTeX filename for the example would be "**biblio.bib**"

Define Bibliography Style
```
\bibliographystyle{plainurl}
```

BibTeX Article Reference Example
```
@article{citationName,
    author = {Donald Knuth},
    title = {Literate Programming},
    journal = {},
    year = {1984},
    volume = {27},
    number = {2},
    pages = {97–111},
}
```

Lecture #07 – Types and Typeclasses

Maybe Type
- **Example of an algebraic data type**
- Enables behavior similar to **null** in Java
- Can be used to provide context.
- **Used when:**
  - **A function may not return a value**
  - **A caller may not pass an argument**
- **Definition:**

```
data Maybe a = Nothing
             | Just a
```

Maybe "Divide" Example
```
divide :: Int -> Int -> Maybe Int
divide _ 0 = Nothing
divide x y = Just $ x `div` y

> divide 5 2
2
> divide 4 0
Nothing
```
DO NOT FORGET THE `Just` IN CORRECT SOLUTION

Maybe Map Example
```
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
```
Since element may not be in the map, you need to use a maybe

Algebraic Data Type
- A **composite data type** (i.e. **a type made from other types**).
- **Created via the Keyword:** `data`
- **Examples:**
  - `Either`
  - `Maybe`
  - `Tree`

Example Algebraic Data Type
```
data Tree k = EmptyTree
            | Node (Tree k) (Tree k) val
            deriving (Show)
```
`k` – **Type parameter**. **Specifies a type not a value.**

`Node`: **Value Constructor that creates values of type "`Tree k`"**

- **`Tree` and `Tree Int` have no types since they themselves form a concrete type.**
- `Node` does have a type:

```
> :t Node
Node :: (Tree k) -> (Tree k) -> k -> (Tree k)
```
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.

Partially Applying a Value Constructor
- Value constructors can be partially applied similar to functions. **Example:**

```
> let leaf = Node EmptyTree EmptyTree

> Node (leaf 3) (leaf 7) 5
```
This creates a three node tree with value 5 at the root and values 3 and 7 at the leaves.

Type of the "+" Operator
```
> :t (+)
(+) :: (Num a) => a -> a -> a
```
Explanation: The plus sign takes two numbers of type "**a**" and returns an object of type "**a**".

Type of a Number
```
> :t 3
3 :: (Num a) => a
```
Explanation: Since "3" has no explicit type, it can for now be any type that satisfies the "Num" type class.

Kinds		Typeclasses	

• "*The type of types*".
 	
• Concrete types have a kind of "*"
 	
• Keyword `:k`, `:kind`
• Example:
 	
`> :k Tree`
`Tree :: * -> *`
 	
Explanation: A Tree requires one type parameter (e.g. `Int`) to be made a concrete type.
	String Kind
	`> :kind String`
	`String :: *`
	
	Map Kind
	`> :k Map`
	`Map :: * -> * -> *`
	
	Maybe Kind
	`> :k Maybe`
	`Map :: * -> *`
	
	Map String Kind
	`> :kind (Map String)`
	`(Map String) :: * -> *`
	
	Explanation: Map String is has one of the two type parameters filled so it has one less asterisk.
		• Similar to interfaces in Java.
		  ○ Like a contract.
		  ○ Implementation details can be included in typeclass definition.
		
		• No relation to classes in object-oriented programming.
		  ○ Example: Do not have any data associated with them.
		
		• Simplify polymorphism.
		
		Example: `Eq` Typeclass
		
		`class Eq a where`
		`    (==) :: a -> a -> Bool`
		`    (/=) :: a -> a -> Bool`
		`    x == y = not (x /= y)`
		`    x /= y = not (x == y)`
		
		The last two lines in the type class definition allow the developer to program either (==) or (/=) but not necessarily both.
			Example: Make `Maybe` an Instance of `Eq`
			
			`instance (Eq a) => Eq (Maybe a) of`
			`    (==) Nothing Nothing   = true`
			`    (==) (Just x) (Just y) = x == y`
			`    (==) _        _        = false`
			
			Need to ensure type "a" supports "Eq" so add that as a class constraint.
			
			Class Constraint
			• Operator: =>
			• Ensures that a type parameter satisfies some typeclass requirement.
			
			Kind of Typeclasses
			
			`> :k Eq`
			`Eq :: * -> Constraint`
			
			`> :k Num`
			`Num :: * -> Constraint`
			
			Note: Typeclasses are a class constaint (not a type) so their kind is different.

Lecture #08 – Functors

Functor Type Class Definition
 	
`class Functor f where`
`  fmap :: (a -> b) -> f a -> f b`
 	
This is very similar to the definition of the higher order function "map"
 	
`map :: (a -> b) -> [a] -> [b]`
	Functor – Something that can be mapped over.
	• Handles things "inside a box"
	
	Example: List (`[]`) as an instance of `Functor`
	
	`instance Functor [] where`
	`    fmap = map`
	
	Explanation: map is a specialized version of fmap for lists.
		Examples: `map` and `fmap` on Lists
		
		`> map (+1) [1, 2, 3]`
		`[2, 3, 4]`
		
		`> fmap (+1) [1, 2, 3]`
		`[2, 3, 4]`
		
		`> fmap (+1) []`
		`[]`
			Examples: `fmap` on Maybes
			
			`> fmap (+1) (Just 3)`
			`Just 4`
			
			`> fmap (+1) Nothing`
			`Nothing`

Example: `Maybe` as an Instance of `Functor` `instance Functor Maybe where` `   fmap _ Nothing = Nothing` `   fmap f (Just x) = Just (f x)` DO NOT FORGET THE `Just` IN VALID SOLUTION	**Either Algebraic Data Type** `data Either a b = Left a` `	Right b` ` deriving (Eq,Ord,Read,Show)`  • `Left` – Error type that is not mappable.  • `Right` – Expected type	Example: `Either` as an Instance of `Functor` `instance Functor (Either a) where` `   fmap _ (Left x) = Left x` `   fmap f (Right y) = Right (f y)` `> fmap (+1) Leftt 20` `20 -- No Change` `> fmap (+1) Right 20` `21 -- Changed`

IO in Haskell

• Haskell avoids side effects but they are inevitable in real programs.
 	
• Monads
  ○ Related to Functors
  ○ Compartmentalize side effects.
 	
• ()
  ○ Unit type in Haskell
	Type Signature of the `main` Function in Haskell
	`main :: IO ()`
	
	Hello World in Haskell
	`main = putStrLn "Hello World"`
	
	Type Signature of getLine
	`getLine :: IO String`
		• `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"	

do Example

```
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
```

return in Haskell

- Unrelated to "return" in other languages
- Better described as "wrap" or "box"

Summary:
`return` – Boxes an IO (since IO is a monad)

`<-` Unboxes an IO

Type of the Unit Type ()

- Base type

```
> :t ()
() :: ()
```

Type of return

```
> :t (return ())
(return ()) :: Monad m => m ()
```

`Monad` is a **typeclass**.

Using IO as a Functor

```
main = do
       line <- fmap (++"!!!") getLine
       putStrLn line
```

Explanation: This function takes a string input from standard in and appends "!!!" at which point it prints it to the console.

Definition of IO as a Functor

```
instance Functor IO where
  fmap f action = do
               result <- action
               return (f result)
```

Explanation: The action object is taken out of the IO box, the function "f" applied to it, and then returned to the IO box.

id Function

- Takes one input parameter and returns that input parameter unmodified. Examples:

```
> id 3
3

> id "Hello World"
"Hello World"
```

Functor Laws

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.

Examples:

```
> fmap id (Just 3)
Just 3
> fmap id Nothing
Nothing
> fmap id [1, 2, 3]
[1, 2, 3]
```

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.

Law #2 Written Formally

```
fmap (f . g) = fmap f . fmap g
```

The Functor laws are NOT enforced. They are good practice that makes the code easier to reason about.

Lecture #09 – Applicative Functors

Functor – Something that can be mapped over. Allow you to map functions over different data types. **Examples**:

- Maybe
- Either
- IO
- Lists
- <*>

Functors return boxed up values.

Functor Example

```
> fmap (+1) [1, 2, 3]
[2, 3, 4]

> let x = fmap (+) [1, 2, 3]
```

Explanation: In this case **x** is:
`[(1+), (2+), (3+)]`

Applicative Functor

- Requires the importing of a special library as shown below:

```
import Control.Applicative
```

Functions in Applicative Typeclass:

- `pure` – Wraps/boxes a value
- `<*>` - Infix version of `fmap`. Is itself a Functor.

Example Uses of pure

```
> pure 7
7

> pure 7 :: Maybe Int
Just 7

> pure 7 :: [Int]
[7]
```

Type Class Definition of Applicative

```
class (Functor f) => Applicative f where
     pure :: a -> f a
     <*> :: f (a -> b) -> f a -> f b
```

Only difference between `<*>` and `fmap` is that the function in `<*>` is boxed while it is not in `fmap` (see the green `f`).

Make Maybe an Instance of Applicative

```
instance Applicative Maybe where
     pure = Just
     Nothing <*> _ = Nothing
     (Just f) <*> x = fmap f x
```

Explanation: `pure` simply wraps the value in `Just`. No need to explicitly check if "x" is maybe as `fmap` will do that for you.

Examples of Applicative Maybe

```
> 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
```

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.**

Making [] an Instance of Applicative

```
instance Applicative [] where
   pure x = [x]
   fs <*> xs = [f x | f <- fs, x <- xs]
```

Explanation: The function is actually a list of functions so list comprehension is needed.

Example Use of Applicative on Lists

```
> (*) <$> [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]
```

Definition of IO as an Instance of Applicative

```
instance Applicative IO where
    pure = return
    a <*> b = do
              f <- a
              x <- b
              return (f x)
```

Example of `Applicative IO`

```
import Control.Applicative

main = do
       a <- (++) <$> getLine <*> getLine
       putStrLn a
```

`liftA2`

A function that simplifies the application of a normal function to two Functors.

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> fc
liftA2 f x y = f <$> a <*> b
```

Example of `liftA2`

```
> (:) <$> Just 3 <*> Just [4]
Just [3, 4]
> liftA2 (:) (Just 3) (Just [4])
Just [3, 4]
```

Applicative Functor Definition

A functor you can apply to other Functors.

Applicative Functor Laws

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

Monoids

Monoid: An **associative** binary function and a value that acts as an **identity** with respect to that function.

Examples

- x * 1 — Identity of Multiplication
- lst ++ [] — Identity of Concatenation
- x + 0 — Identity of Addition

Definition of Monoid Typeclass

```
class Monoid m where
      mempty :: m
      mappend :: m -> m -> m
      mconcat :: [m] -> m
      mconcat = foldr mappend mempty
```

Monoid Rules

Rule #1: ``mempty `mappend` x = x``	Rule #2: ``x `mappend` mempty = x``	Rule #3: ``(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)``

Lecture #10 – Monads

Functor – Something that can be mapped over.
Definition:

```
instance Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Problem with Functors: Do not support chaining of multiple commands. Example:

```
> fmap (+) (Just 3) (Just 4)
```

Returns an error since it cannot resolve `(Just 3+)` and `(Just 4)`

Applicative Functor: A Functor that can be applied to other Functors.

```
class (Functor f) => Applicative f where
    (<*>) :: f (a -> b) -> f a -> f b
```

Requires library `Control.Applicative`

Even with Applicative Functors, it is not possible to chain together multiple commands. Example:

```
> Just (+3) <*> Just (+4) <*> Just (+5)
```

Returns error

Monads: Can chain through a series of functions.

Key Operator:
`>>=` (Bind)

Example #1: Using Just

```
> (Just 3) >>= (\x -> Just (x + 4)) >>= (\y -> Just (y+5))
12
```

Example #2: Using `return`

```
> (return 3) >>= (\x -> return (x + 4)) >>= (\y -> return (y+5))
12
```

Comparing `<*>` and `>>=`

Functor:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Monad:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Differences:

1. Order of the arguments changed.
2. The function is boxed in Functor but not Monad
3. Monad function returns a boxed result.

Example of `<$>`, `<*>` and `>>=`

```
> (\x -> x + 1) <$> Just 3
Just 4

> Just (\x -> x + 1) <*> Just(3)
Just 4

> (Just 3) >>= (\x -> Just(x+1))
Just 4
```

Example: Implement `applyMaybe` that applies a function to a `Maybe`

```
applyMaybe :: Maybe a -> (a -> b) -
> (Maybe b)
applyMaybe Nothing _ = Nothing
applyMaybe (Just x) f = Just (f x)
```

Example: Implement `applyMaybe` that applies a function to a `Maybe`

```haskell
applyMaybe :: Maybe a -> (a -> Maybe b)
           -> (Maybe b)

applyMaybe Nothing _ = Nothing
applyMaybe (Just x) f = Just (f x)
```

Chaining applyMaybe

```haskell
> (Just 3) `applyMaybe` (\x -> Just (x*2))
           `applyMaybe` (\y -> Just (y-1))
Just 5

> (Just 3) `applyMaybe` (\_ -> Nothing)
           `applyMaybe` (\y -> Just (y-1))
Nothing
```

Additional Names for Monoids

- "Programmable Semicolons"
- "Applicative Functors you can chain."

Monad Typeclass Definition

```haskell
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
```

Example a Robot Moving Towards a Goal (Not Failure)

```haskell
--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)
```

```haskell
-- 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)
```

Maybe as an Instance of the Monad Typeclass

```haskell
instance Monad Maybe where

     return = Just

     (>>=) Nothing  _ = Nothing
     (>>=) (Just x) f = f x

     fail  _          = Nothing
```

Example a Robot Moving Towards a Goal (with Failure)

```haskell
-- 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)
```

```haskell
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`

Integer Division Using Monads

Integer Division with Bind and No "do"

```haskell
mydiv :: Maybe Int -> Maybe Int -> Maybe Int
mydiv x y = x >>= (\numer ->
            y >>= (\denom  ->
            if denom > 0
                then Just (div numer denom)
                else fail "Div by zero"))
```

Integer Division with Bind with "do"

```haskell
mydiv :: Maybe Int -> Maybe Int -> Maybe Int
mydiv x y = do
            numer <- x
            denom <- y
            if denom > 0
                then Just (div numer denom)
                else fail "Div by 0"
```

Integer Division with Bind with "do" and return

```haskell
mydiv :: Maybe Int -> Maybe Int -> Maybe Int
mydiv x y = do
            numer <- x
            denom <- y
            if denom > 0
               then return $ div numer denom
               else fail "Div by 0"
```

List Monad

Making `List` an Instance of Monad

```haskell
instance Monad [] where
        return x = [x]
        (>>=) xs f = concat(map f xs)
        fail  _     = []
```

Explnation: `concat` is needed here as `f` returns elements already in a list. As such, `concat` merges the individual lists (from each call to `f`) into a single list.

Example Use of `List` as a Monad

```haskell
listOfTuples :: [(Int, Char)]
listOfTuples = do
                n <- [1, 2]
                ch <- ['a', 'b']
                return (n, ch)
> listOfTuples
[(1,'a'), (1,'b'), (2,'a'), (2, 'b')]
```

Combining a `Maybe` and a `List` Monad

```haskell
> Just [2,3] >>= (\x -> Just( fmap (+1) x))
[3, 4]
```

Lecture #11 – Parsing Combinators

Semantics: Enumerate what a program means. Defined by the interpreter or compiler.

Syntax: Enumerate how a program Is structured. Defined by the lexer and parser.

Compilation Flow

Step #1: Tokenizer/lexer generates a set of tokens.

Step #2: Parser turns the tokens into an abstract syntax tree.

Step #3: Compilers and interpreters convert the AST into machine code or commands respectively.

Lexer

Converts the characters of the program into words of the language.

Examples:

- Lex/Flex (C/C++)
- ANTLR & JavaCC (Java)
- Parsec (Haskell)

Categories of Tokens

- **Reserved Words/Keywords**.
  - Examples: while, if, then, else
- **Literals/Constants**.
  - Examples: 123, "Hello World!"
- **Special symbols**.
  - Examples: ";", "=>", "&&"
- **Identifiers**.
  - Examples: "balance", "myFunction"

Parsing

- **Parser converts tokens to abstract syntax trees**.
- **Defined by context free grammars** (CFG)
- **Types of Parsers:**
  - Bottom-up/Shift-Reduce Parsers
  - Top-down parsers

Context Free Grammars

- Grammars specify the language.
- Specified in Backus-Naur form format. Example:

```
Expr -> Number
      | Number + Expr
```

- **Terminal** – **Cannot be broken down** further.
- **Non-terminals** – **Can be broken down** further.

Example: "0", "1", "2", … , "9" are terminals but digit, number, and expression are not.

Example Grammar

```
expr -> expr + expr
      | expr - expr
      | ( expr )
      | number

number -> number digit
        | digit

digit -> 0 | 1 | 2 | … | 9
```

Bottom-Up / Shift-Reduce Parser

- **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.
  - Examples: YACC/Bison
- **Fading from popularity**

Top-Down Parser

- **Non-terminals are expanded to match tokens**.
- **LL** – *Left to right, Leftmost derivation*
- **LL(k) Parser** – Looks ahead up to *k* elements.
  Examples: Java CC, ANTLR
  - The higher the *k*, the more difficult language is to parse. ***k* can be arbitrary**.
  - ***LL(1)*** - Easy to parse using either LL or recursive descent parsers. **Many computer languages are designed to be LL(1)**.

Parser Combinator

Combine simpler parsers to make a more complex parser.

Example: Parsec

Useful Parsec Functions

- `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

```haskell
import Text.ParserCombinators.Parsec

num :: GenParser st String
num = many1 digit

main = do
       print $ parse num "Hello" "42"
```

```haskell
import Text.ParserCombinators.Parsec

num :: GenParser st Integer
num = do
      str <- many1 digit
      return $ read str

main = do
        print $ parse num "World" "42"
```

- `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.

Example with `try`, `<|>`, and `<?>`

```haskell
eol =  try (string "\n")
   <|> string "\n\r"
   <?> "end of line"
```

- `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
• **Purely Functional** • **Lazy evaluation** • **Fully Curried Language** • **Statically Typed** • **Type Inference** – Via context, Haskell can deduce the type.	• **Referential Transparency** – A **function call** can be replaced with its equivalent value without affecting the program • **No (re)assignment** • **No loop** • **No side effects**	• **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**	• **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	Concise Approach

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

Kind of Show and show

```
> :k Show
Show :: * -> Constraint
```

Type and Kind of show

```
> :k show
Error (A function not a type)
> :t show
show :: (Show a) => a -> String
```

Lambda and ADT Combined

```
> (\x -> Just (x+1)) 1
Just 2
```

Creating Type Alias

```
type String = [Char]
```

Allows for more readable code as developer can use a type name that makes more sense for a given application.

Example: `applyMaybe` that takes a `(Maybe a)` and applies to it a function that takes a normal `a` and returns a `(Maybe b)`

```
applyMaybe :: (Maybe a) -> (a -> Maybe b) -> (Maybe b)
applyMaybe Nothing _ = Nothing
applyMaybe (Just x) f = f x
```

Explanation: Since the function "`f`" already returns a Maybe, you do not need to re-box it. However, since it does not take a Maybe, you need to unbox the first input parameter.

Applying return to Items

```
> return 7
7
> return 7 :: Maybe Int
Just 7
> return 7 :: [Int]
[7] -- Need Int or get an error
```

Conclusion: **Behavior for return is the same as** `pure`. Both put the object in the **minimum default context that still yields that value**.

List comprehension is syntactic sugar for using lists as monads.

Monads and Lambda

When trying to chain multiple functions together in a Monad, remember the Monad must return a **boxed value**. Hence, **Lambda often work well as they simplifying boxing**.

Applicative Typeclass – Allows you to use normal functions on values that have a context (i.e. are inside a Functor).

Monad: Given a value of type, **a**, in a context, **m**, apply a function that takes a normal value of type **a** and returns a value in the context **m**.

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Monads are just applicative functors that support bind (>>=).

Key Difference: **Applicative functors support normal functions that take and return unboxed values while Monads return boxed values**.

`return` – Monad equivalent of "pure" for Applicative Functors.

Cannot use `fmap` in the definition of a `Monad` since `fmap` returns a boxed value while the function of the `Monad` returns a boxed value. Hence, if you used `fmap` with a `Monad`, you would return a double boxed value.

Functor Definitions

Lists

```
instance Functor [] where
  fmap = map
```

Maybe

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

IO

```
instance Functor IO where
  fmap f a = do
          x <- a
          return (f x)
```

Applicative Functor Definitions

Lists

```
instance Applicative [] where
 pure x = [x]
 (<*>) fs xs = [ f x | f <- fs, x <- xs]
```

Maybe

```
instance Applicative Maybe where
  pure x = Just x
  (<*>) Nothing _ = Nothing
  (<*>) (Just f) x = fmap f x
```

IO

```
instance Applicative IO where
  a <*> b = do
           f <- a
           x <- b
           return (f x)
```

Monad Definitions

Lists

```
instance Monad [] where
 return x = [x]
 (>>=) xs f = concat $ map f x
 fail  _   = []
```

Maybe

```
instance Monad Maybe where
  return x  = Just x
  (>>=) Nothing  _ = Nothing
  (>>=) (Just x) f = f x
  fail _    = Nothing
```

IO

```
instance Monad IO where
    (>>=) a f = do
              x <- a
              f x
    fail s = ioerror (userError s)
```

CS252 – Final Exam Study Guide

By: Zayd Hammoudeh

Lecture #12 – Introduction to JavaScript

JavaScript

- Developed at Netscape by Brendan Eichs in 10 days
- Originally named "Mocha"
- Syntax similar to Java

Multiparadigm JavaScript

Supported programming paradigms:

- Imperative
- Functional
- Object-Oriented (**through prototypes**)

Where JavaScript is Run

- **Client Side Versions**
  - Runs on user machine
- **Server-side Versions**
  - JVM: **Rhino** & **Nashorn**
  - **Node.js**

Example: Imperative JavaScript

```
function addList(list){
  var = i, sum = 0;
  for( i = 0; i < list.length ; i++){
    sum += list[i];
  }
  return sum;
}
```

Example: Functional JavaScript

```
function addList(list){
  if(list.length == 0){
    return 0;
  }

  return list[0]
              +
addList(list.slice(1));
}
```

`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.

Example: Object-Oriented JavaScript

```
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)
```

`Adder` – Name of a new constructor. **Convention is to start constructors with a capital letter.**

Example: Functional JavaScript

```
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;}
```

Printing to the Console in JavaScript

- **Standard Approach:**

  `console.log("…")`

  - Not supported by all implementations.
- JVM based JavaScript Approach:
  `print`
- Solution to Support a Single Interface:
  `var print = console.log`

Closures

- Functions whose inner variables refer to independent (free) variables.

Closure Example

```
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"
```

Node.js

- 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

- Functions in JavaScript are first class objects of type "Object".
- Not executed immediately.

Reading from a File with Callbacks in Node.js

```
var fs = require('fs')
fs.readFile('myFile.txt',
    function(err, data){
      if(err)
        throw err;
      else
        console.log("" + data);
    }
}
console.log("All done")
```

"`All done`" prints before the file contents due to callbacks.

`require` – Includes the JavaScript package "`fs`"

Synchronous File IO in Node

```
var fs = require('fs')
var data =
fs.readFileSync('myFile.txt');
console.log("All done")
```

To eliminate callbacks, most function names can be appended with "`Sync`"


```
    var nextInt = 0;
    return function(){
                return nextInt++;
            }() // Double paren run the function
  }

  // c is "undefined" since no return
  var c = noReturnAdd(x, y)
```

```
//Lambda Function
var myLambda = function(x){ return x * x; }
```