# CS252 – Midterm Exam Study Guide
## By: Zayd Hammoudeh

## Lecture #01 – General Introduction

| Reasons for Different Programming Languages | Programming Language Design Choices | Features of Good Programming Languages | |
|---|---|---|---|
| 1. **Different domains** (e.g. web, security, bioinformatics)<br>2. **Legacy code and libraries**<br>3. **Personal preference** | 1. **Flexibility**<br>2. **Type safety**<br>3. **Performance**<br>4. **Build Time**<br>5. **Concurrency** | 1. **Simplicity**<br>2. **Readability**<br>3. **Learnability** | 4. **Safety** (e.g. security and can errors be caught at compile time)<br>5. **Machine independence**<br>6. **Efficiency** |
| | | **Goals almost always conflict** | |

| Conflict: Type Systems | 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 | Advantages of Web and Scripting Languages |
|---|---|---|---|
| • **Advantage:** Prevents bad programs.<br>• **Disadvantage:** Reduces programmer flexibility. | | • **Multi-core "explosion"**<br>• **Big Data**<br>• **Mobile Devices** | • **Examples:** Perl, Python, Ruby, PHP, JavaScript<br>• **Highly flexible**<br>• **Dynamic typing**<br>• **Easy to get started**<br>• **Minimal typing** (i.e. type systems) |

| Major Programming Language Research Contributions | Programs that Manipulate Other Programs | Formal Semantics | Types of Formal Semantics |
|---|---|---|---|
| • **Garbage collection**<br>• *Sound* **type systems**<br>• **Concurrency tools**<br>• **Closures** | • **Compilers & interpreters**<br>• **JavaScript rewriting**<br>• **Instrumentation**<br>• **Program Analyzers**<br>• **IDEs** | • Used to **share information** *unambiguously*<br><br>• **Can formally prove a language supports a given property**<br><br>• ***Crisply define** how a language works* | • **Operational**<br>  ○ Big Step "*natural*"<br>  ○ Small Step "*structural*"<br>• **Axiomatic**<br>• **Denotational** |

### Haskell

| | | |
|---|---|---|
| • **Purely functional** – Define "*what stuff is*"<br>• **No side effects**<br>• **Referential transparency** – **A function with the same input parameters will always have the same result**.<br>  ○ **A function call can be replaced with its value and nothing will change**.<br>• **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.<br>  ○ **More flexible** but **less safe**.<br>  ○ **Supported by Haskell**<br>  ○ **Common in scripting languages** (e.g. Python, Ruby) | **Side Effects in Haskell**<br>• Generally not supported.<br>• **Example of Support Side Effects**: File IO<br>• Functions that do have side effects must be separated from other functions.<br>**Lazy Evaluation**<br>• **Results are not calculated until they are needed**<br>• **Allows for the representation of infinite data structures** |

## Lecture #02 – Introduction to Haskell

| Key Traits of Haskell | ghci / let | Run Haskell from Command Line | Hello World in Haskell |
|---|---|---|---|
| 1. **Purely functional**<br>2. **Lazy evaluation**<br>3. **Statically typed**<br>4. **Type Inference**<br>5. **Fully curried functions** | `ghci` – Interactive Haskell.<br><br>`let` – Keyword required in ghci to set a variable value. **Example**:<br>`> let f x = x + 1`<br>`> f 3`<br>`4` | **Run Haskell from Command Line**<br>Use `runhaskell` keyword.<br><br>**Example**:<br>`> runhaskell <FileName>.hs` | `main :: IO ()`<br>`main = do`<br>`   putStrLn "Hello World"` |

| Primitive Classes in Haskell | Lists | | Ranges |
|---|---|---|---|
| 1. `Int` – **Bounded** Integers<br>2. `Integer` – **Unbounded**<br>3. `Float`<br>4. `Double`<br>5. `Bool`<br>6. `Char` | • **Base 0**<br>• Comma separated in square brackets<br>• **Operators**<br>  ○ `:` Prepend<br>  ○ `++` Concatenate<br>  ○ `!!` Get element a specific index<br>  ○ `head` First element in list<br>  ○ `tail` All elements after head | ○ `last` Last element in the list<br>○ `init` All elements in the list except the last one<br><br>○ `take n` Take first n elements from a list<br><br>○ `replicate l m` Create a list of length `l` containing only `m`<br>○ `repeat m` Create an infinite list containing only `m` | • Can be infinite or bounded<br>• Use the "`..`" notation. **Examples**:<br>`> [1..4]`<br>`[1, 2, 3, 4]`<br><br>`> [1,2..6]`<br>`[1, 2, 3, 4, 5, 6]`<br><br>`> [1,3..10]`<br>`[1, 3, 5, 7, 9]`<br><br>`> [5, 4..1]`<br>`[5, 4, 3, 2, 1]` |
| **Hello World in Haskell**<br>`main :: IO ()`<br>`main = do`<br>`   putStrLn "Hello World"` | **List Examples**<br>`> putStrLn $ "Hello " ++ "World"`<br>`"Hello World"`<br><br>`> let s = bra in s !! 2 : s ++ 'c' : last s : 'd' : s`<br>`"abracadabra"` | | **Infinite List Example**<br>`> let even = [2,4..]`<br>`> take 5 even`<br>`   [2, 4, 6, 8, 10]` |

<table>
<tr><td colspan="2">

**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)]
```

</td></tr>
</table>

**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 ::=<br>    true<br>    \| false<br>    \| if e<br>      then e<br>      else e | **Expressions:**<br>  constant true<br>  constant false<br>  conditional |
| v ::=<br>    true<br>    \| false | **Values:**<br>  constant true<br>  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"


A Review of Compilers

We don't care about lexing or parsing.

source code → Lexer/Tokenizer → tokens → Parser → Abstract Syntax Tree (AST) → Compiler / Interpreter

Compiler → Machine code
Interpreter → Commands

We don't care if we have a compiler or interpreter

---

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

**E-IfFalse:**
$$\frac{}{\text{if false } \textbf{then } e_2 \textbf{ else } e_3 \rightarrow e_3}$$

**E-If:**
$$\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow \text{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

**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.

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

- $f$ – 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 | foldl | |
|---|---|---|
| • Built in Haskell higher order function<br>• **Applies a function to all elements of a list.**<br><br>```map :: (a -> b) -> [a] -> [b]```<br><br>```> map (+1) [1, 2, 3]```<br>```[2, 3, 4]``` | • Built in higher order function<br>• **Does not support infinite lists**.<br>• **Should only be used for special cases**.<br><br>```foldl :: (b -> a -> b) -> b -> a -> b```<br><br>**Example:**<br>```> foldl (\x y -> x - y) 0 [1, 2, 3, 4]```<br>```-10 -- (((0-1) - 2) - 3) - 4``` | foldl $f\,z$ |
| filter | foldr | |
| • Built in Haskell higher order function<br>• **Removes all elements from a list that do not satisfy (i.e. make true) some predicate.**<br><br>```filter :: (a -> Bool) -> [a] -> [a]```<br><br>```> filter (>2) [1, 2, 3, 4]```<br>```[3, 4]``` | • Built in higher order function<br>• **Supports infinite lists**.<br>• "*Usually the right fold to use*"<br><br>```foldr :: (b -> a -> a) -> a -> b -> a```<br><br>**Example:**<br>```> foldr (\x y -> x + y) 0 [1, 2, 3, 4]```<br>```-2 -- 1 - (2 - (3 - (4 - 0)))``` | foldr $f\,z$ |

| Thunk – A delayed computation<br><br>Due to lazy evaluation, **foldl and foldr build thunks rather than calculate the results as they go**. | foldl'<br><br>• **Data.List.foldl'** evaluates its results eagerly (i.e. does not use **thunks**)<br>• **Good for large, but finite lists.** | foldl **in terms of** foldr<br><br>```myFoldl' f acc x = foldr (flip f) acc (reverse x)``` |
|---|---|---|

# Lecture #05 – Small-Step Operational Semantics

| WHILE Language | Small Step Semantics with State | Evaluation Order Rules |
|---|---|---|
| • Unlike the Bool* language, **WHILE supports mutable references**. | • Since the WHILE language supports mutable references, the grammar must be updated to support it. | • **Tend to be repetitive and clutter the semantics.**<br>• **Context based rules tend to represent the same information as evaluation order rules but more concisely.** |

| WHILE Language (grammar) | | Small Step Semantics with State | Evaluation Order Rules |
|---|---|---|---|
| ```e ::= a```<br>   ```| v```<br>   ```| a:=e```<br>   ```| e;e```<br>   ```| e op e```<br>   ```| if e then e```<br>         ```else e```<br>   ```| while (e) e``` | Variable/addresses<br>Values<br>Assignment<br>Sequence<br>Binary Operations<br>Conditional<br><br><br>While Loops | **While Relation:**<br>$$e, \sigma \rightarrow e', \sigma'$$<br>• $\sigma$ – Store. **Maps *references* to values.** | **Reduction Rule**<br>Rewrites the expression. Example:<br>**E-IfFalse:**<br>```if false then e2 else e3 → e3``` |
| ```v ::= i```<br>   ```| b``` | Integers<br>Boolean | **Example Operations:**<br>• $\sigma(a)$ – Retrieves the value at address "$a$"<br>• $\sigma[a := v]$ – Identical to the original store with the exception that it now stores the value $v$ at address "$a$" | **Context Rule**<br>**Specify the order for evaluating expressions**. Example:<br>**E-If:**<br>$$\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$ |
| ```op ::= + | - | * | /```<br>     ```| >= | > | <= | <``` | | | |

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

| Evaluation Contexts | Rewriting Evaluation Order Rules | Data.Map |
|---|---|---|
| • **Alternative to evaluation order rules**.<br>• **Marker (•) / hole** indicate the **next place for evaluation** (i.e. where we will do the work).<br><br>**Example:**<br>  C[**r**]<br>  = **if** (**if** true **then** false **else** false) **then** true **else** true<br><br>    **r** = **if** true **then** false **else** false<br><br>    C = **if** • **then** true **else** true<br><br>C[**r**] is the original expression. | **Context based rules only apply to reducible expressions (redexs). Example:**<br><br>**EC-IfFalse:**<br>  C[**if** false **then** $e_2$ **else** $e_3$] → C[$e_3$]<br><br>---<br><br>**Context Syntax**<br><br>```C ::= •```<br>   ```| if C then e else e```<br>   ```| C op e```<br>   ```| v op C```<br>   ```| ...``` | • **Library:** import Data.Map as **Map**<br>• **Immutable**<br>• **Example Methods:**<br>  ○ **Map.empty** – Creates and returns an empty map<br><br>  ○ **Map.insert k v m** – Inserts a value "**v**" at key "**k**" into map "**m**". **Returns a new, updated map.**<br><br>  ○ **Map.lookup k m** – Returns the value at key "**k**" in map "**m**". **Wrapped in a Maybe.**<br><br>  ○ **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 *&lt;Precondition&gt;*, then *&lt;LeftSideArrow&gt;* evaluates to *&lt;RightSideArrow&gt;*." |
|---|---|---|

# 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`"

### 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.

- **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.

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

# Lecture #08 – Functors

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

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

## IO in Haskell

| • Haskell avoids side effects but they are inevitable in real programs.<br><br>• **Monads**<br>  o Related to Functors<br>  o Compartmentalize side effects.<br><br>• **()**<br>  o Unit type in Haskell | **Type Signature of the main Function in Haskell**<br>`main :: IO ()`<br><br>**Hello World in Haskell**<br>`main = putStrLn "Hello World"`<br><br>**Type Signature of getLine**<br>`getLine :: IO String` | • **do** – Allows for the chaining of multiple IO/Monad commands together. **Syntactic sugar for bind** ">>="<br><br>• **<-** Extracts data out of an IO/Monad "Box"<br><br>• **return** – Places data into an IO/Monad "Box" | |
|---|---|---|---|

| do Example | return in Haskell | Type of the Unit Type () |
|---|---|---|

### 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
**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` | `liftA2` |
|---|---|
| `import Control.Applicative`<br><br>`main = do`<br>    `a <- (++) <$> getLine <*> getLine`<br>    `putStrLn a` | A function that simplifies the application of a normal function to two Functors.<br><br>`liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> fc`<br>`liftA2 f x y = f <$> a <*> b` |

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

## Applicative Functor Laws

| Law 1:<br>    `pure f <*> x = fmap f x` | Law 2:<br>    `pure id <*> v = v` | Law 3:<br>`pure (.) <*> u <*> v <*> w = u <*> (v <*> w)` |
|---|---|---|
| Law 4:<br>  `pure f <*> pure x = pure (f x)` | Law 5:<br>  `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.<br><br>**Examples**<br>• x * 1          Identity of **Multiplication**<br>• lst ++ []        Identity of **Concatenation**<br>• x + 0          Identity of **Addition** | **Definition of Monoid Typeclass**<br><br>`class Monoid m where`<br>    `mempty :: m`<br>    `mappend :: m -> m -> m`<br>    `mconcat :: [m] -> m`<br>    `mconcat = foldr mappend mempty` | |
|---|---|---|

## Monoid Rules

| Rule #1:<br>   `mempty `mappend` x = x` | Rule #2:<br>   `x `mappend` mempty = x` | Rule #3:<br>  `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)` |
|---|---|---|

## Lecture #10 – Monads

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

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

| Comparing `<*>` and `>>=` | Example of `<$>`, `<*>` and `>>=` | |
|---|---|---|
| **Functor**:<br>`(<*>) :: Applicative f => f (a -> b) -> f a -> f b`<br><br>**Monad**:<br>`(>>=) :: Monad m => m a -> (a -> m b) -> m b`<br><br>**Differences:**<br>1. **Order of the arguments changed.**<br>2. **The function is boxed in Functor but not Monad**<br>3. **Monad function returns a boxed result.** | `> (\x -> x + 1) <$> Just 3`<br>`Just 4`<br><br>`> Just (\x -> x + 1) <*> Just(3)`<br>`Just 4`<br><br><br>`> (Just 3) >>= (\x -> Just(x+1))`<br>`Just 4` | **Example:** Implement **applyMaybe** that applies a function to a **Maybe**<br><br>`applyMaybe :: Maybe a -> (a -> b) -> (Maybe b)`<br>`applyMaybe Nothing _ = Nothing`<br>`applyMaybe (Just x) f = Just (f x)` |

<table>
<tr>
<td>

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

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

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

</td>
<td>

**Chaining `applyMaybe`**

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

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

</td>
<td>

**Additional Names for Monoids**

- "Programmable Semicolons"

- "Applicative Functors you can chain."

</td>
</tr>
</table>

<table>
<tr>
<td colspan="2">

**Monad Typeclass Definition**

```
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
```

</td>
<td colspan="2">

**Example a Robot Moving Towards a Goal** (Not Failure)

```
--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)
```

</td>
</tr>
</table>

<table>
<tr>
<td>

**Maybe as an Instance of the Monad Typeclass**

```
instance Monad Maybe where

    return = Just

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

    fail _           = Nothing
```

</td>
<td colspan="2">

**Example a Robot Moving Towards a Goal** (with Failure)

```
-- 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)
```

</td>
<td>

```
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**

</td>
</tr>
</table>

## Integer Division Using Monads

<table>
<tr>
<td>

**Integer Division with Bind and No "do"**

```
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"))
```

</td>
<td>

**Integer Division with Bind with "do"**

```
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"
```

</td>
<td>

**Integer Division with Bind with "do" and return**

```
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"
```

</td>
</tr>
</table>

## List Monad

<table>
<tr>
<td>

**Making List an Instance of Monad**

```
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.

</td>
<td>

**Example Use of List as a Monad**

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

</td>
<td>

**Combining a Maybe and a List Monad**

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

</td>
</tr>
</table>

# Lecture #11 – Parsing Combinators

<table>
<tr>
<td>

**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**.

</td>
<td>

**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.

</td>
<td>

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

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

</td>
</tr>
</table>

## 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** – **L**eft to right, **R**ightmost derivation

- **LALR** – **L**ook-**A**head **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** – **L**eft to right, **L**eftmost 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

```
import Text.ParserCombinators.Parsec

num :: GenParser st String
num = many1 digit

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

```
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 **<?>**

```
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** | a. **True** | a. **False** – Big step | a. **False** – Imperative | a. **True** |
| b. **False** – Lazy evaluation | b. **False** – Applicative functor | b. **True** | b. **True** | b. **False** – Typeclass |
| c. **False** – Lazy evaluation | c. **True** | c. **False** – Use store | c. **False** | c. **True** |
| d. **False** – Statically type | d. **True** | d. **True** | d. **True** | d. **False** |
| e. **True** | e. **True** | e. **False** | e. **True** | e. **False** – Algebraic data type |

| Haskell | Purely Functional | Functional Languages | Operational Semantics |
|---|---|---|---|
| • **Purely Functional**<br>• **Lazy evaluation**<br>• **Fully Curried Language**<br>• **Statically Typed**<br>• **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<br>• **No (re)assignment**<br>• **No loop**<br>• **No side effects** | • **Functions are first class objects** meaning they can be passed to a function, returned from it, or created on the fly.<br><br>• **Higher order function support** | • **Small Step** – Structural Semantics<br><br>• **Big Step** – Natural Semantics<br><br>• "**Get stuck**" – When a function is encountered that does not have an associated rule. |

# CSV Parser Example

| Verbose Approach | Concise Approach |
|---|---|

**Verbose Approach**

```haskell
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**

```haskell
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

<table>
<tr><td>

**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
```

</td><td>

**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.

</td><td>

**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.

</td></tr>
</table>

<table>
<tr><td>

**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**.

</td><td>

**List comprehension is syntactic sugar for using lists as monads.**

</td><td>

</td></tr>
</table>

<table>
<tr><td>

**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**.

</td><td>

**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**.

</td><td>

**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.**

</td></tr>
</table>

## Functor Definitions

<table>
<tr><td>

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

</td><td>

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

</td><td>

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

</td></tr>
</table>

## Applicative Functor Definitions

<table>
<tr><td>

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

</td><td>

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

</td><td>

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

</td></tr>
</table>

## Monad Definitions

<table>
<tr><td>

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

</td><td>

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

</td><td>

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

</td></tr>
</table>