**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** | 1. **Safety** (e.g. security and can errors be caught at compile time) 2. **Machine independence** 3. **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** | | **Ranges** |
| * **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 | * 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 Simple Function**  **> let inc x = x + 1**  **> inc 3**  **4**  **> inc 4.5**  **5.5**  **> inc (-5) -- Negative**  **-4** | **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** |
| **> [(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)]** | **Type Signature**   * Uses symbols “**::**” and “**->**” * **Example**:   **inc :: Int -> Int**  **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**. | | **Formal Semantic Styles**   * **Operational** – **Specify how expressions should be evaluated.**   + Big-Step (“Natural”)   + Small-Step (“structural”) * **Axiomatic** * **Denotational** |  |
| **Abstract Syntax Tree**  Tree representation of the abstract syntactic structure of a program’s source code. **Example is Bool\* language below**. | |
| **Big Step Operational Semantics**   * **Evaluates every expression to a value** * : “***Evaluates to***” symbol in Big-Step operational semantics. * **Example Formatting:** * **Read as:** “Expression e ***evaluates*** to the value v” |
| **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** |

|  |  |  |
| --- | --- | --- |
| **Small-Step Operational Semantics**   * Evaluate an expression until it is in ***normal form*** * **Normal Form** – Any form that cannot be evaluated further. * : “***Evaluates to***” symbol in small step operational semantics. **Example:** * : Many evaluation steps required. **Example:** | **Bool\* Small-Step Operational Semantics Rules** | **Example:** Reduce the expression  **Step #1:** Use rule “E-IfTrue” with “E-If”  **Step #2:** Use rule “E-IfFalse” (Now in normal form) |
| **E-IfTrue:** |
| **E-IfFalse:** |
| **E-If:** |

|  |  |  |  |
| --- | --- | --- | --- |
| **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** | **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.”** |
| **e ::=**  **true**  **| false**  **| if e then e else e**  **| 0**  **| succ e**  **| pred e** |
| **v ::= true | false**  **| IntV**  **IntV ::= 0 | succ IntV** |

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

|  |  |  |
| --- | --- | --- |
| **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 is a rule that associates to each from some set of values a unique from a set of values. * – Name of the function * – Independent variable * – Dependent variable * – Domain * – 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]** | **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** |  |
| **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]** | **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**. | | **Small Step Semantics with State**   * Since the WHILE language supports mutable references, the grammar must be updated to support it.   **While Relation:**   * – Store. **Maps *references* to values.**   **Example Operations:**   * – Retrieves the value at address “” * – Identical to the original store with the exception that it now stores the value at address “” | **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.** |
| **e ::= a**  **| v**  **| a:=e**  **| e;e**  **| e op e**  **| if e then e**  **else e**  **| while (e) e** | Variable/addresses  Values  Assignment  Sequence  Binary Operations  Conditional  While Loops | **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:** |
| **v ::= i**  **| b** | Integers  Boolean |
| **op ::= + | - | \* | /**  **| >= | > | <= | <** | |

|  |  |  |
| --- | --- | --- |
| **Reducible Expression** (**Redex**) – Any expression that can be transformed (reduced) in one step. | **Example: Redex**  This reduces to “” | **Example: Not a Redex**  Not a redex as expression “” 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**:  is the original expression. | **Rewriting Evaluation Order Rules**  **Context based rules only apply to reducible expressions** (redexs). **Example**:  **EC-IfFalse**: | **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. |
| **Context Syntax**  **C ::=**  **| if C then e else e**  **| C op e**  **| v op C**  **| ...** |

|  |  |  |
| --- | --- | --- |
| **Precondition** – Text above the line in a rule. | **Context Rule for Binary Op:** | **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{<S*ubs*ectionName>}**  **\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**  **applyMaybe :: Maybe a -> (a -> Maybe b)**  **-> (Maybe b)**  **applyMaybe Nothing \_ = Nothing**  **applyMaybe (Just x) f = Just (f x)** | **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** | **Additional Names for Monoids**   * “Programmable Semicolons” * “Applicative Functors you can chain.” |

|  |  |  |
| --- | --- | --- |
| **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** | **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)** |

|  |  |  |
| --- | --- | --- |
| **Maybe as an Instance of the Monad Typeclass**  **instance Monad Maybe where**  **return = Just**  **(>>=) Nothing \_ = Nothing**  **(>>=) (Just x) f = f x**  **fail \_ = Nothing** | **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)** | **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”**  **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”**  **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**  **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**  **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**  **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**  **> 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** – ***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** |
| 1. **True** 2. **False** – Lazy evaluation 3. **False** – Lazy evaluation 4. **False** – Statically type 5. **True** | 1. **True** 2. **False** – Applicative functor 3. **True** 4. **True** 5. **True** | 1. **False** – Big step 2. **True** 3. **False** – Use store 4. **True** 5. **False** | 1. **False** – Imperative 2. **True** 3. **False** 4. **True** 5. **True** | 1. **True** 2. **False** – Typeclass 3. **True** 4. **False** 5. **False** – Algebraic data type |

|  |  |  |  |
| --- | --- | --- | --- |
| **Haskell**   * **Purely Functional** * **Lazy evaluation** * **Fully Curried Language** * **Statically Typed** * **Type Inference** – Via context, Haskell can deduce the type. | **Purely Functional**   * **Referential Transparency** – A **function call** can be replaced with its equivalent value without affecting the program * **No (re)assignment** * **No loop** * **No side effects** | **Functional Languages**   * **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** | **Operational Semantics**   * **Small Step** – Structural Semantics * **Big Step** – Natural Semantics * “**Get stuck**” – When a function is encountered that does not have an associated rule. |

**CSV Parser Example**

|  |  |
| --- | --- |
| **Verbose Approach**  **import Text.ParserCombinator.Parsec**  **import System.Environment**  **csvFile :: GenParser Char st [[String]]**  **csvFile = do**  **arr <- many line**  **char eof**  **return arr**  **line :: GenParser Char st [String]**  **line = do**  **result <- many1 cell**  **char ‘\n’**  **return result**  **cells :: GenParser Char st [String]**  **cells = do**  **firstCell <- cellContents**  **nextCells <- remainingCells**  **return (firstCell:nextCells)**  **cellContent :: GenParser Char st String**  **cellContent = many $ noneOf “,\n” -- Two characters**  **remainingCells :: GenParser Char st [String]**  **remainingCells = do**  **(char “,” >> cells)**  **<|> return []**  **main = do**  **args <- getArgs**  **p <- parseFromFile csvFile (head args)**  **case p of**  **Left msg -> error msg**  **Right csv -> print csv** | **Concise Approach**  **import Text.ParserCombinator.Parsec**  **import System.Environment**  **csvFile = line `sepBy` eol**  **line = cell `sepBy` string ","**  **cell = many (noneOf "\n")**  **eol = try (string "\n\r") -- Try more complex case first**  **<|> string "\n"**  **<?> "end of line"**  **main = do**  **args <- getArgs**  **p <- parseFromFile csvFile (head args)**  **case p of**  **Left msg -> error msg**  **Right csv -> print csv** |

**Miscellaneous**

|  |  |  |
| --- | --- | --- |
| **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** | **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. |
| **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. |

|  |  |  |
| --- | --- | --- |
| **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, L**ambda 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). | **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.** |
| **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**. |

**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)** |

**Lecture #12 – Introduction to JavaScript**

|  |  |  |  |
| --- | --- | --- | --- |
| **JavaScript**   * Developed at Netscape by Brendan Eichs in 10 days * Originally named “Mocha” * Syntax similar to Java | **Multi-paradigm 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**.  **this** – Not optional in JavaScript. | **Example**: **Quirks of 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.   **Example: Closure in JavaScript**  **var getNextInt = function (){**  **var nextInt = 0;**  **return function(){**  **return nextInt++;**  **}**  **}(); // Double paren to run the func**  **console.log(getNextInt()); // print “0”**  **console.log(getNextInt()); // print “1”**  **console.log(getNextInt()); // print “2”** | **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. |
| **JavaScript supports both “null” and “undefined”** |

|  |  |  |
| --- | --- | --- |
| **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**” | **Creating a JavaScript Object**  **var myDog = {age : 3,**  **weight: 100}**  **Every object is a map.**  **Adding a Field to a JavaScript Object**  **myDog[‘height’] = 45 // Add a new height field**  **// Note the single quotes**  **Adding a Function to a JavaScript Object’s Prototype**  **myDog.speak = function(){ console.log(“Grr”); }**  **Delete a Function from a JavaScript Object’s Prototype**  **delete myDog.speak** |
| **Undeclared Object Fields**  **Any undeclared object fields or uninstantiated variables are undefined**.  **var y; // Uninstantiated**  **// Both print ‘undefined’**  **console.log(y)**  **console.log(myDog.name)** |

**Prototypes**

|  |  |  |
| --- | --- | --- |
| **Object Prototypes**  **JavaScript prototypes are just like any other object.**  **var dogPrototype = {**  **speak: function(){**  **console.log(“bark!”);**  **}**  **}** | **Defining an Object’s Prototype**  **var** rex = { name: “Rex”,  **\_\_proto\_\_** : dogPrototype} | **Prototypical Inheritance:** If an object does not have a method of field, JavaScript looks to the object’s **\_\_proto\_\_** object. |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Top Prototypes**  **Object.prototype** – Top of all object prototypes  **Function.prototype** – Top of all function prototypes. | **Iterating Using “forEach”**  **var arr = [1, 2, 3];**  **// Print each element in array**  **arr.forEach(function(val){**  **console.log(val);**  **});**  **Note:** This uses parentheses not curly brackets. | **require**   * Used to **import an external module** in **Node.js** * Can be stored in a variable. **Example**:   **var net = require('net');** | **Running from the Command Line**   * Use the keyword “**node**” for Node.js. **Example**:   **$ node my\_program.js** |

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

**Lecture #13 – Lambda Calculus**

|  |  |  |  |
| --- | --- | --- | --- |
| **Expressions**  **e ::= x (Variables, immutable)**  **| λ x.e (Lambda abstraction)**  **| e e (Function application)**  **Note:** Lambda (λ) is simply a function.  **v ::= (λ x.e) (Lambda abstraction)** | **Function Application**  Given a function where **E** is a **complex expression**:  Then:  Hence, “” replaces “” in “”. | **Lambda Calculus is a simple,** **Turing complete** **language**. Hence it is equal in power to a Turing Machine. |  |
| Lambda calculus stops evaluating when the result is in **normal form**. |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Rule: SS-E1** | **Rule: SS-E2** | **Rule:** **SS-Lambda Context** | **Optional Rule:** **Lazy** **SS-Lambda Context** |

**Evaluation Strategies**

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Language Equivalents of** | | **True and False in Lambda Calculus** | |
| **JavaScript:**  **function(x){return e;}** | **Haskell:**  **(\x -> e)** | **True in Lambda Calculus:**  **Note:** This returns the ***first*** parameter in the pair of values. | **True in Lambda Calculus:**  **Note:** This returns the ***second*** parameter in the pair of values. |

|  |  |
| --- | --- |
| **Conditional in Lambda Calculus** | |
| **Example #1:** | **Example #2:** |

|  |  |
| --- | --- |
| **Boolean And** | **Pair**  **Pair** – A tuple-like data structure in Lambda Calculus. |

**Working with a Pair in Lambda Calculus**

|  |  |
| --- | --- |
| **First Element in a Pair** | **Second Element in a Pair** |
| **Note #1:** In the case of both and , the term must be a pair.  **Note #2:** Both of these rely on the or being substituted for the “” in the data structure in term selecting either the first or second element. | |

|  |  |  |
| --- | --- | --- |
| **Church Encoding Numerals** | **Successor Function**  **Example:** | **Plus in Lambda Calculus**  **Example: Use church encoding numerals** |

|  |  |  |
| --- | --- | --- |
| **Omega – Infinite Loop** | **Fix Combinator**  **Note:** **Can be used to do factorial operations.**  **Usage:** |  |

**Lecture #14 – JavaScript Scoping**

|  |  |  |  |
| --- | --- | --- | --- |
| **Example: First Class Function**  **function makeAdder(x){**  **return function(y){**  **return x + y;**  **};**  **}**  **var addOne = makeAdder(1);**  **// Prints "11"**  **console.log(addOne(10));** | **Example: Function Application**  **function makeAdderList(arr){**  **var i;**  **var output = [];**  **for(i = 0; i < output.length; i++){**  **// Need to create a new scope**  **function(){**  **// Can add to arr without append**  **output[i] = function(y){**  **return arr[i] + y;**  **};**  **}**  **}**  **}**  **JavaScript lacks block scope** for the closure to be right, must create the function inside another function. | **Block Scope** – The scope (i.e. visibility) of a variable is limited to a specific block (e.g., **for** loop, **if** statement, etc.).   * Unlike most languages, JavaScript does not have block scope. * **To create a new scope, use an anonymous function**.   **Variable Hoisting** – All variable declarations (i.e., use of “**var**”) are treated as if they are at the beginning of the function. | **“this” in JavaScript**  **this** – Refers to the scope where the function is called.   * **In Normal Function Calls** – **this** refers to the global “**this**” * **Object Methods** – The object itself. * **Constructor (using “new”)** – The newly created object. * **Exceptions:** **apply**, **call**, and **bind**. Inline event handles on DOM elements   **Any time a new function is created, the other** “**this**” **is no longer in scope** |

|  |  |  |
| --- | --- | --- |
| **Execution Context**  Consists of three part:   * **A Variable Object** – **Container for variables and functions**. * **Scope Chain** – Variable object plus **parent scopes** * **Context Object** – **this** | **Global Context**   * **Top Level Context** * Variable object is known as the “**global object**” * **this** – Refers to the global object.   **Any variable declared without** **var** **is added to the global context.** | **Function Contexts**   * **Activation** or **Variable Objects** which include:   + **Arguments passed to the function**   + **A special arguments object**   + **Local variables** |

|  |  |  |
| --- | --- | --- |
| **apply, bind, call Example**  **x = 3;**  **function foo(y) {**  **console.log(this.x + y);**  **}**  **foo(100); // Prints "103"**  **// Array passed for args**  **foo.apply(null, [100]);**  **// Update the context**  **foo.apply({x:4}, [100]);**  **// No array needed**  **foo.call({x:4}, 100);**  **// Create a new function**  **var bf = foo.bind({x:5});**  **bf(100);** | * **apply** – Calls a function with the arguments passed as an array. * **call** – Calls a function with the arguments passed in comma separated. * **bind** – Used to create a new function with a custom context. |  |

**Lecture #14.5 – JSLint and TypeScript**

|  |  |  |
| --- | --- | --- |
| **Issues in JavaScript**   * **No block scope** * **Forgetting** **var** **can lead to** **unexpected behavior since variables become global**. * **Operator** “**==**” **is not transitive**. * **Switch/case statements require** “**break**” | **JavaScript Automatically Inserts Semicolons**  **function makeObject() {**  **return // Semicolon inserted here**  **{**  **madeBy: 'Austin Tech. Sys.'**  **}**  **}**  **var o = makeObject();**  **console.log(o.madeBy); // error** | **Function "parseInt" can Yield Unexpected Results**  **// Drops the " tons"**  **console.log("what do you get? "**  **+ parseInt("16 tons"));**  **// Prints just "1" due to the "Oh"**  **parseInt("1O1");** |

|  |  |  |
| --- | --- | --- |
| **Behavior of "typeOf"**  **typeOf** – Returns a string. **May yield unexpected results**.  **typeOf 5 // "number”**  **typeOf “hi” // “string”**  **typeOf NaN // “number”**  **typeOf null // “object”** | **Behavior of "typeOfChar"**  **typeChar** – Returns a string. **Classifies letters as “digits”**.  **typeOfChar "5" // "digit”**  **typeOfChar "q" // “digit”**  **// "Other character"**  **typeOfChar " "** | **JSLint**   * A tool to write cleaner and safer JavaScript. * Requires that **"use strict";** (**with quotes and followed by semicolon**) **be added at the beginning of all functions**. * **Performs static code analysis**. * **Helps catch common programming errors by requiring:**   + **Variables declared before they are used**.   + **Semicolons are always used**.   + **Double equals never used**. * **Inspired by the** “**lint**” **tool from C** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Benefits of Type Systems**   * **Tips for compilers** * **Hints for IDEs** * **Enforced documentation** * **Prevent code with errors from running**. | **TypeScript**   * Developed by Microsoft * **Static type checker** for JavaScript. * A new “superset” language of JavaScript with:   + **Type annotations**   + **Classes** * **Compiles to JavaScript** | **Function Type Annotations in TypeScript**  **function greet(person: string){**  **console.log("Hello "**  **+ person);**  **}**  **var user : string = "Vlad";**  **// Prints "Hello Vlad”**  **greet(user);** | **Types in TypeScript**   * **number** (**var pi : number = 3.14**) * **boolean** (**var b : boolean = true**) * **string** (**var greet : string = "hi"**) * **array** (**var lst : number[] = [1, 2]**) * **enum** * **any** (**var a : any = 3; var b : any = "hi"**) * **void** |

|  |  |  |
| --- | --- | --- |
| **TypeScript Class**  **class Employee{**  **name : string;**  **salary : number;**  **constructor(name : string, salary : number){**  **this.name = name;**  **this.salary = salary;**  **}**  **display(){ console.log(this.name); }**  **}**  **var emp = new Employee('Jon', 50000);**  **emp.display();** | **TypeScript Function Example**  **function swap(arr : number[], i : number,**  **j : number) {**  **var tmp : number;**  **tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;**  **}**  **function sortAndGetLargest (arr : number[]) {**  **var tmp : number;**  **var i : number;**  **var j : number;**  **tmp = arr[0]; // largest elem**  **for (i=0; i<arr.length; i++) {**  **if (arr[i] > tmp) tmp = arr[i];**  **for (j=i+1; j<arr.length; j++)**  **if (arr[i] < arr[j]) swap(arr,i,j);**  **}**  **return tmp;**  **}**  **var largest = sortAndGetLargest([99,2,43,8,0,21]);**  **console.log(largest); // Prints 99** | **JSLint Requirements**   * **++** - This should be replaced with “+= 1” * **var** – Each var declaration should be on its own line and must be at the top of the function/file. * **Single line if** **and** **for** **statements** still need curly brackets. |

**Lecture #15 – Event-Based Programming and Cryptocurrencies**

|  |  |  |
| --- | --- | --- |
| **JavaScript Embedded in HTML**  Create a button on a website that prints “Hello” when clicked. **Inline event handlers are considered bad practice**.  **<html>**  **<input**  **type='button'**  **onclick='alert("Hello");'**  **value='Say hi' />**  **</html>** | **Improved JavaScript in HTML**  **Give buttons an “id” and update its “onclick” method**  **<html>**  **<input id='thebutton'**  **type='button'**  **value='Say hi' />**  **<script type="text/javascript">**  **var btn = document.**  **getElementById('thebutton');**  **btn.onclick = function() {**  **alert('Groovy');**  **};**  **</script>**  **</html>** | **Adding an Event Listener**   * If clicking a button should perform multiple functions, then an event listener should be used.   **function sayGroovy(){**  **console.log("Groovy");**  **}**  **// Add an "onclick” event listener**  **btn.addEventListener('click',**  **sayGroovy);**  **// Add another event listener**  **btn.addEventListener('click’,**  **function(){**  **console.log("Bogus”);**  **});** |

|  |  |  |
| --- | --- | --- |
| **Removing an Event Listener**   * Event listeners can be **removed by function name**. **Example**:   **btn.removeEventListener('click’,**  **sayGroovy);** | **Event Emitter**   * Import the “events” module using the syntax   **var ee = require('events').EventEmitter;**   * Used to create event via the keyword “**on**”. **Example**:   **ee.on('die', function(){**  **console.log("Died");**  **});**   * Invoking (emitting) an event using the keyword “emit” **Example**:   **setTimeout( function(){**  **ee.emit('die');**  **}, 100); // in ms** | **Create a TCP Server in Node.js Using Event Listeners**  **var net = require('net');**  **var eol = require('os').EOL;**  **var srvr = net.createServer();**  **// Add an event listener**  **srvr.on('connection', function(client) {**  **client.write('Hello there!' + eol);**  **client.end();**  **});**  **srvr.listen(9000);**  **telnet** – Used to connect to a TCP server on the command line.  **127.0.0.1** – IP address of localhost |
| **Events in JavaScript**   * **JavaScript is single threaded**.   An event must run to completion before the next event handler can run. |

**Cryptocurrencies**

|  |  |  |  |
| --- | --- | --- | --- |
| **Types of Keys**   * **Private Key:** Known only by the *owner* * **Public Key**: Known by everyone | **Digital Signature**   * **Non-Repudiation** – **Involves associating actions or changes to a unique individual**.   + **Solution in Cryptocurrency:** Digital signature. * **Procedure:**   + **Step #1**: Owner encrypts the message with his private key   + **Step #2**: Use the public key to decrypt the message. * **Analogy:** Enclosed Bulletin Board | **Private Key Encryption**  Used to transmit sensitive data to a specific recipient.   * **Procedure**:   + **Step #1**: A user encrypts his data using the recipient’s public key.   + **Step #2**: The intended recipient decrypts the data using his private key. * **Analogy:** A public mailbox. Anyone can put letters in, but only the mailman has the key to open the box. | * **update** – Used to update the signature with the specified message contents. Each signature object can only be updated once. * **hex** – Specifies that the output should be in hexadecimal format. * **Sync** – Ensures that the file read is done immediately without relying on a callback. * **SHA** – “Secure Hash Algorithm” * **RSA** – Signature algorithm |

|  |  |
| --- | --- |
| **Example**: **JavaScript Signer Example**  **var crypto = require('crypto');**  **var fs = require('fs');**  **// Constructor for a "Signer" object**  **function Signer(privKeyFile){**  **this.privKey = fs.readFileSync(privKeyFile).toString('ascii');**  **}**  **// Add a "signMessage" function to the Signer prototype**  **Signer.prototype.signMessage = function(msgFileName){**  **var msg = fs.readFileSync(msgFileName).toString('ascii');**  **var sign = crypto.createSign('RSA-SHA256');**  **return sign.update(msg).sign(this.privKey, 'hex');**  **}** | **Double Spending** – **Spend the same funds in multiple places**.  **Solutions to Prevent Double Spending:**   * **Centralized Authority** – Disadvantages include that the central authority would charge a fee and not everyone trusts central authorities. * **Decentralized Authority** – Broadcast transactions to everyone.   **Ledger** – Used to keep a **history of all transactions** and the funds held by all users. |

|  |  |
| --- | --- |
| **Example**: **JavaScript Verifier Example**  **var crypto = require('crypto');**  **var fs = require('fs');**  **// Constructor for a "Verifier" object**  **function Verifier(publicKeyFile){**  **this.publicKey = fs.readFileSync(privKeyFile).toString('ascii');**  **}**  **// Add a "verifySignature" function to the Verifier prototype**  **Verifier.prototype.verifySignature = function(msgFileName, signature){**  **var msg = fs.readFileSync(msgFileName).toString('ascii');**  **// Create a verifier**  **var ver = crypto.createVerifier('RSA-SHA256').update(msg);**  **// Verify signature matches the hash**  **var legit = ver.verify(this.publicKey, signature, 'hex');**  **return legit;**  **}** | **Bitcoin Mining**   * **Block Chain** – Defines the transaction history.   + **Used to prevent double spending.** * **Proof of Work** – Verification of the block chain. * Miners hash transaction details plus a “proof” (i.e. nonce)   + **Reward**: New bitcoins are mined for the first to find a proof. * **Cost to *Derive* a Proof:** 2N­ where N is the number of the initial bits that must be “0” for the proof to be valid. * **Cost to Verify a Proof:** A single hash * Bitcoin protocol is designed to make mining more profitable than cheating. |

**Attributes of a Good Hash Function**

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

**Lecture #16 – Typed Arith**

|  |  |  |  |
| --- | --- | --- | --- |
| **Benefits of Type Systems**   * Tips for **compilers to make code more efficient**. * Tips for **IDEs and other tools to make writing code easier**. * **Enforced documentation**. * **Prevent code with errors from running**. | **The Typed Arith Language**  **e ::= true**  **| false**  **| 0**  **| succ e**  **| pred e**  **| iszero e**  **| if e then e else e**  **v ::= true**  **| false**  **| nv**  **n ::= 0**  **| succ nv**  **T ::= Bool**  **| Int** | **Good and Bad Typing Systems**   * **Good type systems** prevent “bad” programs from running. * **Bad type systems** prevent valid programs from running. * **Typchecking** – In an expression “typechecks”, the expression is either:   + A value   + An evaluation rule reduces the expression to a different expression. | **Type Safety = Progress + Preservation** |
| **Progress**  A well typed expression does not “get stuck”  **Formal Definition:** Given that **e:T**, then either:   1. **e** is a value 2. There exists an **e'** such that **e -> e'** |
| **Typing Rules**  **Format:**  **e : T**  **Meaning:** Expression e falls into one of two categories:   1. **e evaluates to a value of type T** 2. **Goes into an infinite loop**. |
| **Preservation**  A well-type expression will not change its type during evaluation.  **Formal Definition:** Given that **e:T** and **e -> e'**, then:  **e':T** |

**Lecture #17 – Macros and Sweet.js**

|  |  |  |
| --- | --- | --- |
| **Macros**   * Short for “***macroinstruction***” * Rule specifies how an **input sequence** **maps** to a **replacement sequence**. | **Example: C Preprocessor Example**  **#define PI 3.14159**  **#define SWAP(a,b) {int tmp=a;a=b;b=tmp;}**  **int main(void){**  **int x = 4, y=5, diam = 7;**  **double circum = diam \* PI;**  **SWAP(x,y)**  **}** | **Basic Compiler Structure with C-Style Macros** |
| **Macros in C**   * Performed by a ***preprocessor*** * **Rely on text substitution***.* * Embedded languages like PHP, Ruby, etc. use a similar approach. |
| **C Preprocessor Output**  **int main(void){**  **int x = 4, y = 5, diam = 7;**  **double circum = diam \* 3.14159;**  **{int tmp=x;x=y;y=tmp;}**  **}** |

|  |  |  |
| --- | --- | --- |
| **Problem with C Macros (Input)**  **// Macro should be on one line**  **#define SWAP(a,b) {int tmp=a;**  **a=b;**  **b=tmp;}**  **int main(void){**  **int x = 4, tmp = 5;**  **SWAP(x,tmp)**  **}** | **Hygienic Macro** – Any macro whose expansion is **guaranteed not to cause the accidental capture of** **identifiers**. | **Macros in JavaScript**   * **No standard macro system for JavaScript** * Sweet.js has been gaining interest. * Recently redesigned. |
| **Syntactic Macros**   * Derive from Lisp since Lisp programs are essentially one big AST. * **Work at the level of abstract syntax trees**. * Powerful by expensive. * **Hygiene easier to address at the AST level**. * Essentially a **source-to-source compiler**. | **Sweet.js**   * **Borrows concepts from Racket**. * **Source-to-source compiler** (i.e., transpiler) for JavaScript. * Examples of other JavaScript transpilers**:**   + **TypeScript**   + **CoffeeScript**   + **Dart** (includes its own VM) * Project backed by Mozilla |
| **Problem with C Macros (Output)**  **int main(void){**  **int x = 4, tmp = 5;**  **{ int tmp = x;**  **a = tmp;**  **tmp = tmp;**  **}**  **}**  Hence, a **variable name collision** between the two variables named “**tmp**”. This is known as “**inadvertent variable capture**” |
| **Basic Compiler Structure with Syntactic Macros** | **Invoking Sweet.js**   * From command line:   **$ sjs myfile.js –d out/**   * Compiled files run normally (as shown below for Node):   **$ node out/myfile.js** |

|  |  |  |
| --- | --- | --- |
| **Writing a Swap Function in Sweet.js**  **syntax swap = function(ctx){**  **let innerCtx = ctx.next().value.inner();**  **let first = innerCtx.next().value;**  **// Eat the comma**  **innerCtx.next(); // No need for "value()"**  **// Get the second parameter**  **let second = innerCtx.next().value;**  **return #`var tmp = ${first};**  **${first} = ${second};**  **${second} = tmp;`;**  **}**  **swap(a, b); // Invokes the macro**  **Note #1:** The returned string is preceded by a pound (**#**) sign and is enclosed in backticks (**`**).  **Note #2:** Sweet.js variables are declared with “**let**”. | **Concatenating Multiple Result Strings**  This function squares a set of input variables.  **syntax square = function(){**  **var innerCtx = ctx.next().value.inner();**  **// Start with empty results**  **result = #``;**  **while(let stx of innerCtx){**  **result =**  **result.concat(#`${stx}=${stx}\*${stx};`);**  **// Eat comma**  **// Ignored if no comma present**  **innerCtx.next();**  **}**  **}**  **square(a, b, c); // Invokes the macro**  **Note #1:** Use “**.concat**” to concatenate multiple result strings.  **Note #1:** If a token is not present, “**.next()**” does not cause an error. | **Keywords in Sweet.js**   * **let** – Create a Sweet.js variable. * **ctx.next().value** – Get the next value from the context. * **#`...`** - Used to define a result string. * **concat** – Used to combine two result strings. * **let xxx of yyy** – Iterate over a list of tokens. * **isIndentifier** – Used to check if a Sweet.js variable matches some string. |

|  |  |
| --- | --- |
| **Input JavaScript class Code to be Parsed by Sweet.js**  **class Droid{**  **constructor(name, color){**  **this.name = name;**  **this.color = color;**  **}**  **rollWithIt(it){**  **console.log(this.name + " is rolling "**  **+ "with " + it);**  **}**  **}** | **A class in Sweet.js**  **syntax class = function(ctx){**  **let className = ctx.next().value;**  **let bodyCtx = ctx.next().value.inner();**  **// By default assume empty constructor**  **let construct = #`function() { }`;**  **let result = #``;**  **while( let item of bodyCtx ){**  **// Check if constructor**  **if(item.isIndenifier('constructor')){**  **// Get arguments then function code**  **construct = #`function ${className}**  **${bodyCtx.next().value}**  **${bodyCtx.next().value}`;**  **}**  **else {**  **// Add the function to the class prototype**  **result = result.concat(**  **#`${className}.prototype.${item} =**  **function ${bodyCtx.next.value}**  **${bodyCtx.next.value}`);**  **}**  **// Return the constructor and methods**  **return construct.concat(result);** |

**Lecture #18 – Simply Typed Lambda Calculus**

|  |  |  |  |
| --- | --- | --- | --- |
| **The Typed Arith Language**  **(with Lamda Functions)**  **e ::= true**  **| false**  **| 0**  **| succ e**  **| pred e**  **| iszero e**  **| if e then e else e**  **| λ x.e**  **| e e**  **| x**  **v ::= true**  **| false**  **| i**  **| λ x.e**  **T ::= Bool**  **| Int**  **| T -> T** | **Options for Determining Function Type**   1. **Type Inference** 2. **Require explicit type annotations** | **Omega Combinator**  **Definition:**  **Description:** Results in an infinite loop since the combinator keeps returning itself. Omega **is a valid a valid program supported by untyped lambda calculus**.  **Issue:** If this expression is type checked, it enters an infinite type checking loop which causes it to fail |  |
| **Type Annotation**  **λ x:T.e**  **Note:** **T** is the type of the argument “**x**” |
| **Managing the Type of a Variable**   * Use a **typing environment**.   + **Maps variables to types**   + Referred to using the Greek letter: **Γ** (gamma) * **Typing rules** **must be defined in terms of the typing environment**. |
| **@ Symbol in Haskell**  **@** – Allows the programmer to represent an entire expression using a shortened notation.  **Example:**  **e @ (ESucc e1)** |
| **Turnstile**   * **├** - Used in revised typing relations to indicate that the typing rule is with respect to the specified typing environment (**Γ**) |

|  |  |
| --- | --- |
| **Arith Typing Rules Using a Typing Environment ()** | |
| **[T–True]** | **[T-If]** |
| **[T–False]** |
| **[T–succ]** | **[T-FunctionApplication]**  **Note:** is a function while is a parameter. |
| **[T–pred]** | **[T-LambdaVariable]** |
| **[T–iszero]** | **[T-LambdaContext]** |

**Lecture #19 – Metaprogramming and JS Proxies**

|  |  |  |  |
| --- | --- | --- | --- |
| **Metaprogramming**: Writing programs that manipulate other programs.   * Proposed in ECMAScript 6 for JavaScript.   **Terminology in Reflection**   * **Introspection:** Ability to examine (but not modify) the structure of a program. * **Self-modification**: Ability to modify the structure of a program. | **Introspection**  Ability to **examine** (but not modify) **the structure of a program**.  **JavaScript Examples**  **Property Lookup**  **"x" in o; //o is an object**  **Iterate Over All Properties of an Object**  **for( prop in o ){**  **// Do something**  **...**  **}** | **Self-modification**  Ability to **modify the structure of a program.**  **JavaScript Examples**  **o["x"]; // Computed property**  **o.y = 42; // Add new property**  **delete o.y; // Delete property**  **// Reflected method call**  **o["m"].apply(null, [38]);** | **Proxies in JavaScript**   * **Metacircular Interpretation** – The language is able to understand its own language. * Until recently, JavaScript did not support intercession.   + Javascript proxies are intended to fix that. * **Node.js’ implementation of proxies lags behind the standard**. * **Proxies only exist for objects and functions**.   + **Proxies do not exist for primitives**. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Proxies and Common Lisp**   * Lisp was developed before object oriented languages were popular. * Many libraries were created with non-standard OO systems. * **Common Lisp Object System** (**CLOS**) – Standard object oriented system for Lisp. | **Achieving Lisp Object**  **Backwards Compatibility**  **Option #1:** Rewrite all libraries using CLOS. **Disadvantages:**   * Huge number of libraries. * Not feasible to rewrite them all.   **Option #2:** Make a complex API. **Disadvantages:**   * API difficult to understand. * Systems had conflicting features.   **Option #3:** Keep API simple and modify object behavior to fit different systems.   * This approach relies on **metaobject protocols**. | **Proxies and Handlers**   * The behavior of a proxy is determined by **traps** specified in its **handler** (i.e., the **metaobject**). * **Trap** – **Methods that intercept an operation**. * **Handler** – The metaobject that **specifies the details of the trap**. The handler itself is **usually a normal object**. * Using proxies in node requires a special flag: “**--harmony-proxies**”. **Example:**   **$ node --harmony-proxies prog.js** | **Kinds of JavaScript Proxies**   * **Object Proxies** – Defined with:   **Proxy.create(handler, proto)**   * **Functions** (with extra traps) - Defined with:   **Proxy.createFunction(handler,**  **callTrap,**  **constructTrap)**   * **Proxies do not exist for primitives.** |

|  |  |  |
| --- | --- | --- |
| **A Simple Proxy**  **var MyHandler = {**  **get:function(myProxy, name){**  **console.log(name**  **+ " accessed");**  **return 1;**  **}**  **}**  **var p = Proxy.create(MyHandler);**  **// Prints "hello accessed.”**  **var q = p.hello;**  **// Prints "1"**  **console.log(q);**  **// Error since no "set" handler**  **p.name = "Me";** | **A No-op Proxy – All Operations Passed through Unchanged**  **function handlerMaker(obj){**  **// Delete a property from an object**  **delete : function(name){ return obj[name]; },**  **// Check if object has the specified property**  **has : function(name){ return name in obj;},**  **// Check if object (not prototype chain) has property**  **hasOwn : function(name){return Object.property**  **.hasOwn(obj, name);},**  **// Get a property value**  **get : function(name){ return obj[name]; },**  **// Set a property value**  **set : function(rcvr, name, val){ obj[name] = val; },**  **// Get all properties of an object**  **enumerate : function(){**  **var props = [];**  **var prop;**  **for(prop in obj){ props.push(prop); }**  **return props;**  **},**  **// Get all of the keys of an object**  **keys: function(){ return Object.keys(obj); }**  **}** | **Aspect Oriented Programming**   * **Some code not well organized into objects**. Example:   + **Cross-cutting concern** where code is spread throughout a program. * **Canonical Example:** Logging Statements   + **Littered throughout the code**   + **Swapping out a logger requires massive code changes**. * **Solution:** Use a proxy |
| **Read Only Handler**   * **Information Control** – Share a reference to an object, but do allow it to be modified.   + **Example**: Reference to the DOM.   **function ReadOnlyHandler(obj){**  **delete : function(name){**  **return obj[name];**  **}**  **// rcvr can be ignored**  **set : function(rcvr, name, val){**  **return true;**  **}**  **}** |

**Lecture #20 – Introduction to Ruby**

|  |  |  |  |
| --- | --- | --- | --- |
| **Influences of Ruby**   * **SmallTalk**   + Everything is an object   + Blocks   + Metaprogramming * **Perl**   + Regular Expressions   + Function names | **Basic Ruby Syntax**  **puts "Hello World"**  **a = [1, 2, 3]**  **m = { 'a' => "Apple",**  **'b' => "Bear",**  **'c' => "Cat" }**  **# Prints "1"**  **puts a[0]**  **# Prints "Apple"**  **puts m['a']**  **Keywords**  **@** - Represents an object property | **Basic Ruby Class**  **class Person**  **# Constructor**  **def initialize name # Parameter**  **# Attribute**  **@name = name**  **end**  **# Getter**  **def name**  **return @name**  **end**  **# Setter**  **def name = newName**  **@name = name**  **end**  **# Method**  **def say\_hi**  **puts "Hi my name is #{@name}"**  **end**  **end** | **Using Metaprogramming for Getters and Setters**  **class Person**  **# Replaces getters and setters**  **# Uses metaprogramming**  **attr\_accessor :name**  **# Constructor**  **def initialize name**  **# Attribute**  **@name = name**  **End**  **# Method**  **def say\_hi**  **puts "Hi my name is #{@name}"**  **end**  **end** |
| **Ruby on Rails**   * “Killer” app for Ruby   + **Lightweight web framework**   + “**Convention over configuration**” – If use standard configuration, very little configuration required. * Initial framework was PHP, but that was abandoned. |
| **Returning From a Function**   * **Every function in Ruby returns a value**, even if return is not used. * If no return is specified, a function returns the last used value. |
| **Using a Class in Ruby**  **p = Person.new "Joe"**  **puts "Name is #{p.name}"**  **p.say\_hi**  **#{...}** – Embeds a variable in a Ruby String |

|  |  |  |  |
| --- | --- | --- | --- |
| **Getters and Setters the Ruby Way**  **Relies on metaprogramming**   * **attr\_reader** – Getter only * **attr\_writer**- Setter only * **attr\_accessor** – Getter and setter | **Inheritance in Ruby** | | **Mixin**   * Add features to a class * **Similar to interfaces in Java** with the exception that they **can include functionality**. * **module** – Keyword to define a Mixin. * **include** – Keyword to include a Mixin into a class. |
| **Parent Class**  **class Dog**  **# Parentheses optional**  **def initialize(name)**  **@name = name**  **end**  **def speak**  **puts "#{@name} says bark"**  **end**  **end** | **Child Class**  **class GuardDog < Dog**  **attr\_accessor :breed**  **def initialize(name, breed)**  **# Use parent constructor**  **super(name)**  **@breed = breed**  **end**  **def attack**  **puts "Grrr"**  **end**  **end**  **Note:** Inheritance is doing **using the less than** (**<**) operator. |
| **Reopening a Class in Ruby**   * **Class definitions can be changed during runtime** in Ruby. * This is known as “reopening the class” |

|  |  |  |  |
| --- | --- | --- | --- |
| **Blocks in Ruby**   * **Superficially** **similar to blocks in other languages**. * Create custom control structures. * **Can be represented with** **curly brackets** (**{…}**) or **do**/**end**. | **File IO without Blocks**  **file = File.open('test.txt','r')**  **file.each\_line do |line|**  **puts line**  **end**  **file.close**  **Note #1:** Contains “boilerplate” code of open and closing the file.  **Note #2:** It is possible one **may forget to the close the file**. | **File IO with Blocks**  **File.open('test.txt','r') do |file|**  **file.each\_line { |line|**  **puts line**  **}**  **end**  **Note #1:** Eliminates the “boilerplate” code.  **Note #2**: When using a block (both **do**/**end**, and **curly brackets**), **surround the variable names in pipes** (**|**). | **Example: Mixin**  **# Define the mixin**  **module RevString**  **def to\_rev\_s**  **# Object is implicit**  **to\_s.reverse**  **end**  **end**  **# Reopen the Person Class**  **class Person include RevSting**  **def to\_s**  **# Returns the value**  **@name**  **end**  **end** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Dynamic Code Evaluation (eval)**   * **Executes source code dynamically**   + Code passed as either a string (or a block of code) * **Popular feature in JavaScript**   + Early usage was to convert JSON strings to variables since not supported by JavaScript. * **Source of security concerns**. | **Additional Ruby eval Methods**   * **instance\_eval** – Evaluates code within an object’s body.   + **Access the internals of an object**. * **class\_eval** – Evaluates code within a class’ body.   + **Modifies the class’ definition**. * Takes either a string or block of code. Block of code is more secure. | **Example: Use instance\_eval to Change an Object’s Value**  **# Create with the name Bob**  **bob = Person.new “Bob”**  **# Change his name**  **bob.instance\_eval do**  **@name = "Steve"**  **end**  **# Prints "Steve"**  **puts bob.name** | **Regular Expressions in Ruby**   * **sub** – Replaces the first instance of a string match.   + **To perform the modification in place, must include** an **exclamation point** (**!**) after sub. * **gsub** – Replaces all instance of a string match.   + **To perform the modification in place, must include** an **exclamation point** (**!**) after sub. |

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

**Regular Expression Symbols in Ruby**

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

**Important Syntax in Ruby**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **For Each Loop**  **object.each do |val|**  **...**  **end** | **Create a Mixin**  **module Name**  **...**  **end** | **Return from a Block**  **def block\_name**  **...**  **yield x**  **...**  **end** | **Single Line If Statement**  **x = 5**  **# Does nothing**  **x = 3 if (x > 10)**  **puts x # Prints "5"** | **Ranges**  **# Create list from 1 to 5**  **x = (1..5)**  **Note:** Uses parentheses. |
| **Run Ruby on Command Line**  **irb** – Command line for Ruby similar to GHCi. | **Use a Mixin in a Class**  **class MyClass**  **include MixinName**  **...**  **end** | **Reference a Variable in a String**  **age = 30**  **x = “My age is #{age}”**  **Note:** Surround variable name is **#{**…**}** | **Access Command Line Arguments**  **ARGV[0] #First argument**  **ARGV[1] #Second argument** | **For Loop**  **for i in (0..5)**  **puts i**  **end**  **Note:** This loop runs 6 times since the range is inclusive. |
| **Select Case**  **case x**  **when y**  **...**  **when z**  **...**  **else**  **...**  **end** |  |  |  |  |

**Lecture #21 – Blocks and Messages**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Influence of**  **Smalltalk on Ruby**   * **Everything is an object** * **Blocks** * **Message passing** | **Benefits of**  **Blocks in Ruby**   * **Create custom control structures** * **Eliminate boilerplate code.** * **Ruby blocks are closures**, but **they are different than JavaScript blocks**. | **Example:** **do\_noisy Block**  **def do\_noisy**  **puts "About to call block"**  **yield # Calls block code**  **puts “Just called block**  **end**  **Note:** Called with a **do**/**end** or with **curly brackets**. | **Example: Extend Array Class to Return Lowercase Version of Every Element**  **# Reopen the Array class**  **class Array**  **def each\_downcase**  **self.each do |val|**  **yield val.downcase**  **end**  **end**  **end** | **Example: Using the**  **each\_downcase Block**  **arr = ["Alpha","Beta",**  **"So On"]**  **arr.each\_downcase do |val|**  **puts val**  **end** |

|  |  |  |
| --- | --- | --- |
| **Example: Probabilistic Run Block**  **# Probabilistic Run Block**  **def with\_prob(prob)**  **yield if (Random.rand < prob)**  **end**  **with\_prob 0.42 do**  **puts "Prints 42% of time."**  **end** | **Example: Passing Code to a Block**  **def with\_prob2(prob, &blk)**  **blk.call if (Random.rand < prob)**  **end**  **blk** – Block of code passed to the function.  **Note #1:** **Argument name has an ampersand** (**&**) before it.  **Note #2:** **No ampersand is used when calling the block.** | **Example: Sharing Code Between Blocks**  **def half\_the\_time(prob, &blk)**  **with\_prob2(0.5, &blk)**  **end**  **Note:** **Need to pass argument to the function with the ampersand (&).** |

|  |  |  |
| --- | --- | --- |
| **Example: with\_prob in JavaScript**  **function with\_prob(prob, f){**  **if(Math.random() < prob){**  **return f();**  **}**  **}**  **Note:** The JavaScript implementation relies on **callbacks**. | **Example: Difference Between Ruby and JavaScript Blocks** | |
| **Ruby**  **def coin\_flip**  **with\_prob 0.5 do**  **return "Heads”**  **end**  **return "Tails"**  **end**  **Note:** This returns “Heads” half the time and “Tails” half the time.   * This is because a **return in a Ruby block returns for the entire function**. | **JavaScript**  **function coin\_flip(){**  **with\_prob(0.5, function(){**  **return "Heads";}**  **return "Tails";**  **}**  **Note:** This always returns “Tails”   * This is because even if “with\_prob” runs, the return only occurs within the anonymous function. |

|  |  |  |
| --- | --- | --- |
| **Singleton Classes**   * In Ruby, **every object has its own singleton class**. * This class holds **methods and fields unique to that object**. * This is different from Singleton Objects in design patterns. | **Example: Adding a Property to a *Variable* in *JavaScript***  **function Employee(name, salary){**  **this.name = name;**  **this.salary = salary;**  **}**  **var a = new Employee("Alice", 500);**  **var b = new Employee("Bob", 1000);**  **// Add a signing bonus to "Alice"**  **a.signingBonus = 2000;** | **Example: Adding a Property to an *Object* in *Ruby***  **class Employee**  **attr\_accessor :name,:salary**  **def initialize(name, salary)**  **@name = name**  **@salary = salary**  **end**  **def to\_s**  **@name # No return required**  **end**  **end**  **# Create the Objects**  **a = Employee.new("Alice", 500)**  **b = Employee.new("Bob", 1000)**  **# Access the singleton class of "a"**  **class << a**  **def signing\_bonus**  **2000**  **end**  **end** |
| **Accessing Singleton Classes in Ruby**   * To open an object’s singleton class, **use double less than symbols** (“**<<**”). * Code only added to the specific object being reference. |

|  |  |  |
| --- | --- | --- |
| **Example: Using a Singleton Class to Create Static Methods**  **# Add Static Methods to Employee Class**  **class Employee**  **class << self**  **def get\_employee\_by\_name(name)**  **@employee[name] # No return needed**  **end**  **# Called in constructor**  **def add(emp)**  **puts "Adding #{emp}"**  **# Create map if not exist**  **@employee = Hash.new unless @employee**  **@employee[emp.name] = emp**  **end**  **end**  **end** | **Message Passing**   * Represents **inter-object interaction**. * **Sender Sends**:   + **Method name**   + Data: **Method parameters** (if any) * **Receiver**:   + **Processes the message**   + (***Optionally***) **returns data** * **Receiver may not understand the message**. | **Example: missing\_method in Ruby**  **class Person**  **attr\_accessor :name**  **def initialize(name)**  **@name = name**  **End**  **# Called when method unknown**  **def method\_missing(m)**  **puts "Didn't understand #{m}"**  **end**  **end** |
| **Active Record and Message Passing**   * Relational database tool in Ruby. * Specify fields in the database to be extracted based off method names. **Example:**   **Person.find\_by\_first\_name "John"** |
| **method\_missing**   * Method that is part of every class. Can by overridden.   + **Smalltalk Name:** **doesNotUnderstand**   + **Ruby Name:** **method\_missing** * Invoked whenever an unknown method is called. |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Virtual Machine Overview**   * Code is compiled to bytecode   + **Byte code is low level**   + **Platform independent** * The VM interprets the bytecode | **Supported VM Operations**   * **PUSH** – Adds an argument to the stack * **PRINT** – Pops an argument off the stack and prints it. * **ADD** – Pops two elements off the stack, adds them, and places result on the stack. * **SUB** – Similar to add but for subtraction. If “A” is on the top of the stack and “B” is below it, the result is B – A * **MUL** – Similar to add but with multiplication. | **Compilers vs. Interpreters vs. JIT**   * **Compiler**   + **Efficient execution** * **Interpreter**   + **Runtime flexibility** * **JIT**   + **Efficient execution with runtime flexibility.** | **Just-In-Time Compliers**   * Interpret code * “**Hot**” (i.e., heavily-used) **sections** are **compiled at runtime**. * **Advantages**   + **Speed of compiled code**   + **Flexibility of interpreter** * **Disadvantages**   + **Overhead of compiler and interpreter**   + **Complex implementation** |
| **Scheme**   * Similar to an AST. Uses parentheses. * Relies on a stack. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Dynamic Recompilation**   * JIT pursues aggressive optimizations   + **Makes assumptions about the code**   + **Guard conditions verify assumptions** * **Unexpected cases are interpreted** (i.e., not compiled) * **Can in some corner cases outperform static compilation**. | **Types of JITs**   * **Method Based** – Compile Methods * **Trace Based** – Compile loops | **How to Support JITs for a Language**   * **Option #1:** Build your own JIT.   + Study the latest techniques   + Build large code bases to test.   + Profile the code execution * **Option #2:** Use someone else’s Just-In-Time VM. |  |

**Final Exam Review Notes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Parsec**   * Parser combinator in Haskell. * **Example Question:** Write a grammar at the same level of complexity as the CSV parser. | **JavaScript**   * Prototype based language.   + **Inherit from an object not from a class**.   + Add properties and methods on the fly. * **Closure** – By wrapping an inner function with an outer function, the inner function can encapsulate variables.   + **Have a scope chain.** | **Example: Write a Function that Toggles a Variable Each Time the Function is Called** | |
| **Example #1: Use a Global**  **var b = true;**  **function flip(){**  **if(b) b = false;**  **else b = true;**  **return b;**  **}** | **Example #2: Use a Closure**  **var flip = function(){**  **var b = true;**  **return function(){**  **if(b) b = false;**  **else b = true;**  **return b;**  **};**  **}(); // Needed to call out function** |

|  |  |  |  |
| --- | --- | --- | --- |
| **JavaScript – Multiparadigm Language**   * **Imperative** * **Object Oriented** * **Functional**   + **Supports higher order functions** but **not purely functional** since not immutable. | **JavaScript Scoping**   * **No block scope** * **Variable declaration hoisting to the top of the function**. | **this in JavaScript**   * In a **method**, this refers to the associated object. * In a **non-method**, this refers to the global scope. * **Constructor** (**with new**) – Refers to the object being created. * **DOM for Event Listeners** – Refers to the DOM element. * **apply**, **call**, **bind** – User can define what “**this**” refers to. | **Scope Precedence in JavaScript**  **From highest to lowest precedence**   1. **Variable object**    1. **Arguments pass to the function**    2. **Special arguments object**    3. **Local variables** 2. **Scope Chain** 3. **Global object** (i.e., **this**) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Example: Scope Chain**  **function my\_hello(){**  **var x = 5;**  **// Scope Chain**  **function print\_hello(){**  **console.log("Hello " + x);**  **}**  **print\_hello();**  **}**  **my\_hello();** | **Quirks of JavaScript**   * Semicolon insertion * **typeOf**   + **typeOf** **NaN** – Number   + **typeOf** **null** – Object * **==** Not Transitive   + If **a == b** and **b == c**, it is not guaranteed that **a == c** | **JSLint**   * Designed to catch common JavaScript errors. * Based off lint for the C language. * Performs **static code analysis** | **TypeScript**   * Developed by Microsoft * **Source-to-source compiler** (i.e., transpiler) * **Compiles to JavaScript** * **Provides type annotation and classes**. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Event Based Programming**   * Relies on **listeners** * **Events are placed into an event queue**. * **No concurrency**. * **emit** – Invokes an event * **on** – Registers a listener * **Client-Based Programming** – Often used in GUIs. | **Metaprogramming**   * **Reflection** – Two primary categories. **Both occur at runtime**.   + **Introspection** – Examine program’s execution at runtime.   + **Self-modification** – Modify a program execution at runtime. * **Intercession** – Trigger or control interaction at runtime. * Reflection is more common, but intercession is more powerful. | **Aspect-Oriented Programming**  Designed to address cross-cutting code (i.e., code that is interspersed everywhere in a program. | **Metaobject Protocols**   * **Metaobject** – Any object that can reason about the behavior of other objects.   + **Example:** Proxies * **Handler**   + Type of Meta Object.   + **Defines traps**. * **Trap** – Methods that intercept an operation. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Macros Sweet.js**   * Source-to-source compiler (**transpiler**) * **Hygiene** – **No inadvertent variable capture**. * **Text Substitution Macro** – Works at the lexeme or text level. * **Syntactic Macro** – Works at the Abstract Syntax Tree (AST) level. | **Lambda Calculus**   * **Simple, Touring complete language**. * Based off anonymous functions (lambda) * **Expressions:** * **Values:** | **Evaluation Strategies**   * **Strict Evaluation Strategies**   + **Call by Value** – Pass a **copy** of the parameter.   + **Call by Reference** – **Implicit reference** (e.g., a pointer to the parameter is passed. * **Lazy Evaluation Strategies**   + **Call by Name** – Re-evaluate the expression each time it is needed.   + **Call by Need** – Evaluate when needed and memo-ize the result. | **Advantages and Disadvantages**  **of Type Systems**  **Benefits:**   * Enforced documentation * Tips for IDEs and Developers * Prevent code with errors from running.   **Disadvantage:**   * May prevent valid code from running. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Simply-Type Lambda Calculus**   * Relies on a **typing environment** () * **Not Turing complete**.   **Type Safety – Two Components**   * **Progress** – Valid input continues to evaluation or reaches a value. * **Preservation** – Evaluation does not change the type of an object. | **Influences of Ruby**   * **SmallTalk**   + Everything is an object   + Blocks   + **Metaprogramming** – **method\_missing** and message passing. * **Perl**   + Regular expressions   + Names of functions. | **Goals of Ruby**   * Object oriented scripting language. * Dynamically typed * Interpreted. | **Ruby Features**   * **eval** – Execute a string as code. * **Singleton Classes**   + No relation to singleton objects   + **Class for a single object**.   + Can be used to create static methods in a class. |

|  |  |  |
| --- | --- | --- |
| **Virtual Machine**   * Source compiled to **byte code**. * **Byte code executed by an interpreter**. | **Interpreters vs. Compilers**   * **Interpreter** – Runtime flexibility * **Compiler** – Efficient code * **JIT** – **Benefits** ***and* overhead** of both an interpreter and compiler. | **Just-In-Time Compiler**   * Identify “**hot**” (i.e. frequently run) code. * **Optimize for most common cases and skip corner cases**.   + Guards protect for corner cases which are interpreted. * **May outperform statically compiled code**. |

**var b = true;**

**function flip(){**

**if(b) b = false;**

**else b= true;**

**return b;**

**}**

**var flip = function(){**

**var b = true;**

**return function(){**

**if(b) b = false;**

**else b = true;**

**return b;**

**};**

**};**