**CS252 – Midterm 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-code “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**.   + **An expression can be replaced with its value and nothing will change.** * **Supports type inference.** | **Duck Typing** – Suitability of an object for some function is determined not by its type but by presence of certain methods and properties.   * + **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 except the last   + **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 in** | * 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]** |
| **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** |  |

**Lecture #03 – Operational Semantics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Formal Semantics**  ***Crisply define*** **how the language features work**. | | **Formal Semantic Styles**   * **Operational**   + 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  **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 function arguments no arguments. 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:**  **> foldl (\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.** |  |

**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** indicates the **next place for evaluation**. Example:   **Example**:  is the original expression. | **Rewriting Evaluation Order Rules**  **Context based rules only apply to reducible expressions** (redexs). **Example**:  **EC-IfFalse**: | **Data.Map**   * **Library:** Data.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**. |
| **Context Syntax**  **C ::=**  **| if C then e else e**  **| C op e**  **| v op C**  **| ...** |