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
4. Safety (e.g. security and can errors be caught at compile time)
5. Machine independence
6. Efficiency

Goals almost always conflict

Conflict: Type Systems
- Advantage: Prevents bad programs.
- Disadvantage: Reduces programmer flexibility.

Blub Paradox: Why do I need advanced programming language techniques (e.g. monads, closures, type inference, etc.)? My language does not have it, and it works just fine.

Current Programming Language Issues
- Multi-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

- 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`

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

```
> [1..4]
[1, 2, 3, 4]

> [1,2..6]
[1, 2, 3, 4, 5, 6]

> [1,3..10]
[1, 3, 5, 7, 9]
```

Hello World in Haskell

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

List Examples

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

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

Infinite List Example

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

List Comprehension

- **Based off set notation.**
- **Supports filtering** as shown in second example
- If **multiple variables** (e.g. a, b, c) are specified, **iterates through them like nested for loops**.
- Uses the **pipe** (**|**) operator. **Examples:**

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

```
> [(a, b, c) | a <- [1..10], b <-[1..10],
               c <- [1..10], a^2 + b ^2 == c^2]

 [(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)]
```

A Simple Function

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

> inc 4.5
5.5

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

Type Signature

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

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

Pattern Matching

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

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

Recursion

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

Example:

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

Lab #01 – Max Number

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

Lecture #03 – Operational Semantics

Formal Semantics

Crisply define **how the language features work**.

Abstract Syntax Tree

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

Bool * Language

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

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

Formal Semantic Styles

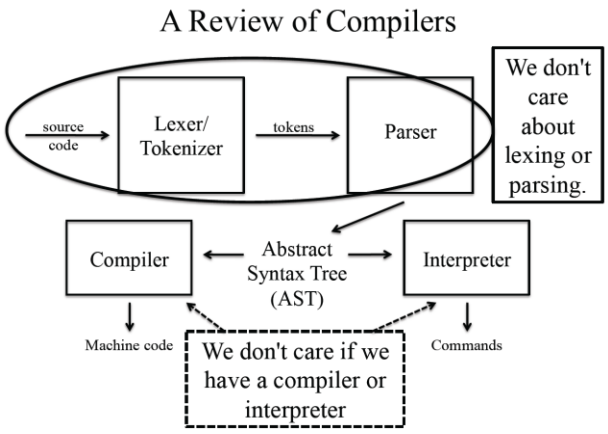
- **Operational**
  - Big-Step ("Natural")
  - Small-Step ("structural")
- **Axiomatic**
- **Denotational**

Big Step Operational Semantics

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

$$e \Downarrow v$$

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



Small-Step Operational Semantics

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

$$e \to e' \to e'' \to v$$

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

$$e \to^* v$$

Bool* Small-Step Operational Semantics Rules

E-IfTrue:

$$\textbf{if } \text{true } \textbf{then } e_2 \textbf{ else } e_3 \to e_2$$

E-IfFalse:

$$\textbf{if } \text{false } \textbf{then } e_2 \textbf{ else } e_3 \to e_3$$

E-If:

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

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

Step #1: Use rule E-IfTrue
if false **then** true **else** false

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

Bool* Extension: Numbers

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

Extended Bool * Language

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

```
v ::= true | false
    | IntV

IntV ::= 0 | succ IntV
```

Literate Haskell

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

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)