# Lecture Notes 15

## Programs as Functions

- Black Box View – Program just converts input to output
- Mathematical Function
  - Range – Considered the output set of the function
  - Domain – Considered the input set of function
  - Independent Variables – The input variables to the function
  - Dependent Variables – The output variables of the function
  - Partial Function – Not all values in Domain are mapped to a value in Range
- Function Definition – Defines how to compute a value given parameters
- Function Application – Call to a function with actual parameters
- Pure Functional – No Side Effects in the Function
  - No Loops – Loops are not possible because requires control variable
  - No Local State
- Value Semantics – Names are only mapped to values
- First-Class Data Values – Functions are viewed as values themselves in functional programming
- Higher-Ordered Functions (or Functional Form) – Takes one or more functions parameters and yields a function
- Functional Composition – Function that takes two functions and returns a function
  $h \equiv f \circ g$
  - Functional Composition Example
    $f(x) = x + 9$
    $g(x) = 2*x$
    $h(x) \equiv f(g(x))$ or $h(x) \equiv (2*x) + 9$
- Apply-to-all – Functional form that takes a function as parameter and applies it to all values in the list parameter
- Referential Transparency – Result only depends upon the value of arguments

## Scheme (Lisp Dialect)

- Metacircular Interpreter – Interpreter written in the language itself
- Scheme EBNF

```
expression → atom | '(' {expression} ')'
atom → number | string | symbol | character | boolean
```

- Applicative Order Evaluation – Sub expressions are evaluated first
- Special Form – Required to control flow of execution (e.g. prevent evaluation of literal list)
- Conditional Expression – Used to control evaluation (some evaluation is delayed)
- Tail Recursion – The recursive step of the function is always the final step
- Accumulating Parameters – Parameters that act like a local variable for accumulation

- All data structures are built from lists
- Free Variable – Variable referenced that is not a formal parameter to that function and not bound within nested function
- Bound Variable – Variable that is a formal parameter of that function

# ML: Functional Programming with Static Typing

- Evaluation Environment – Stores the names of all implicitly and explicitly declared identifiers in a program
- Tuple Type – A Cartesian product of types
- Expression Sequence – Specified evaluation order of expressions
- Value Constructors (or Data Constructors) – Enumerating of data values for constructing a data type
- Currying – A multi-parameter function is viewed as a high-level function that takes in a single parameter and returns a function of remaining parameters
- Fully Curried – Function definitions are treated as curried as well as multi-parameter built-in functions
- Partial Evaluation –

# Haskell

- Pure Functional Language – No side effects are allowed
- Strict Language – All parameters must be evaluated prior to function call
- Non-strict Language – Allows non-strict functions and hence possible lazy evaluation
- Thunks (Pass by name) – Pass a function to allow lazy evaluation
- Memoization – Recalling calculated values instead of recalculating
- Lazy Evaluation – Evaluation is delayed with using of memorization
- Generator-Filter Programming – Separation of generators of data streams and the modification or filtering of the streams

# Mathematics of Functional Programming: Lambda Calculus

- Lambda Abstraction – Analogous to function definition
- Application – Analogous to calling a function
- Reduction Rule – Lambda variables are replaced in expression with applied expression
- Lambda Calculus Syntax
  `exp → constant | variable | (exp exp) | (λ variable . exp)`
- Bound Variable – Lambda variables are bound with scope of expression
- Free Occurrence – Variable outside scope of any binding
- Typed Lambda Calculus – Considers types restricting set of expressions
- Beta-Reduction (Substitution or Function Application) – Similar to calling a function
  $(\lambda\ x\ .\ x\ x)\ y \Rightarrow y\ y$

- Beta-Abstraction – Doing a Beta-Reduction in reverse to abstract it further
- Beta-Conversion – Either Beta-Reduction or Beta-Abstraction
- Name Capture – When a free occurrence is captured (or bound) due to a Beta-Reduction (renaming resolves)
- Alpha-Conversion – Replacing lambda variable for another variable name
  $(\lambda\ x\ .\ +\ y\ x) \Rightarrow (\lambda\ z\ .\ +\ y\ z)$
- Eta-Conversion – Redundant lambda abstraction removal
  $(\lambda\ x\ .\ (+\ 1\ x)) \Rightarrow (+\ 1)$
- $\perp$ (bottom) – Undefined value
- *f* is strict if $(f\ \perp) = \perp$
- Basic Boolean Representations
  - True – $(\lambda\ u\ .\ \lambda\ v\ .\ u)$
  - False – $(\lambda\ u\ .\ \lambda\ v\ .\ v)$
  - Not – $(\lambda\ p\ .\ p\ False\ True)$
  - And – $(\lambda\ p\ .\ \lambda\ q\ .\ p\ q\ p)$
  - Or – $(\lambda\ p\ .\ \lambda\ q\ .\ p\ p\ q)$