Programming in Haskell: Lecture 1

S P Suresh

August 5, 2019

Administrative

- Mondays 10.30 am and Wednesdays 02.00 pm at Seminar Hall

Administrative

- Mondays 10.30 am and Wednesdays 02.00 pm at Seminar Hall
- TAs: Zubin Duggal, Sahil Mhaskar, Dhruv Nevatia

Administrative

- Mondays 10.30 am and Wednesdays 02.00 pm at Seminar Hall
- TAs: Zubin Duggal, Sahil Mhaskar, Dhruv Nevatia
- Moodle page: `https://moodle.cmi.ac.in/course/view.php?id=367`

Administrative

- Mondays 10.30 am and Wednesdays 02.00 pm at Seminar Hall
- TAs: Zubin Duggal, Sahil Mhaskar, Dhruv Nevatia
- Moodle page: `https://moodle.cmi.ac.in/course/view.php?id=367`
- Course page: `https://www.cmi.ac.in/~spsuresh/teaching/prgh19`

Resources

- https://www.haskell.org

Resources

- https://www.haskell.org
- Introduction to Functional Programming using Haskell (Richard Bird)

Resources

- `https://www.haskell.org`
- Introduction to Functional Programming using Haskell (**Richard Bird**)
- Thinking Functionally with Haskell (**Richard Bird**)

Resources

- https://www.haskell.org
- Introduction to Functional Programming using Haskell (Richard Bird)
- Thinking Functionally with Haskell (Richard Bird)
- Real World Haskell http://book.realworldhaskell.org/read/

Resources

- `https://www.haskell.org`
- Introduction to Functional Programming using Haskell (**Richard Bird**)
- Thinking Functionally with Haskell (**Richard Bird**)
- Real World Haskell `http://book.realworldhaskell.org/read/`
- Learn You a Haskell for Great Good!
  `http://learnyouahaskell.com/chapters`

Resources

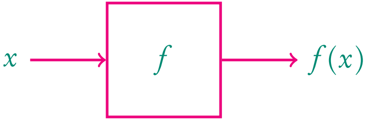
- https://www.haskell.org
- Introduction to Functional Programming using Haskell (Richard Bird)
- Thinking Functionally with Haskell (Richard Bird)
- Real World Haskell http://book.realworldhaskell.org/read/
- Learn You a Haskell for Great Good!
  http://learnyouahaskell.com/chapters
- Haskell Programming: from first principles
  http://haskellbook.com

Resources

- https://www.haskell.org
- Introduction to Functional Programming using Haskell (Richard Bird)
- Thinking Functionally with Haskell (Richard Bird)
- Real World Haskell http://book.realworldhaskell.org/read/
- Learn You a Haskell for Great Good!
  http://learnyouahaskell.com/chapters
- Haskell Programming: from first principles
  http://haskellbook.com
- Plenty of other resources!

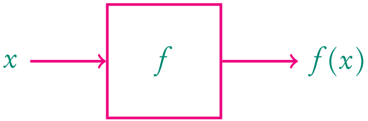
Programs as functions

- Functions transform inputs to outputs



Programs as functions

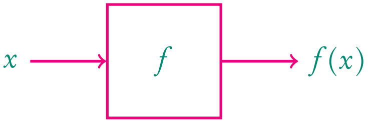
- Functions transform inputs to outputs



- **Program**: rules to produces outputs from inputs

Programs as functions

- Functions transform inputs to outputs



- **Program**: rules to produces outputs from inputs
- **Computation**: process of applying the rules

Building up programs

- How do we describe the rules?

Building up programs

- How do we describe the rules?
    - Start with built-in functions

Building up programs

- How do we describe the rules?
  - Start with built-in functions
  - Use these to build more complex programs

Building up programs

- How do we describe the rules?
  - Start with built-in functions
  - Use these to build more complex programs
- Suppose we have the **natural numbers** `{0, 1, 2, ...}`

Building up programs

- How do we describe the rules?
  - Start with built-in functions
  - Use these to build more complex programs
- Suppose we have the **natural numbers** `{0, 1, 2, ...}`
- ...and the **successor function** `succ`

```
succ 0 = 1
succ 1 = 2
succ 2 = 3
```

Building up programs

- How do we describe the rules?
  - Start with built-in functions
  - Use these to build more complex programs
- Suppose we have the **natural numbers** `{0, 1, 2, ...}`
- ...and the **successor function** `succ`

```
succ 0 = 1
succ 1 = 2
succ 2 = 3
```

- **Note**: We write `succ 0`, not `succ(0)`

Building up programs ...

- We can compose `succ` twice to get a new function

```
plusTwo n = succ (succ n)
```

Building up programs ...

- We can compose `succ` twice to get a new function

```
plusTwo n = succ (succ n)
```

- We can compose `succ` and `plusTwo` to get

```
plusThree n = succ (plusTwo n)
```

Building up programs ...

- We can compose `succ` twice to get a new function

```
plusTwo n = succ (succ n)
```

- We can compose `succ` and `plusTwo` to get

```
plusThree n = succ (plusTwo n)
```

- How do we get `plus` in general? `plus n m` applies the `succ` function `n` times to `m`

Building up programs ...

- We can compose `succ` twice to get a new function

```
plusTwo n = succ (succ n)
```

- We can compose `succ` and `plusTwo` to get

```
plusThree n = succ (plusTwo n)
```

- How do we get `plus` in general? `plus n m` applies the `succ` function `n` times to `m`
  - **Note**: `plus n m`, not `plus(n, m)`!

Inductive/recursive definitions

- `plus n m` applies the `succ` function `n` times to `m`

```
plus 1 m = succ m
plus 2 m = succ (succ m) = succ (plus 1 m)
plus 3 m = succ (succ (succ m)) = succ (plus 2 m)
...
plus n m = succ (succ (... (succ m)...)) = ??
```

Inductive/recursive definitions

- `plus n m` applies the `succ` function `n` times to `m`

```
plus 1 m = succ m
plus 2 m = succ (succ m) = succ (plus 1 m)
plus 3 m = succ (succ (succ m)) = succ (plus 2 m)
...
plus n m = succ (succ (... (succ m)...)) = ??
```

- How do we capture the general rule for `plus`, for all `n` and `m`?

Inductive/recursive definitions

- `plus 0 m = m`, for every `m`

Inductive/recursive definitions

- `plus 0 m = m`, for every `m`
- `plus 1 m = succ m = succ (plus 0 m)`

Inductive/recursive definitions

- `plus 0 m = m`, for every `m`
- `plus 1 m = succ m = succ (plus 0 m)`
- Assume we know how to compute `plus n m`

Inductive/recursive definitions

- `plus 0 m = m`, for every `m`
- `plus 1 m = succ m = succ (plus 0 m)`
- Assume we know how to compute `plus n m`
- Then `plus (succ n) m = succ (plus n m)`

Inductive/recursive definitions

- `plus 0 m = m`, for every `m`
- `plus 1 m = succ m = succ (plus 0 m)`
- Assume we know how to compute `plus n m`
- Then `plus (succ n) m = succ (plus n m)`
- We thus have the following definition

```
plus 0        m = m
plus (succ n) m = succ (plus n m)
```

Computation

- Unravel the definition

```
  plus 3 7
= plus (succ 2) 7
= succ (plus 2 7)
= succ (plus (succ 1) 7)
= succ (succ (plus 1 7))
= succ (succ (plus (succ 0) 7))
= succ (succ (succ (plus 0 7)))
= succ (succ (succ 7))
= 10
```

Recursive definitions ...

- Multiplication is repeated addition

Recursive definitions ...

- Multiplication is repeated addition
- `mult n m` means applying the `plus` function `n` times to `m`

Recursive definitions ...

- Multiplication is repeated addition
- `mult n m` means applying the `plus` function `n` times to `m`
- We have the following definition

```
mult 0        m = 0
mult (succ n) m = plus m (mult n m)
```

Types

- Functions work with values of a fixed type

Types

- Functions work with values of a fixed type
- `succ` takes a natural number as input and outputs a natural number

Types

- Functions work with values of a fixed type
- `succ` takes a natural number as input and outputs a natural number
- `plus` and `mult` take two natural numbers as input, and produce a natural number as output

Types

- Functions work with values of a fixed type
- `succ` takes a natural number as input and outputs a natural number
- `plus` and `mult` take two natural numbers as input, and produce a natural number as output
- Can define analogous functions for real numbers

Types

- How about `sqrt`, the square root function?

Types

- How about `sqrt`, the square root function?
- Even if the input is a natural number, the output need not be a natural number (or even rational)

Types

- How about `sqrt`, the square root function?
- Even if the input is a natural number, the output need not be a natural number (or even rational)
- Fractions and irrational numbers are wholly different types from natural numbers

Types

- How about `sqrt`, the square root function?
- Even if the input is a natural number, the output need not be a natural number (or even rational)
- Fractions and irrational numbers are wholly different types from natural numbers
- This distinction is important in programming, even though in mathematics, natural numbers are often treated as a subset of the reals

Types

- Other types

Types

- Other types
- Consider the following definition

```
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
```

Types

- Other types
- Consider the following definition

```
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
```

- Inputs and outputs for `capitalize` are letters (or **characters**)

Types

- Other types
- Consider the following definition

```
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
```

- Inputs and outputs for `capitalize` are letters (or **characters**)
- We will be careful to ensure that any function we define has a well defined type

Types

- Other types
- Consider the following definition

```
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
```

- Inputs and outputs for `capitalize` are letters (or **characters**)
- We will be careful to ensure that any function we define has a well defined type
- The function `plus` that adds two natural numbers will be different from another function `plus` that adds two real numbers

Functions have types

- A function that takes inputs of type A and produces output of type B has a type $A \rightarrow B$

Functions have types

- A function that takes inputs of type A and produces output of type B has a type $A \to B$
- In Mathematics, we write $f : S \to T$ for a function with domain S and codomain T

Functions have types

- A function that takes inputs of type A and produces output of type B has a type $A \to B$
- In Mathematics, we write $f : S \to T$ for a function with domain S and codomain T
- A type is a just a set of permissible values

Functions have types

- A function that takes inputs of type A and produces output of type B has a type $A \to B$
- In Mathematics, we write $f : S \to T$ for a function with domain S and codomain T
- A type is a just a set of permissible values
- So $f : S \to T$ says that f is of type $S \to T$

Collections

- It is often convenient to deal with collections of values of a given type

Collections

- It is often convenient to deal with collections of values of a given type
- A list of integers

Collections

- It is often convenient to deal with collections of values of a given type
- A list of integers
- A sequence of characters – **words** or **strings**

Collections

- It is often convenient to deal with collections of values of a given type
- A list of integers
- A sequence of characters – **words** or **strings**
- Pairs of numbers

Collections

- It is often convenient to deal with collections of values of a given type
- A list of integers
- A sequence of characters – **words** or **strings**
- Pairs of numbers
- Such collections are also types of values

Haskell

- A programming language for describing functions

Haskell

- A programming language for describing functions
- A function description has two parts

Haskell

- A programming language for describing functions
- A function description has two parts
- Type – of inputs and outputs

Haskell

- A programming language for describing functions
- A function description has two parts
- Type – of inputs and outputs
- Definition or rule for computing outputs from inputs

Haskell

- A programming language for describing functions
- A function description has two parts
- **Type** – of inputs and outputs
- **Definition** or **rule** for computing outputs from inputs
- Example function

```
sqr :: Int -> Int                   -- Type specification
sqr x = x * x                       -- Computation rule
```

Basic types

- `Int` – Integers

Basic types

- **`Int`** – Integers
    - Operations: `+`, `-`, `*`, `/` (Note: `/` produces **`Float`**)

Basic types

- `Int` – Integers
  - Operations: `+`, `-`, `*`, `/` (Note: `/` produces `Float`)
  - Functions: `div`, `mod`

Basic types

- `Int` – Integers
    - Operations: `+`, `-`, `*`, `/` (Note: `/` produces `Float`)
    - Functions: `div`, `mod`
- `Float` – Floating point ("real numbers")

Basic types

- `Int` – Integers
  - Operations: `+`, `-`, `*`, `/` (Note: `/` produces `Float`)
  - Functions: `div`, `mod`
- `Float` – Floating point ("real numbers")
- `Char` – Characters: `'a'`, `'%'`, `'7'`, …

Basic types ...

- `Bool` – Booleans: `True` and `False`

Basic types ...

- `Bool` – Booleans: `True` and `False`
- Operations: `&&`, `||`, `not`, ...

Basic types ...

- `Bool` – Booleans: `True` and `False`
- Operations: `&&`, `||`, `not`, ...
- Relational operators to compare `Int`s, `Float`s &c.

Basic types …

- `Bool` – Booleans: `True` and `False`
- Operations: `&&`, `||`, `not`, …
- Relational operators to compare `Int`s, `Float`s &c.
- `==`, `/=`, `<`, `<=`, `>`, `>=`

Defining functions

- `xor` (Exclusive or)

Defining functions

- `xor` (Exclusive or)
- Input two values of type **`Bool`**

Defining functions

- `xor` (Exclusive or)
- Input two values of type **`Bool`**
- Check that exactly one of them is **`True`**

```
xor :: Bool -> Bool -> Bool            -- why?
xor b1 b2 = (b1 && (not b2)) || ((not b1) && b2)
```

Defining functions

- `isOrdered`

Defining functions

- `isOrdered`
- Input three values of type **`Int`**

Defining functions

- `isOrdered`
- Input three values of type **`Int`**
- Check that the numbers are in order

```
isOrdered :: Int -> Int -> Int -> Bool
isOrdered x y z = (x <= y) && (y <= z)
```

Running Haskell programs

- Haskell interpreter `ghci`

Running Haskell programs

- Haskell interpreter `ghci`
  - Interactively call built-in functions

Running Haskell programs

- Haskell interpreter `ghci`
  - Interactively call built-in functions
  - Load user-defined Haskell code from a text file

Running Haskell programs

- Haskell interpreter `ghci`
  - Interactively call built-in functions
  - Load user-defined Haskell code from a text file
  - Similar to how Python works

Running Haskell programs

- Haskell interpreter `ghci`
  - Interactively call built-in functions
  - Load user-defined Haskell code from a text file
  - Similar to how Python works
- Download and install the Haskell platform at
  `https://www.haskell.org/platform`

Running Haskell programs

- Haskell interpreter `ghci`
  - Interactively call built-in functions
  - Load user-defined Haskell code from a text file
  - Similar to how Python works
- Download and install the Haskell platform at

  `https://www.haskell.org/platform`
- Available for macOS, Windows, Linux

Using `ghci`

- Create a text file (extension `.hs`) with your Haskell function definitions

Using `ghci`

- Create a text file (extension `.hs`) with your Haskell function definitions
- Run `ghci` at the command prompt

Using `ghci`

- Create a text file (extension `.hs`) with your Haskell function definitions
- Run `ghci` at the command prompt
- Load your Haskell code with

```
:load myfile.hs
```

Using `ghci`

- Create a text file (extension `.hs`) with your Haskell function definitions
- Run `ghci` at the command prompt
- Load your Haskell code with

```
:load myfile.hs
```

- Call functions interactively within `ghci`

Compiling Haskell programs

- Can also write **standalone** Haskell programs

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function
- Example program – `hello.hs`

```
main = putStrLn "Hello there! I'm Haskell"
```

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function
- Example program – `hello.hs`

```
main = putStrLn "Hello there! I'm Haskell"
```

- Compile such programs using `ghc` (the **Glasgow Haskell Compiler**)

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function
- Example program – `hello.hs`

```
main = putStrLn "Hello there! I'm Haskell"
```

- Compile such programs using `ghc` (the **Glasgow Haskell Compiler**)
- `ghc hello.hs` produces the executable `hello`)

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function
- Example program – `hello.hs`

```
main = putStrLn "Hello there! I'm Haskell"
```

- Compile such programs using `ghc` (the **Glasgow Haskell Compiler**)
- `ghc hello.hs` produces the executable `hello`)
- Run the executable by issuing `./hello` from the command line

Compiling Haskell programs

- Can also write **standalone** Haskell programs
- Standalone programs require a `main` function
- Example program – `hello.hs`

```
main = putStrLn "Hello there! I'm Haskell"
```

- Compile such programs using `ghc` (the **Glasgow Haskell Compiler**)
- `ghc hello.hs` produces the executable `hello`)
- Run the executable by issuing `./hello` from the command line
- We will concentrate on `ghci` for most of the course