Programming in Haskell: Lecture 3

S P Suresh

August 14, 2019

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

- `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        -- why?
isOrdered x y z = (x <= y) && (y <= z)
```

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: `if-then-else`

- `xor` (Exclusive or)

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 == b2 then False else True
```

Defining functions: `if-then-else`

- `xor` (Exclusive or)

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 == b2 then False else True
```

- **`if - then - else`** is an expression, not a statement!

Defining functions: `if-then-else`

- `xor` (Exclusive or)

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 == b2 then False else True
```

- `if - then - else` is an expression, not a statement!
- The `then` and `else` cases define values

Defining functions: `if-then-else`

- `xor` (Exclusive or)

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 == b2 then False else True
```

- **`if - then - else`** is an expression, not a statement!
- The **`then`** and **`else`** cases define values
- Akin to `cond?expr1:expr2;` in languages like C

Defining xor

- xor can also be defined as:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 /= b2 then True else False
```

Defining xor

- xor can also be defined as:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = if b1 /= b2 then True else False
```

- Can be simplified to:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = b1 /= b2
```

Defining xor

- Another version:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (/=) b1 b2
```

Defining `xor`

- Another version:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (/=) b1 b2
```

- Haskell operators, even infix operators, are just functions!

Defining xor

- Another version:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (/=) b1 b2
```

- Haskell operators, even infix operators, are just functions!
- They can be used like functions (prefix) by enclosing in parentheses

Defining xor

- Another version:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (/=) b1 b2
```

- Haskell operators, even infix operators, are just functions!
- They can be used like functions (prefix) by enclosing in parentheses
- `(*) x y` is the same as `x * y`

Defining xor

- Another version:

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (/=) b1 b2
```

- Haskell operators, even infix operators, are just functions!
- They can be used like functions (prefix) by enclosing in parentheses
- `(*) x y` is the same as `x * y`
- Yet another version:

```
xor :: Bool -> Bool -> Bool
xor = (/=)
```

Pattern matching

- Multiple definitions (for various patterns of input):

```
xor :: Bool -> Bool -> Bool
xor False False = False
xor False True  = True
xor True  False = True
xor True  True  = False
```

Pattern matching

- Not always feasible to list all inputs

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor b1    b2    = False
```

Pattern matching

- Not always feasible to list all inputs

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor b1    b2    = False
```

- Use first definition that matches, from top to bottom

Pattern matching

- Not always feasible to list all inputs

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor b1    b2    = False
```

- Use first definition that matches, from top to bottom
  - `xor False True` matches first definition

Pattern matching

- Not always feasible to list all inputs

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor b1    b2    = False
```

- Use first definition that matches, from top to bottom
  - `xor False True` matches first definition
  - `xor True True` matches third definition

Pattern matching

- When does a function call match a definition?

Pattern matching

- When does a function call match a definition?
- If argument in the definition is a constant

Pattern matching

- When does a function call match a definition?
- If argument in the definition is a constant
  - value supplied in the function call must be the same constant

Pattern matching

- When does a function call match a definition?
- If argument in the definition is a constant
  - value supplied in the function call must be the same constant
- If argument in the definition is a variable

Pattern matching

- When does a function call match a definition?
- If argument in the definition is a constant
  - value supplied in the function call must be the same constant
- If argument in the definition is a variable
  - any value supplied in the function call matches, and is substituted for the variable (**the usual case**)

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

- `xor False True` and `xor False False` match first definition

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

- `xor False True` and `xor False False` match first definition
- `xor True False` matches second definition

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

- `xor False True` and `xor False False` match first definition
- `xor True False` matches second definition
- `xor True True` matches third definition

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

- `xor False True` and `xor False False` match first definition
- `xor True False` matches second definition
- `xor True True` matches third definition
- The argument `b` is used on the RHS in the first two definitions

Pattern matching

- Can mix constants and variables in a definition

```
xor :: Bool -> Bool -> Bool
xor False b     = b
xor b     False = b
xor b1    b2    = False
```

- `xor False True` and `xor False False` match first definition
- `xor True False` matches second definition
- `xor True True` matches third definition
- The argument `b` is used on the RHS in the first two definitions
- `b1` and `b2` are ignored on the RHS in the third definition

Pattern matching

- A better version:

```
xor :: Bool -> Bool -> Bool
xor False b   = b
xor True  b   = not b
```

Pattern matching

- A better version:

```
xor :: Bool -> Bool -> Bool
xor False b   = b
xor True  b   = not b
```

- `b` is used on the RHS in both definitions

Pattern matching: wildcards

- Another example:

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor _     _     = False
```

Pattern matching: wildcards

- Another example:

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor _     _     = False
```

- The symbol _ denotes a “don’t care” argument

Pattern matching: wildcards

- Another example:

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor _     _     = False
```

- The symbol _ denotes a “don’t care” argument
- Any value matches this pattern

Pattern matching: wildcards

- Another example:

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor _     _     = False
```

- The symbol _ denotes a "don't care" argument
- Any value matches this pattern
- The value is not captured, cannot be reused

Guarded definitions

- Use guards to selectively enable a definition

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == b2 = False
    | b1 /= b2 = True
```

Guarded definitions

- Use guards to selectively enable a definition

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == b2 = False
    | b1 /= b2 = True
```

- Definition has two parts

Guarded definitions

- Use guards to selectively enable a definition

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == b2 = False
    | b1 /= b2 = True
```

- Definition has two parts
- Each part is guarded by a conditional expression

Guarded definitions

- Use guards to selectively enable a definition

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == b2 = False
    | b1 /= b2 = True
```

- Definition has two parts
- Each part is guarded by a conditional expression
- Value of the function is the first expression whose guard evaluates to `True`

Guarded definitions

- Use guards to selectively enable a definition

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == b2 = False
    | b1 /= b2 = True
```

- Definition has two parts
- Each part is guarded by a conditional expression
- Value of the function is the first expression whose guard evaluates to `True`
- **Syntax:** Note the indentation

Guarded definitions

- Can mix pattern matching and guards

```
xor :: Bool -> Bool -> Bool
xor False b2      = b2
xor True b2
    | b2 == False = True
    | b2 == True  = False
```

Guarded definitions

- Can mix pattern matching and guards

```
xor :: Bool -> Bool -> Bool
xor False b2      = b2
xor True b2
    | b2 == False = True
    | b2 == True  = False
```

- Two definitions

Guarded definitions

- Can mix pattern matching and guards

```
xor :: Bool -> Bool -> Bool
xor False b2      = b2
xor True b2
    | b2 == False = True
    | b2 == True  = False
```

- Two definitions
- Second definition is guarded

Guarded definitions

- Guards may overlap

Guarded definitions

- Guards may overlap
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | b2 == False = b1
```

Guarded definitions

- Guards may overlap
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | b2 == False = b1
```

- For `xor True False`, both guards evaluate to `True`

Guarded definitions

- Guards may overlap
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | b2 == False = b1
```

- For `xor True False`, both guards evaluate to `True`
  - But the result returned is `not False`, since the first guard evaluates to `True`

Guarded definitions

- Guards may overlap
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | b2 == False = b1
```

- For `xor True False`, both guards evaluate to `True`
  - But the result returned is `not False`, since the first guard evaluates to `True`
- For `xor False True`, neither guard evaluates to `True` (runtime error)

```
*** Exception: Non-exhaustive patterns in function xor
```

Guarded definitions

- Always provide a catchall guard at the end

Guarded definitions

- Always provide a catchall guard at the end
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | otherwise   = b2
```

Guarded definitions

- Always provide a catchall guard at the end
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | otherwise   = b2
```

- `otherwise` is defined to be `True`, so the guard always passes

Guarded definitions

- Always provide a catchall guard at the end
- Guards need not be exhaustive

```
xor :: Bool -> Bool -> Bool
xor b1 b2
    | b1 == True  = not b2
    | otherwise   = b2
```

- `otherwise` is defined to be `True`, so the guard always passes
- This RHS applies if all the previous guards fail

Using case

- A final way to define xor

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = case b1 of
              False -> b2
              True  -> not b2
```

Using case

- A final way to define xor

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = case b1 of
                 False -> b2
                 True  -> not b2
```

- Like switch/case in C

Using `case`

- A final way to define `xor`

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = case b1 of
              False -> b2
              True  -> not b2
```

- Like `switch/case` in C
  - Warning: Expression, not a statement!

Using case

- Can have nested cases too!

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = case b1 of
                 False -> case b2 of
                              False -> False
                              True  -> True
                 True  -> not b2
```

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**
    - Number of arguments

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**
  - Number of arguments
- Instead, assume all functions take only one input!

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**
  - Number of arguments
- Instead, assume all functions take only one input!
- Return **functions** if necessary!

Multiple inputs

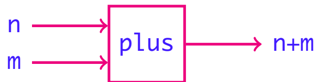
- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**
  - Number of arguments
- Instead, assume all functions take only one input!
- Return **functions** if necessary!
- `plus`, applied to `n`, returns a function that adds `n` to any input

Multiple inputs

- Recall that we write `plus n m`, not `plus(n,m)`
- Normally functions come with an **arity**
  - Number of arguments
- Instead, assume all functions take only one input!
- Return **functions** if necessary!
- `plus`, applied to `n`, returns a function that adds `n` to any input
- This is called **currying**, for the logician **Haskell Curry** (after whom the language is also named)

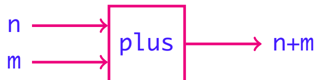
Multiple inputs

- `plus(n,m) = n+m`

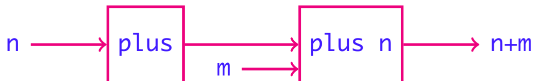


Multiple inputs

- `plus(n,m) = n+m`



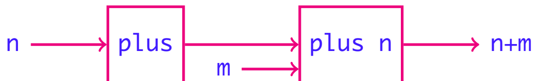
- `plus n m = n+m`



Multiple inputs

- `plus(n,m) = n+m`

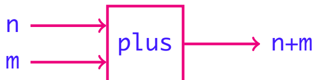
- `plus n m = n+m`



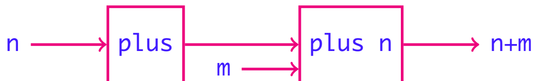
- `plus n` maps `m` to `n+m` `plus n :: Int -> Int`

Multiple inputs

- `plus(n,m) = n+m`



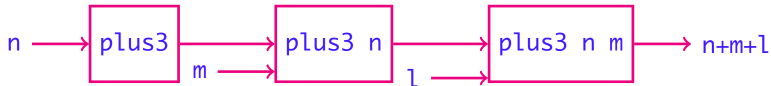
- `plus n m = n+m`



- `plus n` maps `m` to `n+m`   `plus n :: Int -> Int`
- `plus` maps `n` to `plus n`   `plus :: Int -> (Int -> Int)`

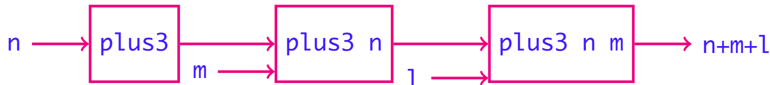
Multiple inputs

- `plus3 n m l = n+m+l`



Multiple inputs

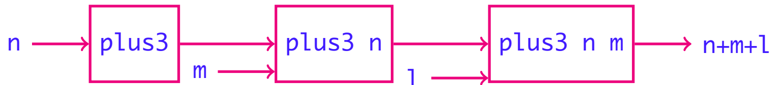
- `plus3 n m l = n+m+l`



- `plus3 n m` maps `l` to `n+m+l` `plus3 n m :: Int -> Int`

Multiple inputs

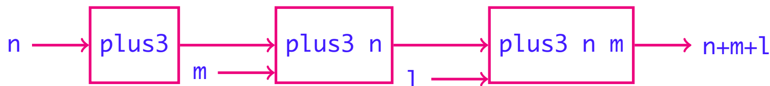
- `plus3 n m l = n+m+l`



- `plus3 n m` maps `l` to `n+m+l` `plus3 n m :: Int -> Int`
- `plus3 n` maps `m` to `plus3 n m` `plus3 n :: Int -> (Int -> Int)`

Multiple inputs

- `plus3 n m l = n+m+l`



- `plus3 n m` maps `l` to `n+m+l` `plus3 n m :: Int -> Int`
- `plus3 n` maps `m` to `plus3 n m` `plus3 n :: Int -> (Int -> Int)`
- `plus3` maps `n` to `plus3 n` `plus3 :: Int -> (Int -> (Int -> Int))`

Multiple inputs

- Consider a function with many arguments

```
f x1 x2 ... xn = e
```

Multiple inputs

- Consider a function with many arguments

```
f x1 x2 ... xn = e
```

- Suppose each `xi` is of type **`Int`**, `e` is of type **`Bool`**

Multiple inputs

- Consider a function with many arguments

```
f x1 x2 ... xn = e
```

- Suppose each `xi` is of type **`Int`**, `e` is of type **`Bool`**
- Then we have:

```
f :: Int -> (Int -> (... (Int->Bool)...)
```

Multiple inputs

- Consider a function with many arguments

```
f x1 x2 ... xn = e
```

- Suppose each `xi` is of type **`Int`**, `e` is of type **`Bool`**
- Then we have:

```
f :: Int -> (Int -> (... (Int->Bool)...)
```

- Correspondingly, we should write

```
(...((f x1) x2) ...) xn = e
```

Multiple inputs

- Consider a function with many arguments

```
f x1 x2 ... xn = e
```

- Suppose each `xi` is of type **`Int`**, `e` is of type **`Bool`**
- Then we have:

```
f :: Int -> (Int -> (... (Int->Bool)...)
```

- Correspondingly, we should write

```
(...((f x1) x2) ...) xn = e
```

- Too many parentheses!

Multiple inputs

- Fortunately, Haskell makes life easy for us

Multiple inputs

- Fortunately, Haskell makes life easy for us
- Implicit bracketing for types is from the right, so

```
f :: Int -> Int -> ... -> Int -> Bool
```

actually means

```
f :: Int -> (Int -> (... ->(Int -> Bool))...)
```

Multiple inputs

- Likewise, function application brackets from left, so

```
f x1 x2 ... xn
```

actually means

```
(...((f x1) x2) ...) xn
```