Programming in Haskell

S P Suresh

http://www.cmi.ac.in/~spsuresh

Lecture 2

August 14, 2017

Haskell

- A programming language for describing functions
- A function description has two parts
  - Type of inputs and outputs
  - Rule for computing outputs from inputs
- Example

```
sqr :: Int -> Int
sqr x = x * x
```

Type definition

Computation rule

Basic types

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

Basic types ...

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

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

- `inorder`
- Input three values of type `Int`
- Check that the numbers are in order
    - ```
      inorder :: Int -> Int -> Int -> Bool
      inorder x y z = (x <= y) && (y <= z)
      ```
      ```

Pattern matching

- Multiple definitions, by cases
    - ```
      xor :: Bool -> Bool -> Bool
      xor True  False = True
      xor False True  = True
      xor b1    b2    = False
      ```
- Use first definition that matches, top to bottom
    - `xor False True` matches second definition
    - `xor True True` matches third definition
      ```

Pattern matching ...

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

```
or :: Bool -> Bool -> Bool
or True  b     = True
or b     True  = True
or b1    b2    = False
```

- `or True False` matches first definition
- `or False True` matches second definition
- `or False False` matches third definition

Pattern matching ...

- Another example

```
and :: Bool -> Bool -> Bool
and True  b = b
and False b = False
```

- In the first definition, the argument `b` is used in the definition
- In the second, `b` is ignored

Pattern matching ...

- Another example

```
and :: Bool -> Bool -> Bool
and True  b = b
and False _ = False
```

- Symbol _ denotes a “don’t care” argument
  - Any value matches this pattern
  - The value is not captured, cannot be reused

Pattern matching ...

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

- Can have more than one _ in a definition

Recursive definitions

- Base case: $f(0)$
- Inductive step: $f(n)$ defined in terms of smaller values, $f(n-1)$, $f(n-2)$, …, $f(0)$
- Example: factorial
  - $0! = 1$
  - $n! = n \times (n-1)!$

Recursive definitions ...

- In Haskell
    - ```
      factorial :: Int -> Int
      factorial 0 = 1
      factorial n = n * (factorial (n-1))
      ```
- Note the bracketing in `factorial (n-1)`
    - `factorial n-1` would be read as
      `(factorial n) - 1`
- No guarantee of termination: what is `factorial (-1)`
      ```

Conditional definitions

- Use conditional expressions to selectively enable a definition
- For instance, "fix" factorial for negative inputs
  - ```
    factorial :: Int -> Int
    factorial 0 = 1
    factorial n
      | n < 0 = factorial (-n)
      | n > 0 = n * (factorial (n-1))
    ```
    ```

Conditional definitions ..

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

- Second definition has two parts
  - Each part is guarded by conditional expression
  - Test guards top to bottom
  - Note the indentation

Conditional definitions ..

- ```
  factorial :: Int -> Int
  factorial 0 = 1
  factorial n
    | n < 0 = factorial (-n)
    | n > 0 = n * (factorial (n-1))
  ```
- Multiple definitions can have different forms
  - Pattern matching for `factorial 0`
  - Conditional definition for `factorial n`
  ```

Conditional definitions ...

- Guards may overlap

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
  | n > 0 = n * (factorial (n-1))
```

Conditional definitions ...

- Guards may not cover all cases
    - `factorial :: Int -> Int`

```
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
```

- No match for `factorial 1`
    - Program error: `pattern match failure: factorial 1`

Conditional definitions ...

- Replace the last guard by `otherwise`

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0  = n * (factorial (n-1))
  | otherwise = factorial (-n)
```

- "Catch all" condition, always true

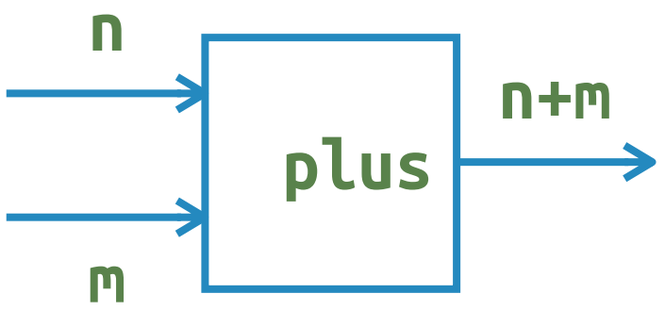
- Ensures that at least one definition matches

Functions with multiple inputs

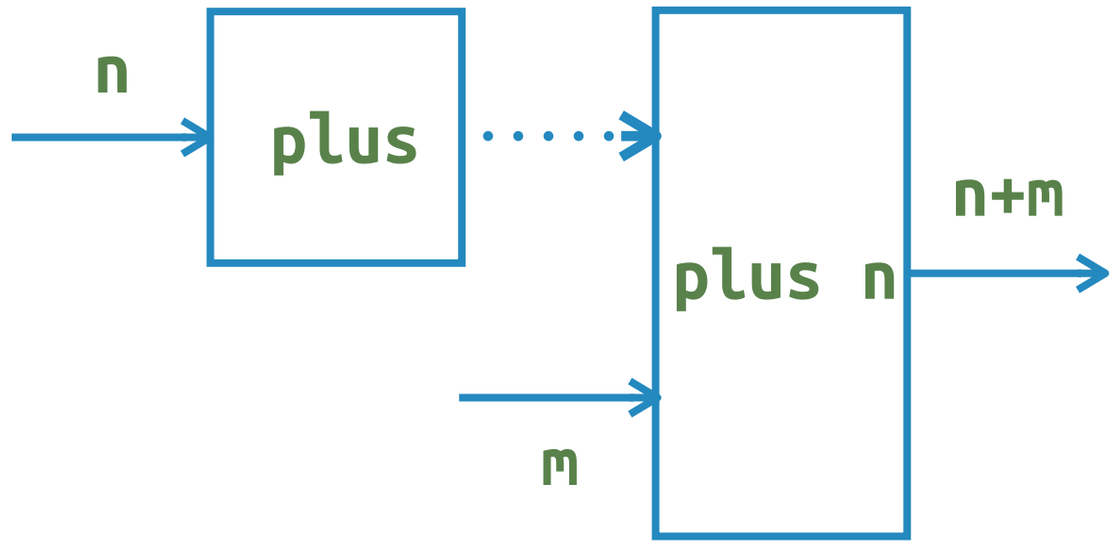
- Recall that we write `plus n m`, not `plus(n,m)`
- Normally, functions come with an arity
  - Number of arguments
- Instead, assume all functions take only one input!
  - This is called **currying**, for the logician **Haskell Curry** (after whom the language is also named)

Multiple inputs ...

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



`plus n m = n+m`



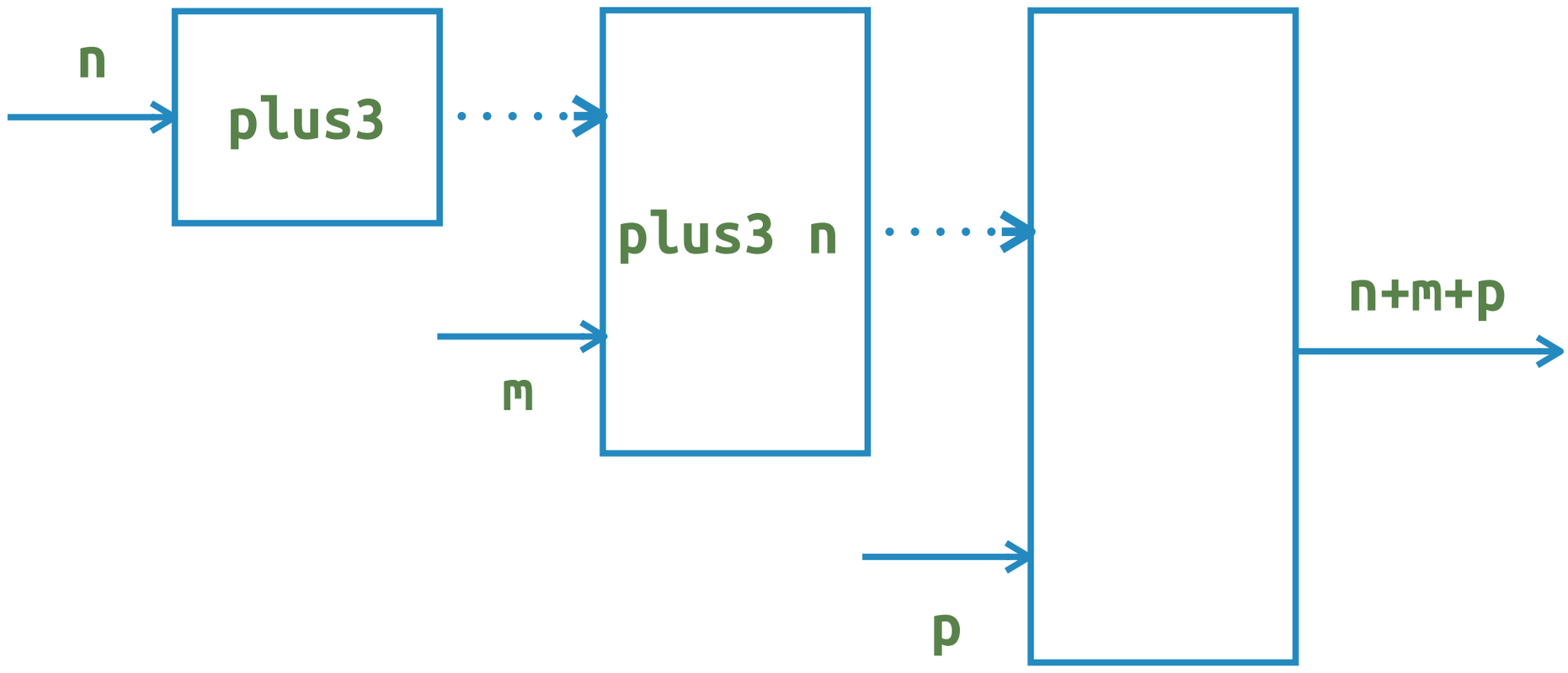
Type of `plus`

`plus n`: input `Int`, output `Int`, so `Int->Int`

`plus` : input `Int`, output `Int->Int`, so `Int->(Int->Int)`

Multiple inputs ...

```
plus n m p = n+m+p
```




```
plus3 : Int -> (Int -> (Int -> Int))
```

Multiple inputs ...

- Consider a function with many arguments

```
f x1 x2 … xn = y
```

- Suppose each `xi` is of type `Int`, `y` is of type `Bool`
- Type of `f` is

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

- Correspondingly, we should write

```
(…((f x1) x2) …) xn = y
```

Multiple inputs …

- Fortunately, Haskell knows this!
- Implicit bracketing for types is from the right, so

```
f :: Int -> Int -> … -> Int -> Bool
```

means

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

Multiple inputs ...

- Likewise, function application brackets from left
- So

  ```
  f x1 x2 … xn
  ```

  means

  ```
  (…((f x1) x2) …) xn
  ```

Summary

- A Haskell function consists of a type definition and a computation rule
- Can have multiple rules for the same function
  - Rules are matched top to bottom
  - Use patterns, conditional expressions to split cases
- Multiple inputs are handled via currying