Introduction to Programming: Lecture 1

K Narayan Kumar

Chennai Mathematical Institute

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

06 Aug 2013

About this course ...

About this course ...

- Learn programming

About this course ...

- Learn programming in Haskell

About this course ...

- Learn programming in Haskell
- Learn to think algorithmically.

About this course ...

- Learn programming in Haskell
- Learn to think algorithmically.

Evaluation

About this course ...

- Learn programming in Haskell
- Learn to think algorithmically.

Evaluation

- About 50% weightage to assignments.
- About 50% weightage to exams.

References

References

- Lecture notes written for this course by Madhavan Mukund. (Available at `http://www.cmi.ac.in/~madhavan/courses/programming08`)

References

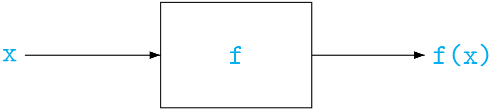
- Lecture notes written for this course by Madhavan Mukund. (Available at `http://www.cmi.ac.in/~madhavan/courses/programming08`)
- Online archive at `http://www.haskell.org`
- Online book at `http://learnyouahaskell.com`

References

- Lecture notes written for this course by Madhavan Mukund. (Available at http://www.cmi.ac.in/~madhavan/courses/programming08)
- Online archive at http://www.haskell.org
- Online book at http://learnyouahaskell.com
- Introduction to Functional Programming using Haskell by Richard Bird.
- A Gentle Introduction to Haskell by Paul Hudak et al.
- Real-world Haskell by Bryan O'Sullivan, John Goerzen and Don Stewart.

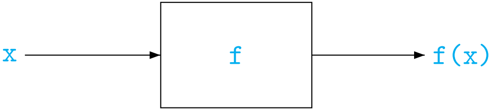
Programs as functions

Functions transform inputs to outputs:



Programs as functions

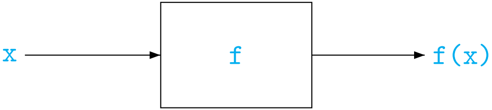
Functions transform inputs to outputs:



A typical program consists of rules to produce an output from an input

Programs as functions

Functions transform inputs to outputs:



A typical program consists of rules to produce an output from an input

Computation is the process of applying the rules described by a program

Building up programs

How do we describe the rules?

Building up programs

How do we describe the rules?

- Start with basic “built in” functions

Building up programs

How do we describe the rules?

- Start with basic "built in" functions
- Use these to build more complex functions

Building up programs . . .

Suppose

Building up programs ...

Suppose

- ...there were the whole numbers, `{0,1,2,...}`

Building up programs . . .

Suppose

- . . . there were the whole numbers, `{0,1,2,...}`
- . . . and one function, `succ` (successor)

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

. . .

Building up programs . . .

Suppose

- . . . there were the whole numbers, `{0,1,2,...}`
- . . . and one function, `succ` (successor)

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

Then, we may define `plusTwo`, as

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

by composing two copies of `succ`.

Building up programs ...

Suspose

- ...there were the whole numbers, `{0,1,2,...}`
- ...and one function, `succ` (successor)

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

Then, we may define `plusTwo`, as

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

by composing two copies of `succ`.

Composing `plusTwo` and `succ` we get

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

Addition....

- `plus n m` means apply `succ` to `n` `m` times

Addition....

- `plus n m` means apply `succ` to `n` `m` times

$$\texttt{plus n m} = \underbrace{\texttt{succ(succ(...(succ}}_{\texttt{m times}}\ \texttt{n)...))}$$

Addition....

- `plus n m` means apply `succ` to `n` `m` times

  $$\texttt{plus n m} = \underbrace{\texttt{succ(succ(...(succ}}_{\texttt{m times}}\ \texttt{n)...))}$$

- How do we describe this rule concisely for all `n` and `m`?

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

  Then `plus n (succ m)` is

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

  Then `plus n (succ m)` is `succ (plus n m)`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

Unravelling the definition yields a computation

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

Unravelling the definition yields a computation

- `plus 7 3` $=$

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

Unravelling the definition yields a computation

- `plus 7 3` = `plus 7 (succ 2)`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

  Then `plus n (succ m)` is `succ (plus n m)`

Unravelling the definition yields a computation

- `plus 7 3` = `plus 7 (succ 2)`

  = `succ (plus 7 2)`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

Unravelling the definition yields a computation

- `plus 7 3` = `plus 7 (succ 2)`
  = `succ (plus 7 2)`
  = `succ (plus 7 (succ 1))`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

Unravelling the definition yields a computation

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

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, ..., `plus n i`, ... for each `i`

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

  Then `plus n (succ m)` is `succ (plus n m)`

Unravelling the definition yields a computation

- `plus 7 3` $=$ `plus 7 (succ 2)`

  $=$ `succ (plus 7 2)`

  $=$ `succ (plus 7 (succ 1))`

  $=$ `succ (succ (plus 7 1))`

  $=$ `succ (succ (plus 7 (succ 0)))`

  $=$ `succ (succ (succ (plus 7 0)))`

  $=$ `succ (succ (succ 7))` $=$ `succ (succ 8)`

  $=$ `succ 9`

Recursive definitions

Goal: Define `plus n 0`, `plus n 1`, . . . , `plus n i`, . . . for each `i`

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

  Then `plus n (succ m)` is `succ (plus n m)`

Unravelling the definition yields a computation

- `plus 7 3` = `plus 7 (succ 2)`

  = `succ (plus 7 2)`

  = `succ (plus 7 (succ 1))`

  = `succ (succ (plus 7 1))`

  = `succ (succ (plus 7 (succ 0)))`

  = `succ (succ (succ (plus 7 0)))`

  = `succ (succ (succ 7))` = `succ (succ 8)`

  = `succ 9` = `10`

Recursive definitions . . .

Multiplication is repeated addition

$$\texttt{mult n m} = \underbrace{\texttt{plus n (plus n (...(plus n}}_{\texttt{m times}}\texttt{ 0)...))}$$

Recursive definitions ...

Multiplication is repeated addition

$$\texttt{mult n m} = \underbrace{\texttt{plus n (plus n (...(plus n}}_{\texttt{m times}}\texttt{ 0)...))}$$

The rule for multiplication

- `mult n 0 = 0`, for all `n`
- `mult n (succ m) = plus n (mult n m)`, for all `n` and `m`

Types

Functions operate on values of a fixed type

Types

Functions operate on values of a fixed type

- `succ` takes a whole number as input and produces a whole number
- `plus` and `mult` take two whole numbers as input and produce another whole number

Types

Functions operate on values of a fixed type

- `succ` takes a whole number as input and produces a whole number
- `plus` and `mult` take two whole numbers as input and produce another whole number

What if we wanted to define `sqrt`, the square root function?

Types

Functions operate on values of a fixed type

- `succ` takes a whole number as input and produces a whole number
- `plus` and `mult` take two whole numbers as input and produce another whole number

What if we wanted to define `sqrt`, the square root function?

- Even if we restrict the input to whole numbers, output will be a real number

Types

Functions operate on values of a fixed type

- `succ` takes a whole number as input and produces a whole number
- `plus` and `mult` take two whole numbers as input and produce another whole number

What if we wanted to define `sqrt`, the square root function?

- Even if we restrict the input to whole numbers, output will be a real number

Other types

- `capitalize 'a' = 'A'`, `capitalize 'b' = 'B'`, ...
- Inputs and outputs are letters or "characters"

Functional programming

Haskell: a programming language for describing functions

Functional programming

Haskell: a programming language for describing functions

A description in Haskell of a function `f` has two parts:

1. Types of inputs and outputs
2. Rule for computing the output from the input

Functional programming

Haskell: a programming language for describing functions

A description in Haskell of a function `f` has two parts:

1. Types of inputs and outputs
2. Rule for computing the output from the input

Example:

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

Basic types and operations in Haskell

- `Int` Integers
  - Operations `+`, `-`, `*`
  - Functions `div`, `mod`
  - Note: `/` takes two `Int`s as input and produces a `Float`
- `Float`
- `Char`
  - Values written in single quotes — `'z'`, `'&'`, ...
- `Bool`
  - Values `True` and `False`.
  - Operations `&&`, `||`, `not`

Defining functions

- Boolean expressions
  - Comparisons on `Int`: `==`, `/=`, `<`, `<=`, `>`, `>=`
  - Boolean combinations `&&` (and), `||` (or) and `not` (negation).

Defining functions

- Boolean expressions
  - Comparisons on `Int`: `==`, `/=`, `<`, `<=`, `>`, `>=`
  - Boolean combinations `&&` (and), `||` (or) and `not` (negation).

- `xor` takes two arguments of `Bool` and checks that exactly one of them is `True`

Defining functions

- Boolean expressions
  - Comparisons on `Int`: `==`, `/=`, `<`, `<=`, `>`, `>=`
  - Boolean combinations `&&` (and), `||` (or) and `not` (negation).

- `xor` takes two arguments of `Bool` and checks that exactly one of them is `True`

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

Defining functions

- Boolean expressions
  - Comparisons on `Int`: `==`, `/=`, `<`, `<=`, `>`, `>=`
  - Boolean combinations `&&` (and), `||` (or) and `not` (negation).

- `xor` takes two arguments of `Bool` and checks that exactly one of them is `True`

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

- `inorder` takes three arguments of `Int` and checks that the numbers are in order

Defining functions

- Boolean expressions
  - Comparisons on `Int`: `==`, `/=`, `<`, `<=`, `>`, `>=`
  - Boolean combinations `&&` (and), `||` (or) and `not` (negation).

- `xor` takes two arguments of `Bool` and checks that exactly one of them is `True`

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

- `inorder` takes three arguments of `Int` and checks that the numbers are in order

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

Definition by cases: Pattern matching

- Defining by pattern matching

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

Definition by cases: Pattern matching

- Defining by pattern matching

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

- When does an invocation match a definition?
  - If definition argument is a constant, the value supplied must be the same constant
  - If definition argument is a variable, any value supplied matches (and is substituted for that variable)

Definition by cases: Pattern matching

- Defining by pattern matching

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

- When does an invocation match a definition?
  - If definition argument is a constant, the value supplied must be the same constant
  - If definition argument is a variable, any value supplied matches (and is substituted for that variable)
- Use first definition that matches, top to bottom
- `xor False True` matches second definition
- `xor True True` matches third definition

Definition by cases: Pattern matching

- Can mix variables and constants in patterns

Definition by cases: Pattern matching

- Can mix variables and constants in patterns

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

Definition by cases: Pattern matching

- Can mix variables and constants in patterns

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

- `or True False` matches first definition

Definition by cases: Pattern matching

- Can mix variables and constants in patterns

```
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

Definition by cases: Pattern matching

- Can mix variables and constants in patterns

```
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

Definition by cases: Pattern matching

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

Recursive definitions

- As we saw earlier, many functions are defined recursively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of n and $f(n-1), \ldots, f(0)$

Recursive definitions

- As we saw earlier, many functions are defined recursively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of n and $f(n-1), \ldots, f(0)$
- For example, factorial
  - $0! = 1$
  - $n! = n \cdot (n-1)!$

Recursive definitions

- As we saw earlier, many functions are defined recursively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of n and $f(n-1), \ldots, f(0)$
- For example, factorial
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- In Haskell

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

Recursive definitions

- As we saw earlier, many functions are defined recursively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of n and $f(n-1), \ldots, f(0)$
- For example, factorial
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- 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 bracketed as `(factorial n) -1`

Recursive definitions

- As we saw earlier, many functions are defined recursively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of n and $f(n-1), \ldots, f(0)$
- For example, factorial
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- 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 bracketed as `(factorial n) -1`
- No guarantee of termination!
  - What does `factorial (-1)` generate?

Functions with multiple inputs

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

- What is the type of `plus`?
  - Mathematically, $\mathit{plus} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
- Need to know arity of functions

Functions with multiple inputs ...

- Assume all functions take only one argument!

Functions with multiple inputs ...

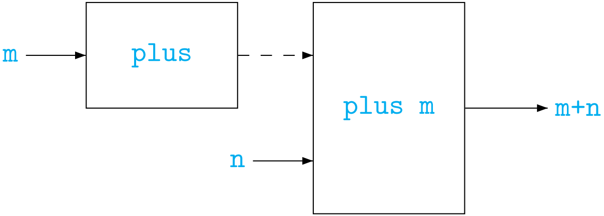
- Assume all functions take only one argument!

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

Functions with multiple inputs . . .

- Assume all functions take only one argument!

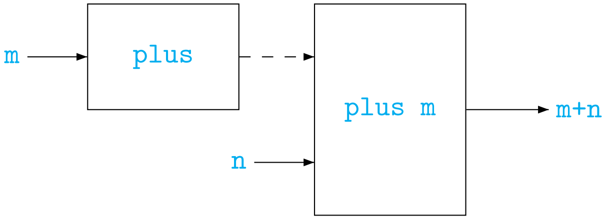
`plus m n = m + n`



Functions with multiple inputs ...

- Assume all functions take only one argument!

  `plus m n = m + n`

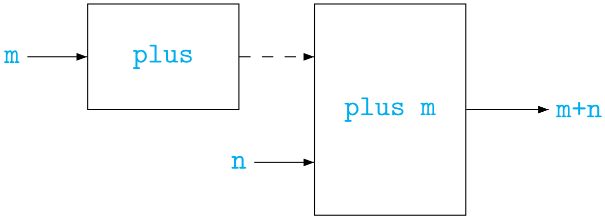


- Type of `plus`
  - `plus m`: input is `Int`, output is `Int`

Functions with multiple inputs . . .

- Assume all functions take only one argument!

  `plus m n = m + n`

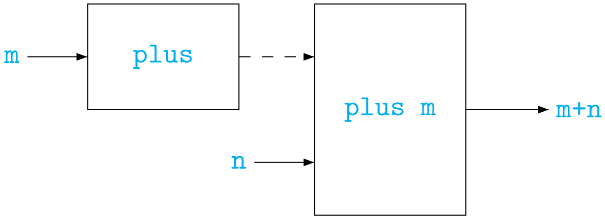


- Type of `plus`
  - `plus m`: input is `Int`, output is `Int`
  - `plus`: input is `Int`, output is a function `Int -> Int`

Functions with multiple inputs ...

- Assume all functions take only one argument!

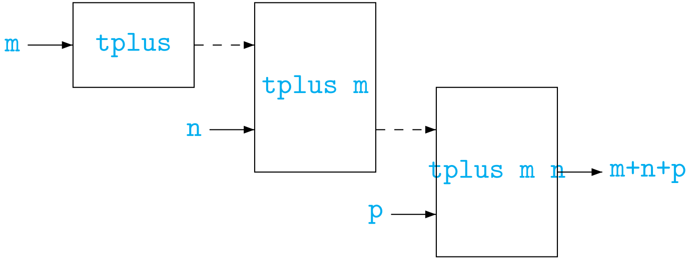
  `plus m n = m + n`



- Type of `plus`
  - `plus m`: input is `Int`, output is `Int`
  - `plus`: input is `Int`, output is a function `Int -> Int`
  - `plus :: Int -> (Int -> Int)`

Functions with multiple inputs . . .

- `tplus m n p = m + n + p`



- `tplus m n p :: Int -> (Int -> (Int -> Int))`

Running Haskell programs

- Write your Haskell program using a text editor
  - `vi`, `Notepad`, ...
- Store it in a file with extension `.hs`

Running Haskell programs

- Write your Haskell program using a text editor
  - `vi`, `Notepad`, ...
- Store it in a file with extension `.hs`
- Use the interactive interpreter `ghci`
- Within `ghci` you can type the following commands:

  `:load filename` — Loads a Haskell file
  `:type expression` — Print the type of a Haskell expression
  `:quit` — exit from `ghci`
  `:?` — Print "help" about more ghci commands

Running Haskell programs

- Write your Haskell program using a text editor
  - `vi`, `Notepad`, . . .
- Store it in a file with extension `.hs`
- Use the interactive interpreter `ghci`
- Within `ghci` you can type the following commands:

  `:load filename` — Loads a Haskell file
  `:type expression` — Print the type of a Haskell expression
  `:quit` — exit from `ghci`
  `:?` — Print "help" about more ghci commands

- Experiment with `ghci` as a "calculator"

Conditional definitions

- Conditional definitions using guards
- For instance, “fix” the function to work for negative inputs

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

Conditional definitions

- Conditional definitions using guards
- For instance, "fix" the function to work for negative inputs

```
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 a condition
  - Guards are tested top to bottom

Conditional definitions

- Conditional definitions using guards
- For instance, "fix" the function to work for negative inputs

```
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 a condition
  - Guards are tested top to bottom
- Indentation to show that definition continues on multiple lines

Conditional definitions

- Conditional definitions using guards
- For instance, "fix" the function to work for negative inputs

```
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 a condition
  - Guards are tested top to bottom
- Indentation to show that definition continues on multiple lines
- Multiple definitions could 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 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))
```

- Guards may not cover all cases

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

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))
```

- 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
```