Functions in Haskell

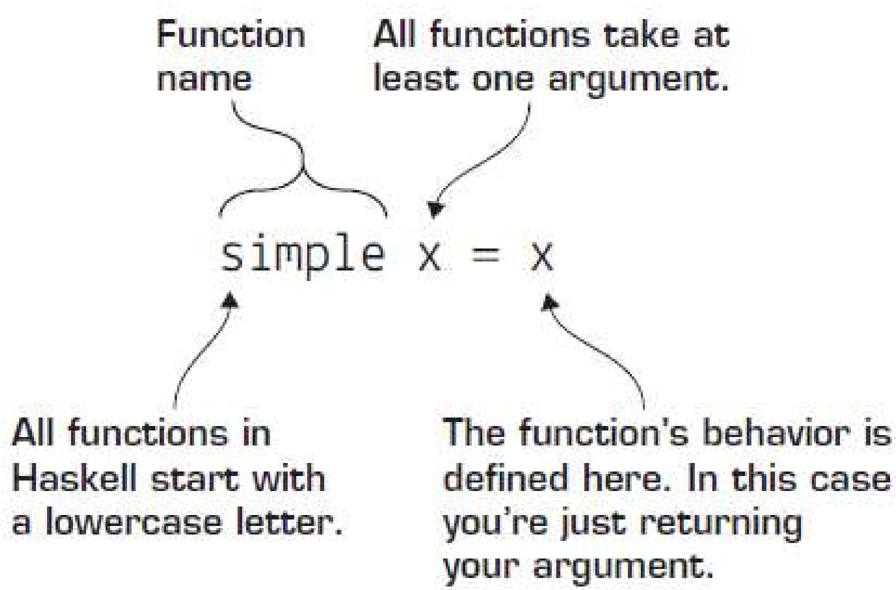
Principles of Programming Languages

What exactly is a function?

- The behavior of functions in Haskell comes directly from mathematics.
- In math, we often say things like $f(x) = y$, meaning there's some function f that takes a parameter x and maps to a value y. [*That is If $f(2) = 2{,}000{,}000$ for a given function f, it can never be the case that $f(2) = 2{,}000{,}001$.*]

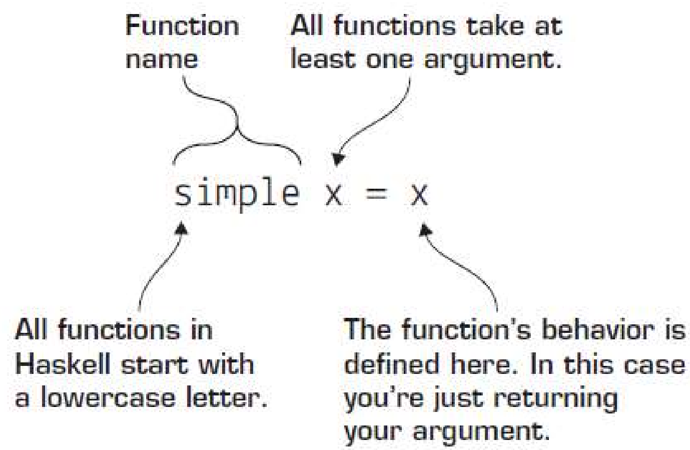
What exactly is a function?

- In Haskell, functions work exactly as they do in mathematics.



Functions in Haskell

- The simple function takes a single argument x and then returns this argument untouched.
- In Haskell you don't need to specify that you're returning a value.
- In Haskell, functions must return a value, so there's never a need to make this explicit.



Functions in Haskell

- To load a function, all you have to do is have it in a file and use

  :load <filename> in GHCi

```
GHCi> simple^2
2
GHCi> simple "dog"
"dog"
```

Functions in Haskell

- All functions in Haskell follow three rules that force them to behave like functions in math
  - All functions must take an argument.
  - All functions must return a value.
  - Anytime a function is called with the same argument, it must return the same value.

- The third rule is part of the basic mathematical definition of a function. When the rule that the same argument must always produce, the same result is applied to function in a programming language, it's called referential transparency.

First-class functions

- The concept of first-class functions is that functions are no different from any other data used in a program.
- Functions can be used as arguments and returned as values from other functions.
- It allows you to abstract out any repetitive computation from your code.
- It allows you to write functions that write other functions.

Functions as values

- Many languages now can treat a function as an ordinary value (like an integer or a string)
  - There is a way to write a "literal" (anonymous) function
  - Functions can be stored in variables and in collections
  - Functions can be passed as parameters to functions
  - Functions can be returned as the result of a function
  - There are operators to combine functions into new functions

Functions in FP languages

- Given a set of input values, a function produces an output value
  - Given the same input values, a function always produces the same output value
    - Functions can use *only* the information provided by their parameters
    - This excludes "functions" that return the date, the time, or a random number
  - Functions have **no side effects**
    - Functions don't do input or output
    - Functions don't change the values of any variables or data
    - Functions *only* return a value
  - Consequently, functions are easier to reason about
    - To understand a function, you need examine only the function itself
    - A function can use other functions, and of course you need to know what those functions are supposed to compute (but nothing about how they do it)
    - In addition, functions can be called in any order, including in parallel

Function Types

- A Function is a mapping or transforming a values of one type -> values in another type.

Type t1 -> Type t2

```
not  :: Bool → Bool

even :: Int → Bool
```

- The arguments and results types are unrestricted.
- Functions with multiple arguments or results are possible using list or tuples.

```
add         :: (Int,Int) → Int
add (x,y)    = x+y

zeroto      :: Int → [Int]
zeroto n     = [0..n]
```

Defining functions in Haskell

- The most basic way of defining a function in Haskell is to ``declare'' what it does. For example, we can write:

```
double :: Int -> Int
double n = 2*n
```

The first line specifies the type of the function, and the second line tells us how the output of double depends on its input.

Defining functions

Consider another example:-

- The successor function that takes an integer value and increments it by one can be defined thus:

```
successor :: Int → Int
successor n = 1 + n
```

- The definition of successor is split into two parts.
  - The first line gives the type of successor.
    - The type denotes that Int -> Int denotes that successor is a function that accepts an integer value and returns an integer value.
  - The second line defines the actual behavior of successor:
    - it states that the function successor applied to an argument n is equal to 1 + n.

Defining functions

```
successor :: Int → Int
successor n = 1 + n
```

- The name n on the left-hand side of = stands for the ***formal argument*** of the function, which is to be replaced by an ***actual argument*** when the function is applied.
- We may now compute the successor of the number 2 thus:

```
> successor 2
3
```

where we write the name of the function to be applied, successor, next to the actual argument to which it is applied, 2.

Defining functions

- Function application has the highest precedence over all the other operators.
- When computing the successor of a compound expression, one should not forget to properly group the expression within parentheses.

```
> successor (2 * 3)
7
```

and

```
> successor 2 * 3
9
```

Example of function definition

Example 1

- To find the reciprocal of a number.

```
recip :: (Fractional a) ⇒ a → a
recip n = 1 / n
```

Example 2

- To test whether an integer number is even or not.
- A function that returns a boolean value is sometimes called *predicate*

```
even :: Int → Bool
even n = if n ‘mod‘ 2 == 0 then True else False
```

or

```
even :: Int → Bool
even n = n ‘mod‘ 2 == 0
```

Conditional Expression

- As in most programming languages, functions can be defined using conditional expressions.

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional Expression

- Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

- In Haskell, conditional expressions must <u>always</u> have an **else** branch, which avoids any possible ambiguity problems with nested conditionals.

Conditional Expression

- if-then-else in Haskell works very similar to other languages.
- Example:-

```
checkNumber :: Int -> String
checkNumber y =
  if (mod y 2) == 0
    then "even"
    else "odd"
```

- Output:-

```
GHCi> checkNumber 10
"even"

GHCi> checkNumber 7
"odd"
```

Guarded Equations

- Haskell gives us a more declarative style of writing functions, by separating the different behaviors and choosing among them by means of ***guards***:
  - As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

As previously but using guarded equations.

Guarded Equations

- Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
absolute :: Int → Int
absolute n = if n >= 0 then n else negate n
```

- Each guard is an expression of type **Bool** preceded by | and followed by =.
- Each guards are tried from top to bottom.

```
absolute :: Int → Int
absolute n | n >= 0 = n
           | n < 0 = negate n
```

Guarded Equations

- It is another way to provide multiple definitions by use of conditional guards. For example:

```
max :: Int -> Int -> Int
max i j  | (i >= j) = i
         | (i < j)  = j
```

In this definition the vertical bar indicates a choice of definitions, and each definition is preceded by a boolean condition that must be satisfied for that line to have effect. If no guards are true, none of the definitions are used. If more than one guard is true, the earliest one is used.

Guarded Equations

- Note - all variables used in patterns are substituted independently-- we cannot directly ``match'' arguments in the pattern by using the same variable for two arguments to implicitly check that they are the same.
- For instance, the following will not work.

```
isequal :: Int -> Int -> Bool
isequal x x = True
isequal y z = False
```

- Instead, we must write

```
isequal :: Int -> Int -> Bool
isequal y z | (y == z) = True
            | (y /= z) = False
```

Otherwise in Guarded expression

- Haskell defines a special guard expression, called **otherwise**

```
absolute :: Int → Int
absolute n | n >= 0 = n
           | otherwise = negate n
```

- **otherwise,** is nothing but an alias for the boolean value **True**:

```
> :type otherwise
otherwise :: Bool
> otherwise
True
```

Special Guard - Otherwise

- When using conditional guards, the special guard otherwise can be used as a default value if all other guards fail, as shown in the following example.

```
max3 i j k | (i >= j) && (i >= k) = i
           | (j >= k)             = j
           |  otherwise           = k
```

- otherwise, is nothing but an alias for the boolean value True:

```
> :type otherwise
otherwise :: Bool
> otherwise
True
```

Pattern matching

- Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.
- Patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether some property of a value (or several of them) are true or false.
- The thing is that guards are a lot more readable works with patterns.

Pattern matching

- Pattern matching can either fail, succeed or diverge.
  - A successful match binds the formal parameters in the pattern.
  - Divergence occurs when a value needed by the pattern contains an error (_|_).
  - The matching process itself occurs "top-down, left-to-right."
  - Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried.
  - If all equations fail, the value of the function application is _|_, and results in a run-time error.
- When defining functions, you can define separate function bodies for different patterns. You can pattern match on any data type – numbers, characters, lists, tuples, etc. This leads to a very expressive code that is also simple and readable.

Defining functions in Haskell

- We are not restricted to having single line definitions for functions.
- We can use multiple definitions combined with implicit pattern matching.
- Consider the function:

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n =  x * (power x (n-1))
```

Here, the first equation is used if the second argument to power is 0. If the second argument is not 0, the first definition does not ``match'', so we proceed to the second definition. When multiple definitions are provided, they are scanned in order from top to bottom.

Defining functions in Haskell

- Another example of a function specified via multiple definitions, using pattern matching. We can use multiple definitions combined with implicit pattern matching.
- Consider the function:

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

Here, the first two lines explicitly describe two interesting patterns, and the last line catches all combinations that do not match.

Pattern matching

- Consider the example :-

```haskell
isValidName :: String -> String
isValidName "" = "It is not a valid name :("
isValidName name = "Valid name: " ++ name
```

This function isValidName behavior in 2 scenarios:

- *receiving an empty string* as the parameter, then returning a message about an invalid name;
- *receiving any string value* — observe that when typed the receiving parameter as a String. If tried to send a Number instead of a String, it would throw an error — , then returning a message including the valid name.

where ?

- Consider

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

- Notice that we repeat weight / height ^ 2 three times.
- It would be ideal if we could calculate it once, bind it to a name and then use that name instead of the expression.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

where ?

- To make it more readable by giving names to things and can make our programs faster since stuff like bmi variable here is calculated only once.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

where ?

- where bindings aren't shared across function bodies of different patterns. If one wants several patterns of one function to access some shared name, it has to be defined it globally.
- Also use where bindings to pattern match! The previous can again be modified as:

```
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

where?

- Consider a function where we get a first and a last name and give someone back their initials.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Next - Recursion