Lambdas in Haskell

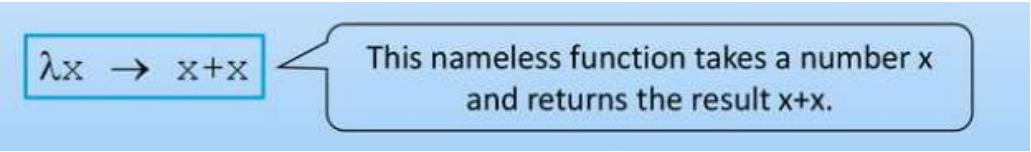
Principles of Programming Languages

Lambdas in Haskell

- Lambdas are basically anonymous functions that are used because we need some functions only once.
- We make a lambda with the sole purpose of passing it to a higher order function.
- To make a lambda, we write a \ because it kind of looks like the Greek letter lambda and then we write the parameters, separated by spaces. After that comes an → and then the function body.

Lambda Expression

- Functions can be constructed without naming the functions by using lambda expressions.



- The symbol λ is the Greek letter lambda and is typed at the keyboard as a backslash \.

General rule for evaluation Lambda

The general rule for evaluating lambda expressions is

$$(\lambda x.\, N)\; M$$

$$=$$

$$(\texttt{let}\; x = M\; \texttt{in}\; N)$$

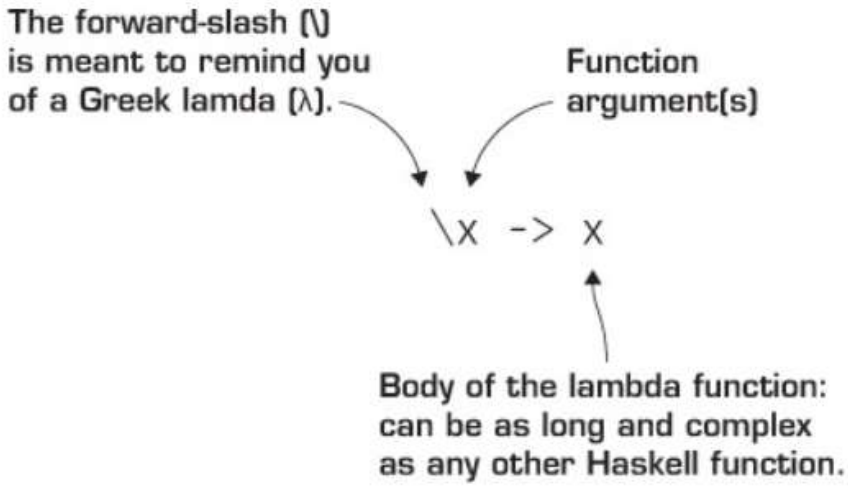
If you have a lambda-expression applied to an argument ...

... replace x by M when evaluating N

This is sometimes called the β rule (or beta rule).

Lambda

- \ is a λ missing a leg: `\ <args> -> <expr>`
- Things like (+ 5) and max 5 are also unnamed functions, but the lambda syntax is more powerful.
- You can give a name (sum) to an expression (2+2): `sum = 2+2`
- But you can also write anonymous expressions — expressions that just appear but are not given names.



Evaluation of Lambda expressions

```
   (\x -> x > 0) 3
=
   let x = 3 in x > 0
=
   3 > 0
=
   True
```

```
   (\x -> x * x) 3
=
   let x = 3 in x * x
=
   3 * 3
=
   9
```

Why are Lambdas useful

- Lambda expressions can be used to give a formal meaning to functions defined using currying.

  For example:

```
add x y  = x + y
square x = x * x
```

  means

```
add    = \x -> (\y -> x + y)
square = \x -> x * x
```

Lambda expressions and currying

```
  (\x -> \y -> x + y) 3 4
=
  ((\x -> (\y -> x + y)) 3) 4
=
  (let x = 3 in \y -> x + y) 4
=
  (\y -> 3 + y) 4
=
  let y = 4 in 3 + y
=
  3 + 4
=
  7
```

Why are Lambdas useful

- Lambda expression can be used to avoid naming functions that are only referenced once. For example

```
odds n = map f [0..n-1]
         where
            f x = x * 2 + 1
```

  can be simplified as

```
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

- Lambda expression can be bound to a name (function argument)

```
incrementer = \x -> x + 1
add (incrementer 5) 6
```

Sections

- SECTIONS are a convenient shorthand for writing partially-applied functions.
- Where x goes depends on where the argument was - x goes in place of the missing argument.
- It's the fact that these functions are curried that makes this work.

`(> 0)` is shorthand for `(\x -> x > 0)`

`(2 *)` is shorthand for `(\x -> 2 * x)`

`(+ 1)` is shorthand for `(\x -> x + 1)`

`(2 ^)` is shorthand for `(\x -> 2 ^ x)`

`(^ 2)` is shorthand for `(\x -> x ^ 2)`

Next – IO Monads