Introduction to Functional Programming

Operators, currying, partial application, and lambda expressions

Some slides are based on Graham Hutton's public slides

Recap previous lecture

GÖTEBORGS UNIVERSITET
CHALMERS

- Building an executable
  - Forgot to mention optimisation! 😬
- Sorting and showing of with QuickCheck
- More list functions
  - With multiple arguments
- Accumulating parameters

- Announcements:
  - Guest lecture on Thursday by prof. John Hughes
  - Only work on exercises on Mondays
  - Presenting lab 2 and 3

Today

GÖTEBORGS UNIVERSITET

CHALMERS

- Defining operators
- Operator fixity and binding precedence
- Currying
- Lambda expressions
- Operator sections

- A larger example: tic-tac-toe

CURRYING

GÖTEBORGS UNIVERSITET
CHALMERS

Function types

- In Haskell a function is a mapping of a value of one type to a value of another type.
  - `a -> b` is the type of functions that map values of type `a` to values of type `b`
  - There are no restrictions on the argument and result types.
- In fact, Haskell *only* supports functions that take *one* argument and have *one* result.
  - These arguments and results can be, for example, tuples and lists, such that we can input and return multiple values.
- However, the result type of a function is not restricted and can therefore be another function!

```
not :: Bool -> Bool

even :: Int -> Bool
```

```
add :: (Int,Int) -> Int
add (x, y) = x+y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried functions

- Functions with multiple arguments are also possible by returning *functions as results.*
- `add` and `add'` produce the same final result, but `add` takes its two arguments at the same time, whereas `add'` takes them one at a time.
- Functions that take their arguments one at a time are called *curried* functions, celebrating the work of Haskell Curry on such functions.

`add'` takes an integer `x` and returns a function `add' x`. In turn, this function takes an integer `y` and returns the result `x+y`

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

```
add :: (Int, Int) -> Int

add' :: Int -> (Int -> Int)
```

GÖTEBORGS UNIVERSITET
CHALMERS

Curried functions

- Functions with more than two arguments can be curried by returning nested functions

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

`mult` takes an integer `x` and returns a function `mult x`, which in turn takes an integer `y` and returns a function `mult x y`, which finally takes an integer `z` and returns the result `x * y * z`

Why is currying useful?

- Curried functions are more flexible than functions on tuples, because useful functions can often be made by *partially applying* a curried function.

```
add' 1 :: Int -> Int

take 5 :: [Int] -> [Int]

drop 5 :: [Int] -> [Int]
```

Currying conventions

- To avoid excess parentheses when using curried functions, two simple conventions are adopted:
  1. The arrow `->` associates to the *right.*
  2. As a consequence, it is then natural for function application to associate to the *left*.

- Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Means `Int -> (Int -> (Int -> Int))`

```
Int -> Int -> Int -> Int
```

```
mult x y z
```

Means `((mult x) y) z`

LAMBDA EXPRESSIONS

GÖTEBORGS UNIVERSITET
CHALMERS

Lambda expressions

- 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 \.
- In mathematics, nameless functions are usually denoted using the $\mapsto$ symbol, as in $x \mapsto x + x$
- In Haskell, the use of the λ symbol for nameless functions comes from the *lambda calculus*, the theory of functions on which Haskell is based.

```
\x -> x + x
```

the nameless function that takes a number `x` as argument and returns the result `x + x`

Why are lambda's useful?

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

```
add x y = x + y
```

means

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

- Lambda expressions can be used to avoid naming functions that are only referenced once.

OPERATOR SECTIONS

GÖTEBORGS UNIVERSITET
CHALMERS

Operator sections

- An operator written *between* its two arguments can be converted into a curried function written before its two arguments by using parentheses.

```
ghci> 1 + 2
3

ghci> (+) 1 2
3
```

- This convention also allows one of the arguments of the operator to be included in the parentheses.

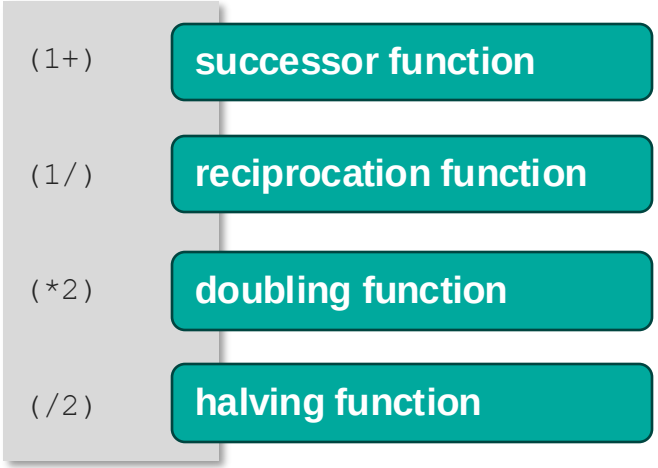
```
ghci> (1 +) 2
3

ghci> (+ 2) 1
3
```

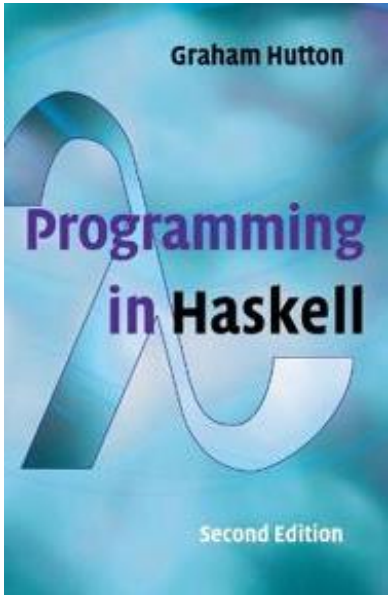
- In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(\oplus\, x)$ and $(x \oplus)$ are called *sections*.

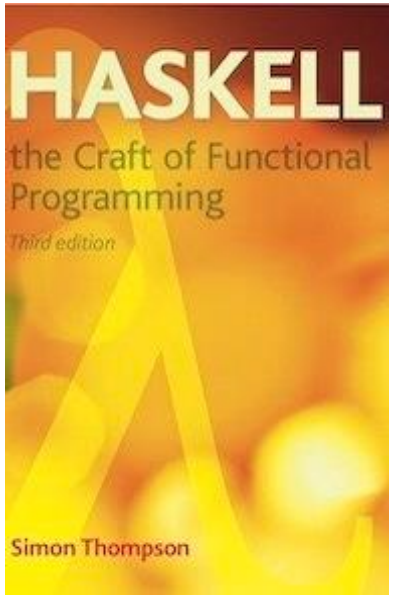
GÖTEBORGS UNIVERSITET
CHALMERS

Why are sections useful?

- Useful functions can sometimes be constructed in a simple way using sections. For example:







GÖTEBORGS UNIVERSITET

CHALMERS

Reading suggestions

- Hutton:
  - Chapters 3.5, 3.6, 4.5, 4.6
- Thompson:
  - The topics are intertwined with higher-order functions, which we will cover next week. Use the reading suggestions for next week.
- Both books offer many exercises!

GÖTEBORGS
UNIVERSITET

CHALMERS