Functional programming

Introduction

- High level of abstraction
- Based on lambda calculus
- Language of choice: Haskell
- Functional programming -> expressions > statements.

Example:

```
--Add the first ten numbers together
sum[1..10]
```

- Install `Hugs`, Haskell interpreter (`ghci` is another interpreter)

First steps

- Start `hugs` in terminal

Examples:

```
> head[1,2,3,4] --take first element
1
> tail[1,2,3,4] --remove first element
[2,3,4]
> [1,2,3,4] !! 2 --element #2
3
> take 3 [1,2,3,4] -- generalization of head
[1,2,3]
> drop 3 [1,2,3,4,5] --generalization of tail
[4,5]
> [1,2,3]++[4,5] --append
[1,2,3,4,5]
```

- First element in list has index 0
- List different to array -> indexing bad idea, not in constant time but in linear

Function application: function application is denoted by space. Higher priority

```haskell
f a b + c*d --f(a,b) + c d from math
f a + b --f(a) + b from math
```

- Haskell file (script) –> `.hs`
- Define function in script, then open Hugs with script as argument so that functions are available. If script is changed use `:reload`. Also possible to load using `:load` *script*
- *Infix operator*: `xfy --> f x y`

- *Naming*:
- function and parameter name must begin with lowercase
- can use quotes (*prime*)
- type has to start with uppercase
- convention –> `s` at the end means list, `ss` list of lists
- Indentation like Python, implicit grouping
- Useful commands –> `:load` *script*, `:reload`, `:edit` *script*, `:type` *expression*,`:?`
- Comments: one line `--comment`, nested:

```haskell
{-
very long
comment goes
here
-}
```