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

```
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:

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

Types and classes

- **Type**: name for a collection of related values. Example `Bool`
- Applying a function to a wrong type makes a *type error*
- `e :: t` –> `e` has type `t`
- *Type inference*: compiler calculates type of expression prior to execution. Haskell programs are *type safe*, type error never happens in run time
- `:type <exp>` to calculate type of expression

Table 1: Different types in Haskell

Type	Explanation
`Bool`	Logical value: `True` or `False`
`Char`	Single character, enclosed in sigle quotes: `'a'`
`String`	String of characters, double quotes: `"abc"`
`Int`	Fixed precision integer
`Integer`	Arbitrary precision integer, doesn't overflow
`Float`	Single precision floating point number

- **List**: sequence of values with same type. Can be infinite. Examples:

```
[False, True, False]::[Bool] -- list of elements type Bool
['a', 'b']:: [Char] -- list of elements type char
```

- **Tuple**: sequence of values of different type. Number of elements is called *arity*. Finite number of elements because type of all have to be calculated. Example:

```
(False, 'a') :: (Bool, Char) -- length appears in type
```

- **Function**: mapping from values of a type to values of another type. Examples:

```
not :: Bool -> Bool
isDigit :: Char -> Bool

function :: t1 -> t2 -- from domain to range in general
```

- *Curried function*: functions that return arguments one at a time (functions can return functions): `a -> (a -> a)` equivalent to `a -> a -> a`, arrow associates to the right. Any function that returns more than one values can be curried. Useful for *partially applying functions*. **Most** functions applied in curried form, if tuples are not explicitly declared.