Haskell Workshop

FINN.no

FINN.no

https://github.com/mariatsji/haskell-workshop

Agenda

- 3 hours
- 7 parts of 15-30 minutes
- Solutions walkthrough for those who want

Haskell

- Purely functional
- Statically Typed
- Lazy
- GHC: Glasgow Haskell Compiler

Stack

- A Build tool
- Runs compiler
- Runs tests
- Manages dependencies
- ..
- See README.md for how to install

Create a function

```haskell
myFunction :: Int -> Int -> Int
myFunction    a      b   =  (a + 1) * b ^ 2
```

Applying the function

```
> myFunction 1 2
8
```

Curried signatures

```
myFunction :: Int -> Int -> Int
              ^arg0  ^arg1  ^res
```

```
myFunction :: Int -> (Int -> Int)
              ^arg0  ^res
```

```
> myPartiallyAppliedFunction = myFunction 1
> myPartiallyAppliedFunction 2
8
```

Pattern matching

```
xor :: Bool -> Bool -> Bool
xor False True  = True
xor True  False = True
xor _     _     = False
```

```
xor :: Bool -> Bool -> Bool
xor a b = a /= b
```

Everything is an expression

```
isNine :: Int -> Bool
isNine i = if i == 9
  then True
  else False
```

You must have an `else`.

Return types must match.

Lists

```
listOfInts :: [Int]
listOfInts  = [1,2,3]
```

```
concat :: [a] -> [a] -> [a]
concat as bs = as ++ bs
```

Recursion on lists

There are no for/while loops in haskell.

```haskell
loopThrough :: [a] -> [a]
loopThrough []       = []
loopThrough (a : as) = a : loopThrough as
```

Recursively looping through a list and changing nothing.

Tips

- Indentation matters
- Slack: #finn-haskell-workshop
- Examples-folder
- README.md
- presentation/summary.pdf
- presentation/index.html

REPL

```
$ stack repl ./src/Part1.hs
(...)
*Part1>
```

Unloading and Loading

Unloading all modules:

```
*Loaded Modules> :load
Ok, no modules loaded.
Prelude>
```

Loading single module:

```
Prelude> :load Part1
Ok, one module loaded.
*Part1>
```

Reloading current modules:

```
*Part1> :reload
Ok, one module loaded.
*Part1>
```

Evaluating values and types

```
>1 + 1
2

>:type 1
1 :: Num a => a
```

Exercise time:

First shell

```
stack repl ./src/Part1.hs
```

Second shell

```
./runtests 1
```

Higher order functions

Functions are values and can be passed as arguments to, and be returned from, other functions.

```
applyTwice :: (a -> a) -> a -> a
applyTwice    f           x =  f (f x)
```

Infix functions 1

You can turn an infix function into a prefix function by wrapping it in parantheses.

This is required to pass it as an argument.

```
(+) 1 2 == 1 + 2
```

Infix functions 2

You can include one operand inside the parantheses to create a partially applied function.

```
(/ 2) 1 == 1 / 2
```

```
(2 /) 1 == 2 / 1
```

let .. in expressions

```
cylinderVolume :: Float -> Float -> Float
cylinderVolume diameter height =
  let radius = diameter / 2
      area = pi * radius ^ 2
  in area * height
```

Variables can not be reassigned.

Exercise time:

First shell

```
stack repl ./src/Part2.hs
```

Second shell

```
./runtests 2
```

Exercise time:

First shell

```
stack repl ./src/Part3.hs
```

Second shell

```
./runtests 3
```

Anonymous functions (lambda)

```
\a b c -> 2 * a + c
```

```
filter (\x -> x ^ 2 > 5) [1,2,3,4]
```

Sometimes more convenient than creating a named function, or partially applying an existing function.

Unused variables

Use an underscore to tell the compiler (and yourself) that an argument is intentionally not used.

```
\a _ c -> 2 * a + c
```

Exercise time:

First shell

```
stack repl ./src/Part4.hs
```

Second shell

```
./runtests 4
```

List data constructor

`(:)` is a function.

```
> :type (:)
(:) :: a -> [a] -> [a]
```

Tuples

Product of two types (which may be different)

```haskell
(,) 'a' 1 == ('a', 1)
```

Exercise time:

First shell

```
stack repl ./src/Part5.hs
```

Second shell

```
./runtests 5
```

Creating a type

```haskell
data TrafficLight = Red | Yellow | Green

safe :: TrafficLight -> Bool
safe Red    = False
safe Yellow = False
safe Green  = True
```

Part 6

- Tests are green
- Keep them green after bumping to Lib.CCLib2
- Expand the datatype Bit as instructed