Style

Unoffical Imperial Haskell Style Guide

This is a set of do's and don'ts for Haskell programming. These

Do's

Use 2 spaces to indent

A where clause or guard should be indented 2 spaces more than its function.

Align the equals in guards

For example:

```
fib :: Int -> Int
fib n = fib' (n - 1) 0 1
  where
    fib' :: Int -> Int -> Int -> Int
    fib' n a b
      | n == 0    = b
      | otherwise = fib' (n - 1) b (a + b)
```

Use helper functions when you want to recurse with more parameters

For example in the Fibonacci function above, the helper needs 2 more parameters than the orignal function, so instead of modifiying the original function's type signature we create a helper function.

Use where clauses instead of let clauses wherever possible

This makes the code easier to read.

Use pattern matching over guards

However don't do this if several pattern matches need the same variable from a where clause.

Use guards over if expressions

Guards are easier to read. However simple if expressions in lambda functions are usually better than creating mini helper functions.

Don'ts

Magic numbers

Don't have random numbers floating around in your code (they make the code harder to read).

Exceptions:

1. Each one has a comment explaining it
2. The number's use is obvious e.g. 0, 1, -1, 2

To avoid these set a constant to these values (your compiler will probably optimise it out anyway).

Redundant brackets

These make code harder to read.

Use variables that are only used once

Except to avoid magic numbers

Use head, tail and null

It is usually better to pattern match than use these.

Exceptions:

1. When applying functions.
2. When switching to pattern matching would make the code less readable.

Reference Sheet for CO120.1 Programming I

Autumn 2016

This reference sheet does not cover many interesting types and classes, such as `Either`, `IO`, `Complex`, and `Monad`. These are not included in the 120.1 syllabus, but even a limited understanding of them may be helpful in examinations.

1 Bools

```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not  :: Bool -> Bool
```

2 Maybes

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Given default value, a function, and `Maybe` value: Returns default value if `Maybe` value is `Nothing`; Otherwise applies function to value inside `Just`.

```
isJust    :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
```

Requires ***`Data.Maybe`***.

```
fromJust :: Maybe a -> a
fromMaybe :: a-> Maybe a -> a
```

Requires ***`Data.Maybe`***.
`fromJust`: Given a `Maybe` value: Returns value inside `Just` or error if `Nothing`.
`fromMaybe`: Given default value and `Maybe` value: Returns value inside `Just` or default value if `Nothing`.

```
catMaybes :: [Maybe a] -> [a]
mapMaybe  :: (a -> Maybe b) -> [a] -> [b]
```

Requires ***`Data.Maybe`***.
`catMaybes`: Given list of `Maybe` values: Returns a list of all `Ju`
`mapMaybe`: Given function from value to `Maybe` value and list of
a list of all `Just` values from mapping function to list of values.

3 Tuples

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

It is usually better to use pattern matching unless used with hig
tions.

```
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: a -> b -> c -> (a, b) -> c
```

`curry`: Given uncurried function $f(x, y)$: Returns curried funct
`uncurry`: Given curried function $f\ x\ y$: Returns uncurried func

```
swap :: (a,b) -> (b,a)
```

Requires ***`Data.Tuple`***.

4 Enums

```
succ :: a -> a
pred :: a -> a
```

`succ`: Given a value: Returns its successor.
`pred`: Given a value: Returns its predecessor.

```
[n..]
[n,n'..]
[n..m]
[n,n'..m]
```


- When returning Bools, just put the condition instead of using guards
- Only place comments when your code is not understandable by itself
- Use u' instead of uDash
- Reduce comments!
- Elem function
- Single space between operator and operands
- Prefer "" to ['']
- :doc
- :l Num
- Always define the signature