# SNAP FRAMEWORK HASKELL STYLE GUIDE

(Adapted from Johan Tibell's style guide.)

This document describes coding and comment style for the Snap projects. Currently we're more interested in building a working web framework than strictly enforcing this style guide; think of these as "aspirational" guidelines. When something isn't covered by this guide you should stay consistent with the style used in our existing code.

## 1. FORMATTING

### LINE LENGTH

Maximum line length is *78 characters*.

### INDENTATION

Tabs are illegal. Use spaces for indenting. Indent your code blocks with *4 spaces*. Indent the `where` keyword two spaces to set it apart from the rest of the code and indent the definitions in a `where` clause 2 spaces. Usually we indent guards two spaces. Some examples:

```
-- | Reads the user's name and prints a greeting to stdout.
sayHello :: IO ()
sayHello = do
    name <- getLine
    putStrLn $ greeting name
  where
    greeting name = "Hello, " ++ name ++ "!"

-- | 'filter', applied to a predicate and a list, returns the list of
-- those elements that satisfy the predicate; i.e.,
--
-- > filter p xs = [ x | x <- xs, p x]
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

### BLANK LINES

Two blank lines between top-level definitions, and a line of "-" characters to delineate top-level definitions from each other. No blank lines between type signatures and function definitions. Add one blank line between functions in a type class instance declaration if the functions bodies are large. Use your judgment.

## WHITESPACE

Surround binary operators with a single space on either side. Use your better judgment for the insertion of spaces around arithmetic operators but always be consistent about whitespace on either side of a binary operator. Don't insert a space after a lambda.

## DATA DECLARATIONS

Vertically align the constructors in a data type definition. Example:

```
data Tree a = Branch a (Tree a) (Tree a)
            | Leaf
```

For long type names the following formatting is also acceptable:

```
data HttpException
    = InvalidStatusCode Int
    | MissingContentHeader
```

Format records as follows:

```
data Person = Person
    { firstName :: String  -- ^ First name
    , lastName  :: String  -- ^ Last name
    , age       :: Int     -- ^ Age
    } deriving (Eq, Show)
```

## PRAGMAS

Put pragmas immediately following the function they apply to. Example:

```
id :: a -> a
id x = x
{-# INLINE id #-}
```

In the case of data type definitions you must put the pragma before the type it applies to. Example:

```
data Array e = Array
    {-# UNPACK #-} !Int
    !ByteArray
```

### HANGING LAMBDAS

You may or may not indent the code following a “hanging” lambda. Use your judgment. Some examples:

```
bar :: IO ()
bar = forM_ [1, 2, 3] $ \n -> do
          putStrLn "Here comes a number!"
          print n

foo :: IO ()
foo = alloca 10 $ \a ->
      alloca 20 $ \b ->
      cFunction a b
```

### EXPORT LISTS

Format export lists as follows:

```
module Data.Set
    (
      -- * The @Set@ type
      Set
    , empty
    , singleton

      -- * Querying
    , member
    ) where
```

## 2. IMPORTS

Imports should be grouped in the following order:

1. standard library imports
2. related third party imports
3. local application/library specific imports

Put a blank line between each group of imports. The imports in each group should be sorted alphabetically, by module name.

Always use explicit import lists or `qualified` imports for standard and third party libraries. This makes the code more robust against changes in these libraries. Exception: The Prelude.

## 3. COMMENTS

### LINE LENGTH

Maximum line length is *78 characters*. This increases readability as the eye has to travel back to the start of the next line.

### PUNCTUATION

Write proper sentences; start with a capital letter and use proper punctuation.

### TOP-LEVEL DEFINITIONS

Comment every top level function (particularly exported functions), and provide a type signature; use Haddock syntax in the comments. Comment every exported data type. Some examples:

```
-- | Send a message on a socket.  The socket must be in a connected
-- state.  Returns the number of bytes sent.  Applications are
-- responsible for ensuring that all data has been sent.
send :: Socket      -- ^ Connected socket
     -> ByteString  -- ^ Data to send
     -> IO Int      -- ^ Bytes sent

-- | Bla bla bla.
data Person = Person
    { age  :: Int     -- ^ Age
    , name :: String  -- ^ First name
    }
```

For functions the documentation should give enough information to apply the function without looking at the function's definition.

### END-OF-LINE COMMENTS

Separate end-of-line comments from the code using 2 spaces. Align comments for data type definitions. Some examples:

```
data Parser = Parser
    Int         -- Current position
    ByteString  -- Remaining input

foo :: Int -> Int
```

```
foo n = salt * 32 + 9
  where
    salt = 453645243  -- Magic hash salt.
```

## 4. NAMING

Use mixed-case when naming functions and camel-case when naming data types.

For readability reasons, don't capitalize all letters when using an abbreviation. For example, write `HttpServer` instead of `HTTPServer`. Exception: Two letter abbreviations, e.g. `IO`.

## 5. MISC

### WARNINGS

Code should be compilable with `-Wall -Werror -fno-warn-unused-binds`. There should be no warnings.