Syntax Cheatsheet

Define values and functions

```
yourLowercaseVariableName :: Int
yourLowercaseVariableName = 1234

yourLowercaseFunctionName :: Text -> Int -> Bool
yourLowercaseFunctionName text int =
    ... (implementation)
```

Apply functions

```
partialFunction :: Int -> Bool
partialFunction = yourLowerCaseFunctionName "abc"

fullyApplied :: Bool
fullyApplied = yourLowerCaseFunctionName "test" 123
```

Infix functions and operators

`1 + 2` is the same as `(+) 1 2`

`mod 5 2` is the same as `5 `mod` 2`

Composing functions

`f (g a)` is the same as `f $ g a` and `(f . g) a`

Data Types

Useful predefined basic data types: `Int, Bool, Float, Double, Text, ()`

Useful predefined containers: `[a], Map a, Set a, Maybe a, Either e a, (a, b, ⋯)`

Define a custom product type

```
data YourUpperCaseTypeName = YourUpperCaseConstructorName
  { fieldName1 :: Int,
    fieldName2 :: [Text],
    fieldName3 :: Maybe Bool
  }
  deriving (Show, Eq)
```

Update a custom product type value

```
updateFirstTwoFields :: YourUpperCaseTypeName -> YourUpperCaseTypeName
updateFirstTwoFields record =
  record
    { fieldName1 = fieldName1 record + 1,
      fieldName2 = []
    }
```

Define a custom union type

```
data AnotherTypeName
  = FirstConstructor
  | SecondConstructor Int
  | ThirdConstructor (Maybe Text) Bool
  deriving (Show, Eq)
```

Pattern matching

```
isEmpty :: [a] -> Bool
isEmpty [] = False
isEmpty (first : rest) = True

someFunction :: AnotherTypeName -> Int
someFunction FirstConstructor = 0
someFunction (SecondConstructor n) = n
someFunction (ThirdConstructor _ True) = 1
someFunction (ThirdConstructor _ False) = -1

withCaseOf :: Maybe a -> [a]
withCaseOf mayA = case mayA of
    Just a -> [a]
    Nothing -> []
```

Typeclasses

```
class JSConversion a b where
    cast :: a -> b

instance JSConversion Bool Int where
    cast True = 1
    cast False = 0

instance JSConversion Int Bool where
    cast 0 = False
    cast _ = True
```

Helper methods

```
complicatedOperation :: Int -> Int
complicatedOperation n =
  let x = n * 2
      y = x - 5
  in x + y

complicatedOperation2 :: Int -> Int
complicatedOperation2 n =
  x + y
  where
    x = n * 2
    y = x - 5
```

Do notation and IO

```
readAndPrint :: IO ()
readAndPrint = do
    input <- getLine
    putStrLn (input <> " received!")

readTwoLines :: IO (Text, Text)
readTwoLines = do
    firstLine <- getLine
    secondLine <- getLine
    return (firstLine, secondLine)
```

Modules

```
module Path.To.File (functions, AndTypes, thatArePublic) where

import BasicPrelude
import IntoGlobalScope
import qualified OnlyAccessibleWithPrefix
import OnlySelectedExports (thisOne, andThisOne)
import qualified With.Alias.Prefix as TheAlias

...
```

Functions you will need

```
-- ($): Function application

appliesLengthLast = length $ [1,2,3] ++ [4,5,6]
-- this is the same as below
alternativeLengthLast = length ([1,2,3] ++ [4,5,6])

-- (.): Function composition (right to left)
multiplyThenAdd = (+ 5) . (* 3)

-- fmap: Mapping a function over a container
-- This equals [2,3,4]
everyElementIncrementedByOne = fmap (+ 1) [1,2,3]
-- This equals (Just 3)
theSameButForMaybe = fmap (+ 1) (Just 2)

-- filter: Filtering a list by a condition
-- This equals [2,4]
evenNumbers = filter ((== 0) . (`mod` 2)) [1,2,3,4,5]

-- find: Find the first element in a container that satisfies a condition
-- This equals (Just 2)
firstEvenNumber = find ((== 0) . (`mod` 2)) [1,2,3,4,5]

-- traverse: Swap the order of two container like things after a mapping operation
listOfIOs :: [IO Text]
listOfIOs = fmap (\ _ -> getLine) [1,2,3]

getFirstThreeLines :: IO [Text]
getFirstThreeLines = traverse (\ _ -> getLine) [1,2,3]

-- sequence: Same thing but without the mapping operation
firstThreeLinesAlternative :: IO [Text]
firstThreeLinesAlternative = sequence listOfIOs
```

```haskell
-- foldl': Flatten a container from the left into some other structure
-- sumFromOneToThree = foldl' (+) 0 [1,2,3]

-- bracket (from Control.Exception): Safely close IO resources like DB connections
doWithDatabaseConnection = bracket connect disconnect
  (\conn -> someThingUsingTheConnection conn)
```