Epitech Documentation

Haskell Coding Style

Keep your functional code nice and clean

1.0

EPITECH.

The *Epitech Haskell Coding Style* is a set of rules, guidelines and programming conventions that has been created within the school, and that you have to respect.

It applies to all the source files present in your projet, with the notable exception of the **test/** and **bonus/** directories.

It is compulsory on all programs written in Haskell as part of Epitech projects, regardless of the year or unit.

It's easier and quicker to follow the guide style from the beginning of a project rather than to adapt an existing code at the end.

The goal of this *Coding Style* is to prevent the usage of common anti-patterns and to encourage students to write idiomatic haskell.

Its philosophy is to write code in a **declarative** style as much as possible, avoiding patterns remembering imperative constructs.

There are 3 levels of severity: **major**, **minor** and **info**.

There are many and many ways to produce unclean code.
Even though one cannot mention all of them in this document, they have to be respected.
We call them **implicit rules** when not explicitly defined in this document.

Implicit rules are considered as infos.

This document is inspired by Haskell's Wiki Programming Guidelines, which you should read too.

0- Files organization

01- Contents of the delivery folder

- Your delivery folder should contain only **files required for compilation**.

This means no compiled(`.o`, `.hi`, `.a`, `.so`, ...), temporary or unnecessary files (`*~ * #`, `*.d`, `toto`,...).

02- File extension

- Sources in a Haskell program should only have extension `.hs`.

03- File coherence

- A Haskell project must be organised in **modules**, each of which should match a **logical entity**, and group all the functions and data structures associated with that entity.

There is no limit to the number of functions in a single file

04- Naming files and folders

- The name of a file should match the name of its module. Therefore, files and modules must be named in **UpperCamelCase** and in English.

It is tolerated for the file containing the main to be named after the project's executable, in lower-case.

E- Extensions

E1- Language Extensions

- All languages extensions are forbidden.

T- Types

T1- Top level bindings signatures

- All top level bindings must have an accompanying type signature.

```
-- T1 violation
main = putStrLn "hello world"

-- No T1 violation
main :: IO ()
main = putStrLn "hello world"
```

M- Mutability

M1- Mutable variables

- Mutable variables are strictly forbidden.

This forbids the use of all types allowing mutation, such as **Data.IORef**, **Data.STRef** or **Control.Concurrent.STM.TVar** but not **Control.Monad.Trans.State**

F- Functions

F1- Coherence of functions

A function should only do **one thing**, not mix the different levels of abstraction and respect the principle of single responsibility (a function must only be changed for one reason).

A function should be less than 10 lines long and 80 columns wide.

F2- Naming functions

The name of a function should **define the task it executes** and should **contain a verb**.

For example, the `voyalsNb` and `dijkstra` functions are incorrectly named. `getVoyalsNumber` and `searchShortestPath` are more meaningful and precise.

All function names should be in English, according to the `lowerCamelCase` convention. Special characters are tolerated as long as they are justified and used sparingly.

Abbreviations are tolerated if they significantly reduce the name without losing meaning.

V- Variables

V1- Naming identifiers

All identifier names should be **in English**, according to the `lowerCamelCase` **convention**

Abbreviations are tolerated as long as they significantly reduce the name length without losing meaning, or if they are idiosyncratic to Haskell (for example: **x:xs**)

The type names and constructors should be **in English**, according to the `UpperCamelCase` **convention**.

C- Control structure

Unless otherwise specified, all control structures are **allowed**.

C1- Conditional branching

Nested **If** statements are stricly forbidden.

```
-- C1 violation
foo :: Int -> Int -> Int
foo a b = if a == 0
          then b
          else if b > 0
               then a-b
               else b-a

-- No C1 violation
foo :: Int -> Int -> Int
foo 0 b = b
foo a b | b > 0 = a-b
        | otherwise = b-a
```

C2- Guards and pattern matching

Guards which can be expressed as pattern matchings must be expressed as such.

```
-- C2 violation
foo :: [Int] -> Int
foo lst | lst == [] = 0
        | head lst == 1 = 1
```

```
-- No C2 violation
foo :: [Int] -> Int
foo [] = 0
foo (1:_) = 1
```

D- Do notation

Unjustified use of the do notation is forbidden.

D1- Useless do

The **Do** notation is forbidden unless it contains a generator (a statement with a left arrow).

```
-- D1 violation
foo :: Int -> Int -> Int
foo x y = do
  let x2 = x * x
  let y2 = y * y
  sqrt $ x + y

-- No D1 violation
foo :: Int -> Int -> Int
foo x y = let x2 = x * x
              y2 = y * y
          in sqrt (x + y)

-- No D1 violation
printLineLength :: IO ()
printLineLength = do
  line <- getLine
  print $ length line
```

For simple chaining of actions, use the >> operator:

```
-- D1 violation
printUsage :: IO ()
printUsage = do
    putStrLn "usage: foo [options] [files]"
    putStrLn "       -s : simple"
    putStrLn "       -h : hard"

-- No D1 violation
printUsage :: IO ()
printUsage = putStrLn "usage: foo [options] [files]" >>
             putStrLn "       -s : simple" >>
             putStrLn "       -h : hard"
```

To prevent any circumvention of rule D1, useless generators are forbidden (for example: x <- return 42 is an implicit D1 violation)