



Epitech Documentation

Haskell Coding Style

Keep your functional code nice and clean



1.1



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 project, with the notable exception of the **test/**, **tests/** 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.

You will find alongside this document a Linux binary called **lambdananas** which is the tool we use to check your code. Use it to check if your code respects the coding style.



lambdananas takes its input from stdin if run without argument, try **-help** to see how to use it

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.



O- FILES ORGANIZATION

O1- 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,...).

O2- FILE EXTENSION

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

O3- 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. Every haskell file (including Main) should declare a module.



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

O4- 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.

O5- MODULE EXPORTS

- ⚠ All modules should explicitly declare their exported definitions. Except the Main module, all modules are expected to export at least one definition.



G1- EPITECH HEADER

- 💡 Every haskell file should start with a standard Epitech header:

```
{-  
-- EPITECH PROJECT , 2022  
-- $PROJECT_NAME  
-- File description:  
-- $DESCRIPTION  
-}
```

E- EXTENSIONS

E1- LANGUAGE EXTENSIONS

- ⚠ All language extensions are forbidden except if the project's subject says otherwise.

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 AND SAFETY

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`

M2- UNSAFE FUNCTIONS

⚠ Functions performing unsafe operations are strictly forbidden.



This forbids the use of all functions labeled as unsafe, such as `unsafePerformIO`

M3- FORBIDDEN MODULE IMPORTS

⚠ In order to enforce rules M1 and M2, importing any of the following module is strictly forbidden:

- `Data.IORef`,
- `Data.STRef`
- `Control.Concurrent.STM.TVar`
- `System.IO.Unsafe`



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).

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.

F3- LINE LENGTH

- 💡 A Line must be less than 80 characters long.

F4- FUNCTION LENGTH

- 💡 A Function body must be 10 lines or less.

When using pattern matching a function can have several bodies, they are counted independently. `let ... in` and `where` blocks are counted in function bodies though.

```
-- This function has two bodies of one line each.
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False

-- This function has one body of 3 lines.
isZero :: Int -> Bool
isZero x = case x of
  0 -> True
  _ -> False
```



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.

C4- CONDITIONAL BRANCHING

⚠ Nested If statements are strictly 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
```

C5- GUARDS AND IFS EXPRESSED AS PATTERN MATCHING

⚠ Guards and if statements 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

-- C3 violation
bar :: Bool -> Int
bar a = if a == True
         then 42
         else 84

-- No C3 violation
bar :: Bool -> Int
bar True = 42
bar False = 84
```

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"
```

D2- USELESS GENERATOR

⚠ To prevent any circumvention of rule D1, useless generators are forbidden. Useless generators are generators which should be simple **let** statements.

```
-- D2 violation
x <- return 42

-- No D2 violation
let x = 42
```