Keera Studios™ Haskell Style Guide V1.3.0

Keera Studios Ltd

Keera Studios™ Haskell Style Guide V1.3.0

1 Introduction

This document serves as the **complete** definition of Keera Studios™ coding standards for Haskell files. A Haskell file is described as being *in Keera™ Style* if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, rules in this document focus primarily on the **hard-and-fast rules** that we follow universally, and avoids mandating *advice* that isn't clearly enforceable (whether by human or tool).

This guide includes two kinds of items: rules and suggestions. Rules must be enforced to be policy compliant. Suggestions should be used to improve the coding style, but need not be enforced. The use of the words must vs should indicates whether the element is a rule or a suggestion.

1.1 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Keera™ Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

2 Source file basics

2.1 File name

Rule: The source file name consists of the case-sensitive name of the module it contains, plus the `.hs` extension.

2.2 Special characters

2.2.1 Whitespace characters

Rule: Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation and not allowed anywhere in the file.

3 File structure

Rule: A Haskell file consists of, **in order**:

1. Pragmas
2. Compilation options
3. Module haddock documentation, including any copyright notices applicable
4. Module declaration
5. Imports
6. Definitions

Rule: **No blank lines** separate pragmas, the module haddock declaration, and the module declaration. **Exactly one blank line** separates the module declaration from the imports section, and the imports section from the definitions section.

3.1 Language pragmas

Rule: Language pragmas must be alphabetized.

Rule: Each pragma must be listed on a separate line.

Rule: The opening and closing comment indicators of pragmas must be aligned.

3.2 Compilation options

Rule: The opening and closing comment indicators of comment options must be aligned.

3.3 Module documentation

Rule: All Haskell files must include a copyright notice in a comment. The copyright notice must be included in a format parseable by Haddock:

```
-- |
-- Copyright   : (c) HOLDER, YEAR
-- License     : RESERVED
-- Maintainer  : CONTACT ADDRESS
```

For example, a package developed in the year 2014 by Keera Studios™ would have a notice:

```
-- |
-- Copyright   : (C) Keera Studios Ltd, 2014
-- License     : All Rights Reserved
-- Maintainer  : support@keera.co.uk
```

Additional notices at the top may include the full Copyright license, a pointer to the license contained in a file, or other confidentiality notices and warnings.

Suggestion: include haddock documentation describing the purpose of the module.

3.4 Module declaration

Suggestion: the module declaration should explicitly list the definitions exported by the module.

Suggestion: the module declaration should use haddock indicators to group the elements exported so as to facilitate navigating the documentation.

Suggestion: any such documentation groupings should be consistent across modules and across packages.

3.5 Imports

Suggestion: imports should be grouped between external imports, external imports from the same project (but different libraries) and internal imports. When groups are used, comments must be used to indicate the nature of each group. If there are no distinct groups, they are considered to be all in the same group.

Rule: imports belonging to the same group should be listed in alphabetical order.

Rule: elements imported from a module should be listed in alphabetical order.

Suggestion: imports should be explicit (aka. no wildcard imports).

Suggestion: imports should not be redundant (aka. no unused imports).

Suggestion: imports belonging to the same group should not be separated by white lines.

Suggestion: the same module should not be imported more than once in the same form (qualified or unqualifed).

Suggestion: imports groups must be separated by one white line.

See also rules regarding horizontal alignment (4.6.6 Horizontal alignment).

3.6 Definitions

Other definitions must follow the imports, like types, functions, classes, etc.

3.6.1 Type signatures

Rule: All top-level functions must include a type signature.

3.6.2 Order and structure

Suggestion: Where there is an explicit export list, the file should follow the same structure and order as the export list.

Suggestion: Where no export list is included and the module defines an abstract data type, the order should be: type, constructors, accessors, modifiers.

Suggestion: In other cases (main programs, modules without explicit export lists, etc.), a top-down listing should be used, where elements that use other elements appear before them.

Suggestion: Auxiliary definitions not exported by the module should follow the definition that uses them, or be listed in an auxiliary section last in the file.

3.6.3 Documentation

Suggestion: All top-level definitions should include haddock documentation.

Suggestion: Each argument function or record field should be documented separately.

3.6.4 Redundancy

Suggestion: No redundant (unused) definitions should be included. A definition may be redundant because it is internal to a module or function and not used in the module, or because it is not used by any project that uses a library or module.

4 Formatting

4.1 Naming conventions

Suggestion: underscore should not be used in names, except to indicate lack of effects (as the last character in the name) or in functions that export elements from other libraries via the FFI.

Suggestion: prime should not be used in names exported by the module.

4.2 Indentation: +2 spaces

Suggestion: Each time a new expression or block-like construct is opened, the indent should increase by two spaces. When the section ends, the indent returns to the previous indent level. The indent level applies to both Haskell expressions and comments throughout the block.

Exceptions: Indentation can be decreased when a new line break is forced. For example, the following would be allowed:

```
function = do
  action1
  action2
  return value
```

Exceptions: Indentation can be increased by more than 2 spaces when a keyword opens a new block-like construct. For example, the following would be allowed:

```
function = if x
             then longFunctionWithArguments
                    firstArgumentWhichDoesNotFitOnFirstLine
             else longFunctionWithArguments
                    firstArgumentWhichDoesNotFitOnFirstLine
```

Suggestion: For lists or tuple elements separated by commas in which each value is listed in a separate line, and when the Haskell syntax allows it, the comma must be the first character of every line but the first. For example:

```
function = f ( longExpression1
             , longExpression2
             , longExpression3
             )
```

4.3 Column limit: 80

Suggestion: We set the code limit for Haskell files at 80 characters. A "character" means any Unicode code point. Except as noted below, any line that would exceed this limit should be line-wrapped, as explained in Section 4.5, Line-wrapping.

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in a comment, or a long module name, or a long function or constant name).
2. Command lines in a comment that may be cut-and-pasted into a shell.

4.4 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called *line-wrapping*.

Suggestion: line wrapping should be used to help clarify the code, and not avoided to produce code that fits in the smallest number of lines.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code. While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit *may* be line-wrapped at the author's discretion.

4.5 Whitespace

4.5.1 Vertical Whitespace: always recommended

Suggestion: A single blank line should appear:

1. *Between* consecutive definitions.
2. *Between* consecutive entries in where and let clauses that have type signatures, or that are conceptually disjoint.
3. As required by other sections of this document (such as Section 3, Source file structure).

4.5.2 Vertical Whitespace: never allowed

Suggestion: Do not leave more than two consecutive empty lines anywhere in the file.

Rule: Do not leave any empty lines at the end of the file.

4.5.3 Horizontal whitespace: always recommended

Suggestion: Beyond where required by the language or other style rules, and apart from literals and comments, a single ASCII space should also appear in the following places:

1. After the comma (,) in lists and tuples. For example, the following is discouraged:

   ```
   times (x,y) = x * y
   ```

   Instead, you should add one or more spaces (as determined by horizontal alignment rules below):

   ```
   times (x, y) = x * y
   ```

2. On both sides of any binary or ternary operator.

3. After a function being applied and before its arguments.

This suggestion is never interpreted as requiring or forbidding additional space at the start or end of a line; it addresses only *interior* space.

4.5.4 Horizontal whitespace: never recommended or allowed

4.5.4.1 No space before comma

Suggestion: Do not leave a space before the comma (`,`) that follows an expression. For example, the following is not recommended:

```
times (x , y) = x * y
```

Instead, the comma should follow the variable:

```
times (x, y) = x * y
```

4.5.4.2 No trailing spaces

Rule: Do not leave any spaces at the end of any line.

4.5.4.3 No unnecessary spaces

Suggestion: Do not leave any extra spaces except as mandated or recommended by these rules. For example, avoid having two or more spaces before an argument except when it is done to align it horizontally with other arguments in other lines.

4.5.5 Horizontal alignment

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

Rule: Horizontal alignment is required in the following:

- For the module names and lists of imported symbols in contiguous import declarations (that is, those beloging to the same group or those without empty lines separating them). For example:

```
-- External imports
import qualified Long.Module.Name as X
import           Short.Module     (Type1, function1)

-- Internal imports
import Other.Module (Type2, function2)
```

Suggestion: Horizontal alignment is strongly encouraged in the following:

- For commas in comma-separated fields listed in multiple lines. For example:

```
function = f ( longExpression1
             , longExpression2
             , longExpression3
             )
```

- For the arguments and the equals sign of small function left-hand sides with the same structure. For example:

```
f [] _ = Nothing
f (x:xs) ys = g x ys : xs
```

should be written as:

```
f []     _  = Nothing
f (x:xs) ys = g x ys : xs
```

- For types and equals sign of several small, contiguous, related definitions inside `where` or `let` clauses not separated by empty lines. For example:

  ```
  let x = 55 + yVal
      yVal = longerExpression
  ```

  should be written as:

  ```
  let x    = 55 + yVal
      yVal = longerExpression
  ```

- For arrow head and arrow ends in arrow notation blocks for several small, contiguous, related definitions not separated by empty lines. For example:

  ```
  x <- arr (+) -< (55, yVal)
  y <- longerExpression -< x
  ```

  should be written as:

  ```
  x <- arr (+)          -< (55, yVal)
  y <- longerExpression -< x
  ```

Tip: Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

4.6 Parenthesis

Suggestion: Do not wrap expressions in parenthesis unnecessarily.

4.7 Comments

Rule: Comments must always be line comments, except for pragmas and GHC options, which can be listed as block comments.

5. Haskell constructs

5.1 Prefer `where` to `let`

Suggestion: When allowed, prefer `where` to `let` clauses (except as needed in do blocks and arrow blocks).

5.2 `where` always appears on a single line

Suggestion: prefer `where` appearing alone on a single line, not at the end of a line or on the same line as the definition it introduces. For example:

```
f (x:xs) ys = g x ys : xs where
  g = something
```

and

```
f (x:xs) ys = g x ys : xs
  where g = something
```

should be written as:

```
f (x:xs) ys = g x ys : xs
  where
    g = something
```

5.3 Name elements consistently

Suggestion: Name elements consistenly. For example, an API should not refer to the word configuration as `Conf` in some function names and `Config` in others. Similarly, a variable that defines a runtime context should not be called `rCtx` in one scope and `rtctxt` in another.

5.4 Prefer longer, more descriptive names

Suggestion: Prefer longer, more descriptive names to short, abstract ones. For example, a function that takes a length as argument should not call the variable `l` and, instead, call it `len` or a longer name.

5.5 Prefer consistency over local optimizations

Suggestion: Prefer consistent names, structure, and improvements over localized optimizations. For example, a module might define an API in a particular order, and a module with a similar but distinct API might list elements in a completely different order. Wheverver possible, code should be consistent.

5.6 Prefer typing expressions

Suggestion: Type expressions in where and let clauses, even when the type checker can infer the types.

5.7 Avoid warnings

Suggestion: Avoid any compilation warnings. Where the warning is acceptable, document it with a comment and, optionally, with an indicator that makes the compiler not trigger the warning.

5.8 Avoid typos

Suggestion: Avoid typos in definitions, module names, documentation, and comments.

5.9 Avoid commented code

Suggestion: Avoid including commented code, either old or alternative implementations, or ideas.

5.10 Avoid TODO/FIXME notes

Suggestion: Avoid including TODOs or FIXMEs in the code. Instead, document the issue as part of the element's interface, or create an issue for it in the issue tracker.

Appendix A Notices