Programming 1 Haskell Style Guide

This document sets out the style guide for Haskell programming in the coursework, on which the **quality** marking is based. There are two aspects, **presentation** and **technique**.

Presentation

The aim of good presentation is to make your code as readable as possible without affecting the program logic. We distinguish three categories: layout, naming, and commenting.

- **Layout:** a good use of horizontal and vertical space is extremely important in making your code readable. The following aspects are particularly important:
  - horizontal indentation to create code blocks;
  - horizontal alignment to highlight patterns and similarities, and to make pattern-matching and guards easily readable;
  - vertical spacing between functions and code blocks in larger functions to visualise the structure of the code;
  - ordering of code within larger functions to create blocks with a similar functionality, either within the block or between blocks;
  - ordering of functions to create groups with similar functionality and to give a logical structure to the code;
  - breaking up very long lines of code to form a code block (for instance by putting the different clauses of a list comprehension on separate lines, aligned horizontally).

  Good layout reflects the logical structure of the program, and makes it attractive to read. Things to avoid in particular are very long lines and large or dense wall-of-text blocks of code.

- **Naming:** good variable and function naming strikes a balance between being concise and being descriptive. Shorter names are more readable and make it easier to create a good layout, while longer names make it easier to recall the meaning of a variable or function. The following conventions are good practice.
  - Use a fixed naming convention such as camelCase or snake_case.
  - Defined functions should be given descriptive names, never single letters. Abbreviations should be avoided.
  - For function inputs, even if they are functions, single-letter names are generally preferred, with some exceptions: when there are many of them, or in larger functions with a large distance between where they are taken as input and where they are used.

  - Be consistent with variable names, in particular for single-letter variables, using a fixed alphabet (e.g. `f`, `g`, `h` for functions, `i`, `j`, `k` for indices, `n`, `m` for numbers, and `str` for strings) and using a prime (an apostrophe) for very similar or derived values (e.g. `xs` and `xs'`).
  - Adhere to general naming practices and preferences, such as using nouns for values and verbs for functions, plural (e.g. `xs`) for lists, `show` - (as prefix) for returning strings to be displayed, `print` - for printing to console, `is` - for boolean checks, etc.

- **Commenting:** the purpose of commenting is to provide formal documentation for your code. It should give a precise, informative description of the functionality you have implemented aimed at professional maintainers (not at the lecturer marking your work).

  The gold standard is to write **self-documenting** code, i.e. code that is perfectly transparent without the need for commenting. This is not always attainable, and that is where commenting comes in.

  The purpose and functioning of any given function should be clear from its name, type signature, and documentation (which should be placed above it, not besides or below the signature). The reader should not have to inspect your code to understand the purpose of your function. This way, it is possible to check if the function is correct, since the intention expressed in the signature and comments can be compared against the actual functioning of the code.

  Mid-function comments should be used extremely sparsely. Their use is for instance: to introduce code blocks in longer `IO` functions, to recall conditions in conditional branches (particularly `else`-branches), or to recall implicit constraints in data being processed (for instance that an input list is assumed to be sorted).

  Generally, comments should describe **what** your code intends to do, not **how** it does it; i.e. comments are about the **meaning**, not the **implementation** of your function. The exception is when your code implements a particular familiar algorithm, such as a **breadth-first tree traversal**, in which case you should mention this.

Technique

The aim of good programming technique is to promote efficient and transparent program logic. This involves breaking a problem up into sensible, manageable parts, and approaching each with appropriate means. We distinguish four categories: constructions, functions, algorithms, and abstraction.

- **Constructions:** like any programming language, Haskell offers a range of standard (and less standard) constructions and techniques with which to build a program, such

as recursion, conditionals (guards and if–then–else), pattern-matching, list comprehension, higher-order functions, and where- and let-clauses. Good code requires choosing appropriate constructions for the task at hand, and using them well.

The choice of construction is important because there are trade-offs: for example, recursion is extremely powerful but verbose and not very transparent; list comprehension is highly transparent but must return a list; and higher-order list functions are concise but not very flexible. For conditionals, choosing the right boolean test can make a huge difference to the structure and readability of a program, as can organising code well in where- and let-clauses.

Some key examples: try to avoid redundant pattern-matching (when both branches are treated similarly); choose well between pattern-matching or guards; avoid the pattern `if x then True else False` (since it is equivalent to just `x`); use where-clauses to organise your code; and use partial application, sections, and function composition where this gives more readable code, but not elsewhere.

- **Functions:** a good program consists of clear functions with a logical purpose, so that each function can be readily understood and its functionality remembered. Functions with obscure parameters and/or behaviour make a program slow and frustrating to read: if the purpose of a function is not easily remembered, one has to constantly look it up. Large function bodies likewise are difficult to inspect. And as repeated often in this unit, the purpose of a function should be clear from its name and type signature (and, if necessary, comments).

  Things to avoid are functions that are too long or too complex, that have a very large number of parameters (especially of the same type), or that have an unclear purpose. Don't nest constructions too deeply (in particular conditionals), and avoid unnecessary mutual recursion (two or more recursive functions calling each other)

- **Algorithms:** a good program uses appropriate algorithms to compute the desired solution. This generally means using a simple, transparent algorithm of the right computational complexity. While readability is in many instances more important than pure speed, there is a complication: an obviously inefficient algorithm is unexpected, which means it takes more time and effort to understand the program.

- **Abstraction:** when the same functionality is needed in several places in the same program, good practice is to abstract this in a single function that is then called multiple times. Good abstraction reduces the overall amount of code needed, creates good structure in a program, and aids in readability and maintainability. Note that the abstracted function should serve a logical purpose, and not simply carry a large number of parameters to handle different cases.