
CO2008 Functional Programming

Credits: 10 **Convenor:** Dr.F.J. de Vries **Semester:** 1st

Prerequisites: *Essential: CO1003, CO1004*

Assessment: *Coursework: 40%*

Three hour exam in January: 60%

Lectures: 15 *hours*

Surgeries: 5 *hours*

Laboratories: 10 *hours*

Private Study: 45 *hours*

Subject Knowledge

Aims The module will give the student an introduction to programming in the functional style using the language Haskell.

Learning Outcomes Students will be able to demonstrate: skilled use of basic functions and techniques to solve simple problems, with some practical applications; detailed knowledge of numbers, lists, recursion, and patterns; some understanding of higher order functions; and ability to apply Haskell's mechanism for defining new datatypes.

Methods Class sessions together with lecture slides, recommended textbook, worksheets, printed solutions, and some additional hand-outs and web support.

Assessment Marked coursework, traditional written examination.

Skills

Aims To teach students how to methodically solve problems given the techniques available to them.

Learning Outcomes Students will be able to: breakdown simple problems to identify essential elements; create a plan to solve a problem; implement a planned solution and evaluate the implementation.

Methods Class sessions together with worksheets.

Assessment Marked coursework, traditional written examination.

Explanation of Prerequisites It is essential that students have taken a first course in basic (imperative) programming, which includes skills involving both coding and design, to a level which includes data structures such as lists and trees.

A grounding in the basic mathematical concepts of sets and functions is useful, but detailed knowledge of other areas of elementary discrete mathematics and logic is not essential.

Course Description Many of the ideas used in *imperative* programming arose through necessity in the early days of computing when machines were much slower and had far less memory than they do today. Languages such as C(++) and Pascal carry a substantial legacy from the past. Even Java, despite its OO features, has been devised to look 'a bit like C'. If one were to start again and design a programming language from scratch what would it look like?

For many applications, the chief concern should be to produce a language which is concise and elegant. It should be expressive enough for a programmer to work productively and efficiently but simple enough to minimize the chance of making serious errors. Rapid development requires the programmer to be able to write algorithms and data structures at a high level without worrying about the details of their machine-level implementation. These are some of the criteria which have led researchers to develop the *functional* programming language Haskell.

The flavour of programming in Haskell is very different from that in an imperative language. Much of the irrelevant detail has been swept away. For example, there are at least two different uses for a variable in Java: as a storage location, and as parameter in a method. There is only one use for a variable in Haskell: it stands for a

quantity that *you don't yet know*, as is standard in mathematical practice. The constructs in Java include expressions, commands, and methods; whereas in Haskell there are only expressions and functions. The meaning of a program in Java or C is understood by the effect it has on the 'state' of the machine as it runs. Haskell does away with the idea of 'state'—the meaning of a program is the values it computes.

On the other hand, Haskell is a very expressive language. The type system allows functions to be written *polymorphically* so that the same code can be re-used on data of different types, e.g. the same `length` function works equally well on lists of integers as on lists of reals or lists of strings. Furthermore, it allows one to write functions which take other functions as parameters. These are known as *higher order* functions and they give a second form of code re-use. There are powerful mechanisms for introducing user-defined datatypes such as trees, sets, graphic objects, etc. Haskell also makes a great deal of use of recursion. The combination of these features makes for very clean, short programs, which, with some experience, are easier to understand than many imperative programs.

This course teaches how to program in Haskell, which exemplifies the functional style.

Detailed Syllabus Basic types, such as `Int`, `Float`, `String`, `Bool`; examples of expressions of these types; overloading. Functions and declarations, with a high level explanation of a function with general type $a_1 \rightarrow a_2 \rightarrow a_3 \dots \rightarrow a_n$. Booleans and guards; correspondence of guards with if-then-else expressions. Pairs and n-tuples; `fst` and `snd` functions for dismantling pairs and tuples. Pattern matching and cases, especially defining functions on lists and tuples. Numeric calculation. Simple recursion, with examples on the natural numbers and lists; list comprehension; list processing examples which use patterns, recursion and comprehensions. Higher-order functions, polymorphism and code re-use; examples such as the reversal of a list. Algebraic and recursively defined datatypes. Examples such as lists and trees.

Reading List

- [A] G. Hutton, *Programming in Haskell* ISBN: 0-521-69269-5, Cambridge University Press. 2007.
- [B] R. Bird and P. Wadler, *Introduction to Functional Programming*; ISBN: 0134843460, Prentice Hall 1988.
- [B] S. Thompson, *Haskell: The Craft of Functional Programming, 2nd Edition*; ISBN: 0201342758, Addison-Wesley. 1999.
- [C] L. Paulson, *ML for the Working Programmer, 2nd Edition*; ISBN: 052156543X, CUP 1997.

Resources Lecture slides, web page, study guide, worksheets, handouts, lecture rooms with OHP, dataprojector and whiteboard; past examination papers.

Module Evaluation Course questionnaires, course review.