Preface

In mathematics, one speaks of $x^2 = 1$, an *equation*, as having solutions $x = 1$ and $x = -1$. But $x = 1$ and $x = -1$ are equations too, since they both have the form of two expressions separated by '='. If they are all three equations, then what gives the latter two the additional quality of being solutions?

There are two characteristics of solutions. The first is that they solve something, and in the example above we have the diagram

$$x^2 = 1 \quad \begin{cases} \Leftarrow & x = 1 \\ \Leftarrow & x = -1 \end{cases},$$

showing that we say that '$x = 1$' *solves* '$x^2 = 1$' because of the implication 'if $x = 1$ then $x^2 = 1$'. (Note that the reverse implication does not hold.) The same applies for '$x = -1$'.

The second characteristic of solutions is that the value they determine can be recovered by inspection, without further calculation. It hardly needs saying that '1' is a value for x that makes $x = 1$ true: it can be seen almost without thought.

One can also regard computer programming, or program development as it is known these days, in terms of 'solving' and 'solutions'. Instead of equations, however, we will have programs: some programs will be regarded as solving others (we will say 'refining'); and some programs will be so simple we will regard them as solutions (we will say 'code').

Using a special notation for statements like 'set x so that $x^2 = 1$', we can write specifications in a way that looks like programming. And by equipping them with a carefully defined meaning, we can say that they indeed *are* programs. The above specification, for example, we would write 'x: $[x^2 = 1]$' , and the *is refined by* relation '$\sqsubseteq$' between programs would then allow us an analogue of the above 'is solved by' diagram:

$$x\colon \left[x^2 = 1\right] \quad \begin{cases} \sqsubseteq & x := 1 \\ \sqsubseteq & x := -1 \end{cases}.$$

All three components of the above are programs. The first characteristic of solutions is repeated in the implication 'if we require an x such that $x^2 = 1$, then $x := 1$ will deliver one'. The second is repeated in that the value delivered by the programs on the right — they are both *code* — can be discovered by running them on a computer. And that requires no thought at all.

This book teaches elementary programming in the style of the above analogy: that specifications and code are all programs; that there is a refinement order between programs; and that there is a specially restricted sub-language called 'code' that allows programs written in it to be executed with 'no thought at all'. The thinking is necessary elsewhere, to *find* the code that refines a given specification.

The approach rests on the work of Dijkstra, Hoare, and Floyd [Dij76, Hoa69, Flo67]; and the programming language is Dijkstra's *guarded commands*, extended with specification constructs like 'x: $[x^2 = 1]$' above. The language itself is presented in the early chapters, and each of its constructs is characterised by the refinement laws that it satisfies. The effect on a computer is described informally, as an aid to the intuition.

Later chapters are split between case studies and more advanced programming techniques. Each case study treats a programming example from beginning to end, using the methods available at that point, and is well known rather than especially intricate. The more advanced programming techniques are *procedures*, *recursion*, *recursive data structures*, *modules* and finally *state transformation* (including *data refinement*).

The other chapters deal with the necessary infrastructure, most notably the predicate calculus and basic types. The former has its own laws, and a large collection of those appears as Appendix A. That avoids relying on any particular method of logical proof from first principles (properly the subject of a different book). Indeed, the introductory Chapter 1 uses the predicate calculus before its proper exposition in Chapter 2: the refinement calculus, not the predicate calculus, is our main subject. Nevertheless Chapter 2 is independent, and may be read first if desired. The basic types are the ordinary sets of numbers from arithmetic, augmented with constructions for powersets, bags, and sequences.

A concluding chapter summarises the main features of our programming style, and discusses its effect on the practices of *documentation*, *modification*, *testing*, and *debugging*.

Beyond the conclusion are several chapters more advanced than the main text. The first two treat fairly complex case studies; the first is iterative (an example of dynamic programming); the second is seriously recursive. The third advanced chapter is a case study in specification itself, thus concentrating on modules, design (of a system), design changes, and both ordinary and data refinement as a tool in careful design.

The final chapter gives the semantics for all the preceding material.

Appendices include a collection of propositional and predicate calculus laws, answers to some exercises, and a summary of the refinement laws introduced in the text. The last is sometimes convenient to have at hand when deriving programs, or studying others' derivations, and for that reason it may be copied and distributed for educational use. Those adopting the book for teaching may obtain from the publisher a separate booklet containing both the summary of refinement laws and answers to all the exercises.

The book is intended to be useful both to those learning to program and to those who — programmers already — would like to make the link between their existing skill and the specifications from which their programs (should) spring. Experience based on nearly 10 years' exposure to second-year computing undergraduates suggests, however, that the best approach is at first to exercise a fairly light touch on the refinement laws. For beginning programmers they should assist, not prescribe: at that stage, the list of refinement laws is for reference, not for certification. And learning to use invariants for iterations is work enough on its own.

Light touch or no, the underlying theme of specification, refinement and code is one that students respond to, and it informs their approach to other courses and to computing generally. More experienced programmers may recognise some of those features in specification and development methods such as *Z* and *VDM* [Jon86, Hay93, Kin90], for parts of which the refinement calculus forms an adjunct or even an alternative.