Preface

This book was written to support a short course in the second or third year of an undergraduate computer science, software engineering, or software design program. The prerequisites are fairly modest: some programming experience (ideally in C or C++ or a related language such as JAVA) and some exposure to the most basic concepts of discrete mathematics (sets, functions, binary relations, sequences) and to the language of elementary logic (connectives and quantifiers). It is intended to be only an *introduction* to software specifications, *not* a systematic survey of requirements engineering, formal methods, compilers, or computation theory suitable for a senior or graduate-level course. A course based on this book would provide a good foundation for such courses but should not replace them.

The contents may be summarized briefly as follows:

- specification, verification, and development of simple algorithms using pre- and post-conditions and loop invariants;
- specification, verification, and development of simple data representations using abstract models and representation invariants; and
- specification and systematic development of recognizers for formal languages using regular expressions, grammars, and automata.

These techniques have been well studied and are sound and useful. They may be presented to and immediately used by undergraduate students on simple but nontrivial examples. They may be taught without requiring upper-level prerequisites or major investments of time to teach complex notations or computer-based tools. But such material is not often presented at this level, nor in this combination. To explain why I have written this book, I will briefly describe its origins. Perhaps readers will recognize some similarities with the situations at their institutions.

At Queens's University, the undergraduate program in computer science has for many years included the following final-year courses:

- a "theory" course: formal languages, automata, and elementary complexity and computability theory;
- a "compilers" course: aspects of formal languages and automata relevant to development of scanners and parsers, as well as other topics on compilers such as symbol tables, code generation, and optimization;
- a "formal methods" course: various notations and tools for software specification and validation.

But a few years ago a controversy arose about whether such courses should be *required* of every graduating student. Some argued that every graduate of our degree programs should know *basic* material on computability and complexity, syntax analysis, and specification methods. But the instructors of the courses complained that there was not enough time available to treat all the material they thought should be covered and that many of the students were ill prepared for material involving mathematical formalism. On the other hand, many students were of the opinion that much of the material in these courses, which they called "abstract theory," had no practical relevance.

These issues were addressed by creating a new course. It was to be taken in the second or third year of our program and was to cover "basic" material formerly in the three final-year courses. The new course is now a prerequisite to these three courses and also to a variety of other courses in our program, including software engineering and foundations of programming languages. The final-year courses are now selected as *options* by students who are interested in those particular subjects (subject to some "breadth" constraints).

This approach to curriculum design has had several benefits. The basic material previously covered in the final-year courses is now required for almost all graduating students, without forcing every student to study advanced specialized material in areas of little interest to them. The duplication of material on formal languages and automata in the theory and compiler courses has been avoided by moving basic material into the new course. The final-year courses now have time to do advanced material, and the students in those courses are better prepared and more motivated. Perhaps the most important benefit has been that many students discover early in their programs that "theory" is actually *useful* because they now have an opportunity to apply mathematical rigor to programming problems at their level of expertise.

The main difficulty in presenting a "nonstandard" course is in finding a suitable text. Material on program and data specifications may be supported by a number of specialized texts [Rey81, Bac86, Gor88, Dro89, LG00], and a few books give an applications-oriented introductory presentation of formal-language material [Gou88, AU92], in addition to many specialized books on compilers; however, many of these books are now out of print, and a *unified*

treatment of the two bodies of material is clearly preferable. Also, students are unhappy if less than half of the material is covered from each of *two* expensive texts.

The present book, based on my lecture notes for the new course, addresses these problems. The main pedagogical novelties would seem to be the following:

- the hands-on and pragmatic approach to what is usually taught as theory, or as abstract discussion of "large complex systems";
- the way the material on formal languages has been integrated into a specification-oriented framework by treating state diagrams, regular expressions, and context-free grammars as specialized specification languages: formalisms for specifying language recognizers.

Students seem to find this approach far more relevant and convincing than traditional approaches to formal methods and formal languages. It must be emphasized, however, that the material presented here is intended to be only a *prelude* to, and not a *replacement* for, conventional compiler, theory-of-computation, and software-engineering courses.

I have *not* provided an introductory survey of discrete mathematics and logic on the assumption that students studying this material have recently taken or are concurrently taking a course in basic concepts of discrete mathematics and logic and have available a suitable textbook that they will be able to use as a reference. Some of the notation used here is superficially nonstandard, such as the "C-like" bounded quantifiers described in Section 1.3.2, but the concepts should be familiar.

The choice of a programming language to use for the examples was difficult. JAVA has become the most popular introductory language in computer science programs, despite some rather serious deficiencies in this role [BT97, AB+98, Gre]. But JAVA seems even less well suited to *this* material: it uses reference assignment and reference equality for `=` and `==` on objects, it lacks `enum` and (assignable) `struct` types, its scope rules and exception handling are complex and intrusive, it lacks a standard library for straightforward textual input, and simple algorithmic code doesn't fit easily into its object-oriented framework. PASCAL and similar languages such as MODULA, ADA, and TURING might be the most appropriate for the material, but students these days find the syntax strange and do not perceive them as being "practical" languages.

I decided to use a small fragment of C, with some simple use of C++ classes when information hiding is needed. Students who have programmed in JAVA or in another imperative language have very little difficulty reading and adapting simple C programs when unfamiliar idioms such as pointer arithmetic are avoided, as they have been here.

I am grateful to Michael Norrish (University of Cambridge), David Gries (University of Georgia), several anonymous reviewers, my Ph.D. student Dan Ghica, and my colleagues Jürgen Dingel and David Skillicorn for comments on draft versions of the material and to Tim Marchen and Tran Pham for programming assistance. Any remaining errors are my responsibility.

I would be pleased to receive comments and corrections; these may be sent to me at `rdt@cs.queensu.ca`. Errata will be posted here:

`http://www.cs.queensu.ca/home/specsoft`

R. D. T.
January 4, 2002

References

[AB+98] P. Andreae, R. Biddle, G. Dobbie, A. Gale, L. Miller, and E. Tempero. Surprises in teaching CS1 with Java. Technical Report CS-TR-98/9, Department of Computer Science, Victoria University, Wellington, New Zealand, 1998.

[AU92] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. W. H. Freeman, 1992.

[Bac86] R. C. Backhouse. *Program Construction and Verification*. Prentice Hall International, 1986.

[BT97] R. Biddle and E. Tempero. Learning Java: Promises and pitfalls. Technical Report CS-TR-97/2, Department of Computer Science, Victoria University, Wellington, New Zealand, 1997.

[Dro89] G. Dromey. *Program Derivation: The Development of Programs from Specifications*. Addison-Wesley, 1989.

[Gor88] M. J. C. Gordon. *Programming Language Theory and Its Implementation*. Prentice Hall International, 1988.

[Gou88] K. J. Gough. *Syntax Analysis and Software Tools*. Addison-Wesley, 1988.

[Gre] R. Green. Java gotchas. Available here: `http://www.mindprod.com/gotchas.html`.

[LG00] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[Rey81] J. C. Reynolds. *The Craft of Programming*. Prentice Hall International, 1981.