1

A WORD ABOUT USING THIS TEXT

This text, I believe, is unique in the computer science education market. Its format has been carefully thought out over approximately five years of research into problem-based and case-based learning methodology. This is a student-centered text: there is no attempt to make this a typical university textbook. There are many very good textbooks on the market that focus on encyclopedic inclusion of issues in the development of programming languages; this is *not* one of them.

Student-centric can mean different things to different people. This text is meant to be read by students, not faculty. This distinction means that the text presents problems to be solved and then presents the information needed by the student to solve the problem. The tone is informal and not academic. The ultimate purpose of the text is to show students how to be successful computer scientists professionally. At Clemson, this is considered our capstone course.

1.1 EXPECTATIONS FOR THE STUDENT AND INSTRUCTOR

To start the text I will give my assumptions and goals for the project.

1. Students are competent in one program from the C class of languages (C, C++, or Java). Their competence is assumed minimal at the professional level because they have not spent enough time to gain expertise.
2. Students are familiar with Unix.
3. Students are comfortable using search engines. A running joke in my classes is "Google is your friend."

If students will faithfully follow the project in this text, then experience teaching this course at Clemson shows the following:

1. The students will develop a full-scale project using professional-grade development management techniques. Anything less than full participation and the students will not gain all this text has to offer.
2. The students will master concepts, languages, and experiences to prepare them to understand the current research in programming languages.

In keeping with the problem-based nature of the text, here is the one problem you want to solve:

> What theory and knowledge do you need to develop a compiler for a higher-level language? What personal and professional methods do you need to successfully develop a large project? You're successful if you can explain all of the steps and algorithms to compile the following program and successfully run it:

```
[define :[g :[x int] int]
        [if [= x 0] 1 [* x [g [- x 1]]]]]
```

1.2 OPENING COMMENTS

This text is student-centric because it does not try to be encyclopedic, but rather guides students through the development of a small compiler as a way for them to understand the issues and commonly used approaches to the solutions of those issues. For this reason, the text is written in an informal, chatty style. I am attempting to set up a professional, design-team atmosphere and to promote strategies that will work in professional practice.

One issue that I have noticed uniformly across years, disciplines, and students is a complaint that textbooks are boring and therefore students do not read as they should. On closer examination, what one finds is that students rarely master fundamental study skills. In my classes I encourage by the way I make assignments what I called the FSQRRR method: Focus issue, Survey, Question, Read, Recite, Review. SQRRR is a familiar technique in K–12 education but not at the university level. The student first needs to understand the issues. Students often do not develop a questioning attitude and, therefore, when they do read, there is no focus or purpose.

The planning of the reading is a metacognitive exercise. There are many good online sites that describe SQRRR*; I encourage students and teachers alike to survey this technique and apply it to course work.

Without going into too much detail, a word about problem-based learning. There is a wealth of research that students learn best when they have a realistic problem before them and that problem *must* be solved. Please read the National Research Council's *How People Learn: Brain, Mind, Experience, and School* (2000). The research indicates that the problem should be ill-structured. Certainly, asking students to write a compiler for a language they have never seen qualifies as "ill-structured" and I claim it is a "realistic" expectation that a senior/post-graduate be able to design and implement such a compiler *if* the proper guidance is given. Therefore, this text seeks to provide guidance.

I have made a distinction in the planning of this text not usually drawn in the education literature. Problem-based learning is generally taken as "small" problems, hardly the adjective for a compiler project. The ideas of problem-based learning come under a more general term, *inquiry-based learning*. There are several other forms of inquiry-based learning but the rubric of *case-based learning* has a connotation separate from problem-based learning. Case-based learning is used in business and the law and stresses the judgmental issues of problem solving. Computer science is filled with judgments throughout every design. Every problem has many ways to be solved, each with its own computational costs in time and space. Computational costs are often not known in detail but require judgments by the designer. I have planted many case-based issues throughout the project: treat these as a minimal set; any judgment call can be discussed in a case-based rubric.

It is customary in case-based situations to provide commentary on the case. Such commentaries, if the students were to get them, would subvert the entire learning process. For that reason, the *Instructors' Manual* is a separate book, keyed to the cases in the student edition.

The approach of this text stimulates broad participation in the computer science enterprise because it emphasizes self-education and self-evaluation, as well as the design and implementation of complex systems. Just as metacognition is an important lifelong learning technique, so is explanation. I encourage instructors and students alike to spend time explaining the details of the problem at hand to one another. I like to emphasize an observation credited to Albert Einstein: "You don't understand something unless you can explain it to your grandmother." This observation emphasizes the idea that difficult technical ideas must be

* Google for it.

explainable in simple, ordinary English (or whatever natural language you're using).

The last educational issue that I want to discuss is the question, "How do I know the learner has learned?" My view is that in computer science, you only know something if you can present a working algorithm that produces only correct answers. I encourage the users of this text to spend time doing metacognitive exercises. Metacognition plays a crucial role in successful learning. Metacognition refers to higher-order thinking, which involves active control over the cognitive processes engaged in learning. Activities such as planning how to approach a given learning task, monitoring comprehension, and evaluating progress toward the completion of a task are metacognitive in nature. As Plato said, "The unexamined life is not worth living."

Along with the FSQRRR issue there are several issues concerning design of large software systems. There are at least two different aspects that should be addressed:

1. In light of the comment about metacognition, students should be urged to take stock of their practices. This is not a text on the personal software process (PSP) outlined by Humphrey (1996), but I have included a chapter on PSP, augmented with some design forms that I use in my class. My students—and, I shamefacedly admit, I—do not make good use of our time, and mostly because we do not take stock of how our time is spent. Do yourselves a favor: get a firm grip on your time—it's your only capital.
2. Current computer science curricula tend to focus on single procedures and not software systems. Therefore, students' design skills are fairly rudimentary. The project inherent in this text is far too complicated to give without much guidance. There is a fine line between "doing all the intellectual work for them" and "allowing them to figure things out for themselves." The goal is that the students learn how to do decomposition, analysis, and synthesis but my experience is that we, the instructors, need to give them guidance. I often quote George Pólya (1957): "If you cannot solve this problem, there is a simpler problem in here you can solve. Find and solve it." But I warn you, the students don't find this helpful the first time(s) they try to use it.

1.3 POSSIBLE SEMESTER COURSE

This section provides guidance in using this text and one possible sequence of material (see Figures 1.1 and 1.2).

	Class Periods		
Week	*Period 1*	*Period 2*	*Period 3*
1	Opening. Get data sheet. Explain how and why this course is different. Assign the specification as reading.	Organize groups. Organize Q&A on specs. Assign in-class exercise to code Hello World. Start *one-minute write* habit. Assign Chapter 5. Set Milstone I due date.	Address the focus question. Use the discussion to develop study plan. Assign Chapter 6 to focus question "How are programming languages like natural language?"
2	Explore the focus question. Assign Chapter 7.	This and the next period should pull together a complete block diagram of a compiler. Assign Chapter 8.	Ibid.
3	Work through the calculator. Assign Chapter 9.	Ibid.	Wrap up. Discuss overview of *Gforth*. Case 2 Introduction for Milestone I
4	Case 3.	Case 4 and 6.	Milestone I due.

Figure 1.1 Sample Organization for Introduction

1.3.1 General Planning

Good educational practice outlined in *How People Learn* (Bransford, Brown, and Cocking 2000) dictates that the course be built around three learning principles: (1) prior knowledge—learners construct new knowledge based on what they already know (or do not know); (2) deep foundational knowledge—learners need a deep knowledge base and conceptual frameworks; (3) metacognition—learners must identify learning goals and monitor their progress toward them.

1.3.1.1 Learning Principle 1: Prior Knowledge

The course I teach primarily requires deep knowledge of data structures. This deep knowledge includes not just the usual fare of lists, trees, and

	Class Periods		
Week	*Period 1*	*Period 2*	*Period 3*
5			
6			
7			Milestone II and III due
8			
9			Milestone IV due
10			
11			Milestone V due (passing grade)
12			
13			Milestone VI due
14			
15			Milestone VII: entire project due
16		Metacognitive final	

Figure 1.2 Sample Milestone Schedule

graphs, but also the ability to

1. Formulate detailed time and space requirements
2. Implement fundamental-type systems such as strings
3. Formulate alternative designs and choose an optimal design based on detailed considerations

I find my students have only surface knowledge of data structures. It would be helpful if the students have had a computability course that emphasizes formal languages. We have such a course at Clemson and about half my students have completed that course. I find that such courses may not be sufficient unless it, too, is a problem-based learning course.

Learners must have some architectural experience. I find that assembler-level programming, even among computer engineering students, is on the wane. My use of Gforth in my course was prompted by *Computer Architecture and Organization: An Integrated Approach* (Heuring and Jordan 2003) because the hardware description language in the text is easily simulated in Gforth.* Expert programmers know that the implementation language is just a language. Seniors (and graduate students) in computer science are unlikely to be that experienced, so mastering Forth represents a true departure from the steady C diet.

Without trying to compete with Watts Humphrey's views expressed in the *Personal Software Process* (1996), I find that students have no software

* A newer (2006) edition of this title is available but I have no experience with the text.

process—period. In particular, students have no well-developed idea of testing. This spills over into learning principle 2.

In summary, although the students all believe they know these subjects, the majority will not have deep operational experience. You must leave room in the syllabus for in-depth design and implementation. A perennial problem in my classes is Gforth support of the primitive `string` type.

1.3.1.2 Learning Principle 2: Foundational Knowledge

In discussions of knowledge, the type of knowledge referred to here is *implicit* knowledge. Implicit knowledge is held by the individual and by definition is not something that can be written down. In the expertise literature, implicit knowledge is what we ascribe to experts that make them experts.

Here, too, I find the students struggle. The expertise literature suggests that it takes about 10,000 hours of effort to gain expertise in a subject; that's about five years. Things that are obvious to the instructor are generally not obvious to the students due to this novice–expert difference. For example, an experienced instructor can reel off several alternative designs; novices struggle to get one.

This implicit knowledge comes into play at the higher levels of work: design. Figure 1.3 shows three considerations that are important in the education literature. Bloom's taxonomy (column one) is regarded by education specialists as a seminal insight; despite its age, the basic breakdown is useful. Instructors tend to work at the evaluation–synthesis–analysis level, whereas students tend to work at the application–comprehension–knowledge level. Anderson and Krathwohl (2001) reworked the taxonomy (columns two and three): (1) conceptual and factual knowledge are learning principle 1; (2) implicit knowledge is learning principle 2; and (3) metacognitive knowledge is learning principle 3. Implicit knowledge

		Metacognitive Knowledge
Evaluation	Create	
Synthesis	Evaluate	Implicit Knowledge
Analysis	Analyze	
Application	Apply	
Comprehension	Understand	Conceptual Knowledge
Knowledge	Remember	Factual Knowledge

Figure 1.3 Bloom's Taxonomy of Educational Objectives (Data from Bloom 1956; Anderson and Krathwohl 2001.)

(also called procedural knowledge) is the knowledge of *how* to do things: play the violin, add two columns of numbers, and recognize patterns.

1.3.1.3 Learning Principle 3: Metacognition

Metacognition is informally defined as "thinking about thinking." George Pólya (1957), for example, in the classic *How to Solve It*, emphasized that thinking about the solution of a problem was the last step. I use metacognition in two basic ways in this course. Every class period each student fills out a *one minute write* (OMW) exercise using a 3 × 5 scratch pad. Students answer two questions: (1) "What did you learn today?" and (2) "What questions do you have?" OMW exercises count as a participation grade *and* serve as a peek into the minds of the students. I consider this as a privileged communication with the student. I read them all and comment on them. The questions can get far off track, but they can also help the instructor spot troubled students.

The second way I use metacognition is in milestone reports. The milestone reports in general are factual, such as how the student exactly used a particular technique. It also contains a metacognitive exercise, "What did you learn?" I always look forward to this section, because—at least in my experience—the students are usually totally honest. The milestone report is 25 percent of the milestone grade so the students cannot take it lightly. I have included the reports of one of my A students to indicate what I consider a good series of reports (see Appendix A).

1.4 DETAILED SEMESTER PLAN

1.4.1 Milestone Maps

The organizational feature of the course is the milestone. The milestones interact as shown in Figure 1.4. Note that Milestone I is actually the last one in line as far as the total project is concerned, but it is done first so that the students understand what they are compiling to. In terms of more compiler-conventional terminology, the milestone control and data flow is shown in Figure 1.5.

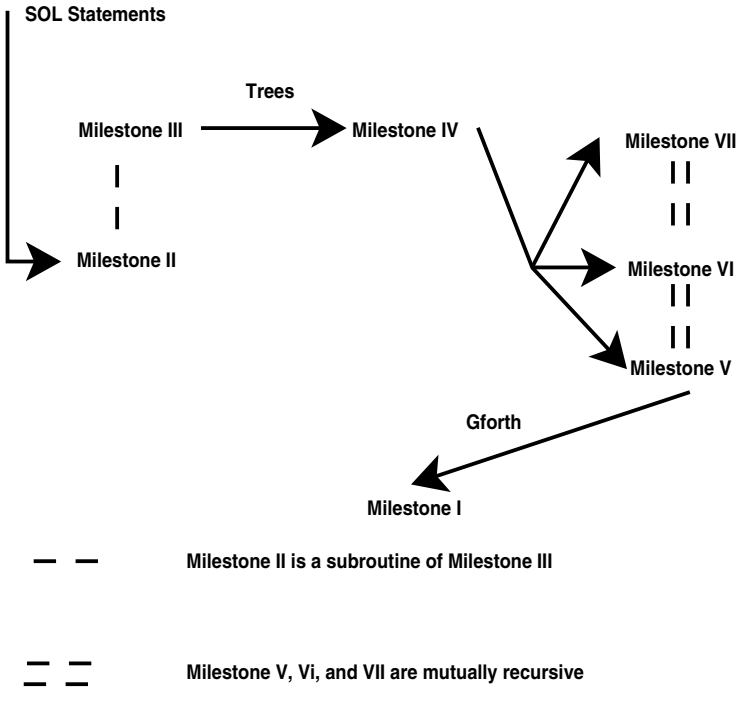

Figure 1.4 Relationship of the Milestones

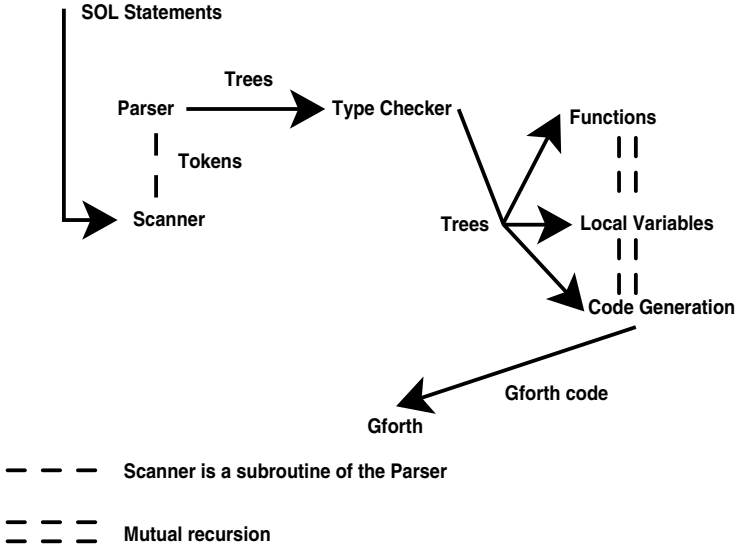

Figure 1.5 Control and Data Flow