

# Some Things That Computer Science Majors Should Know

Eric Allen

Sun Labs

eric.allen@sun.com

## 1. Introduction

In the broad field of computer science, one of the few sources of fundamental concepts that help us to characterize and to think about problems across its subdisciplines is that of programming languages. The field of programming languages provides us with the best understood mechanisms we have for rigorously specifying and reasoning about software systems. Moreover, the concepts that have been developed for characterizing programming languages have provided us with insights and mechanisms for reasoning about the behaviors of all computations. By failing to expose computer science majors to these concepts, we are depriving them of critical knowledge of their chosen field, making them less effective in their profession, whether they become software developers or researchers (or both). In this paper, I discuss some of the most critical topics in programming languages that every computer science major should learn, and why they are important.

## 2. Basic interpreter implementation

One of the most central notions in the field of programming languages is that of an interpreter, and many programming languages courses have been structured as a series of exercises in interpreter implementation. What is often not realized is that the knowledge gleaned through such interpreter implementation exercises has far reaching applicability. This applicability is due to the fact that the essential structure of many computer programs is to take structured input from a user, process

it, and return a result. Tax calculators, web browsers, printer drivers, PDF renderers, scripted robot control systems, spreadsheets, and video and audio players are just a few of the popular applications that programmers are called upon to implement that follow this structure. Implementing such a system competently involves checking the well-formedness of the input (a.k.a., parsing), possibly checking that the system exhibits a set of context-sensitive properties hold (a.k.a., type checking) and walking over the input to compute a result. This is exactly the structure of an interpreter; in fact, we can view the notion of an interpreter as the common structure that these programs share. The *language* interpreted by any such program consists of a definition of the valid inputs to such a program, together with a relation defined between inputs and results. *The fact that many of the programs listed are not currently viewed as interpreters by their designers, and that their inputs are not specified as languages with well-defined semantics, highlights the critical need to expose computer science majors to this general concept.* When programmers do not understand how to design and implement a simple interpreter, they cannot benefit from the wealth of knowledge accumulated about interpreter implementation, and they are likely to repeat decades-old mistakes.

For example, consider the issue of lexical vs. dynamic scoping. It is well known that dynamic scoping breaks encapsulation of code in a function; the very meaning of a function can change based on the context in which it is applied. There is no way for a programmer defining a function to know whether the body of a function will be semantically well-formed in a particular calling context; the free variables in the function's body could be bound to arbitrary values, with arbitrary run-time types. But despite all of this, we continue to see the development of new systems that make the fundamental mistake of introducing dynamic scoping. The most obvious examples are the barrage of scripting lan-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WXYZ '05 date, City.

Copyright © 2005 ACM [to be supplied]. . . \$5.00

guages that implement higher-order functions with dynamic scoping. But the problem appears in many other types of applications as well. Formulas entered in many popular spreadsheet programs break when transferred across tables because the references to cells they contain are not propagated appropriately. In operating systems that support symbolic links, if a link is moved to another directory, the path it keeps to the file it refers to is interpreted relative to its new location, rendering the link useless. As with other knowledge in the field, the lesson of the evils of dynamic scoping has applicability across many different kinds of systems and should be part of the working knowledge of every programmer.

### 3. Safety

Every computer science student should learn the basic notion of safety: “A safe language doesn’t allow you to violate its own abstractions.” But safety is an essential notion in specifying the behavior not just of programming languages, but of any computer system, as it defines the level of abstraction at which users should reason about a system. Reasoning at a well defined level of abstraction is essential for managing the complexity of modern computer systems, and (consequently) for defining reliable and predictable technology. Without safety, users of a system must think simultaneously at multiple levels of abstraction; not only is this inefficient, it makes it more difficult to isolate and diagnose bugs. Students who do not learn to appreciate this fundamental notion will produce lower quality software systems and specifications because of it.

### 4. Type Soundness

Type systems, a core part of the field of programming languages, are also our best understood framework for statically ensuring simple behavioral properties of computations. When designing a system where static assurance of properties is important, a programmer should consider whether a type system can be used (or devised) to perform the needed checks. But the ability to make such a consideration requires knowledge of how to define type systems and (if the task is to be done competently) how to state and prove the soundness of such a system. If students do not learn to understand type systems at this level, they do not understand the assurances that a static type checker provides them, and they cannot reason as effectively about the absence of errors in their own programs.

### 5. Safe Substitution and the Lambda Calculus

The notion of safe variable substitution (and the pitfalls of unsafe substitution) are at the heart of computer science. The need to perform substitution arises in many practical contexts; even business tools such as word processors and spreadsheets allow users to define macros to automate behavior. To competently design and build such macro facilities into software tools, programmers need to understand the notion of safe substitution, and (in particular) hygienic macro expansion. More generally, any endeavor in which we wish to write expressions containing variables benefits from an understanding of safe variable substitution. Of course, designers of general-purpose programming languages need to be aware of it.

A concrete discussion of safe substitution can be carried out in the context of the lambda calculus, which can be thought of as the simplest system embodying the semantics of safe substitution. Of course, it is a fascinating and important phenomenon that this simple calculus also serves as a Turing-complete model of computation; this alone should justify inclusion of the lambda calculus in the core of any computer science curriculum.

### 6. Side Effects and Purity

Often, it is only after studying the semantics of a language with side effects that students are able to reliably distinguish the notion of variable rebinding (e.g.,  $v := 2$ ) and reference mutation (e.g.,  $e.f := 2$ ). Moreover, often it is only after studying the semantics of a language with side effects that students appreciate the significant conceptual overhead that side effects introduce to the semantics of a programming language, and how much easier it is to reason about the errors in a program without side effects. All of this knowledge is important for effective programming and the avoidance of errors. If students are not exposed to this material in a programming languages course, where are they exposed to it?

### 7. Avoiding copy-and-paste coding

Understanding the advantage of factoring out common code to keep a single point of control for each functionally distinct element of a program is something every programmer should learn. But the extent to which it is possible to adhere to this principle depends on the features of the underlying programming language. For ex-

ample, first-class functions allow programmers to factor out common code that would have to be copied without them. Type systems invariably introduce constraints on programs that require copying. Macros allow programmers to factor out more code in some situations where even first-class functions require copying. If students do not understand the ways in which the features of a programming language affect their ability to factor out common code, they are not well positioned to choose a particular language for a particular task.

## 8. Continuations

The notion of a continuation, as a way to talk about what remains to be done in a computation, is a basic notion that, like decidability or algorithmic complexity, should be part of the vocabulary of anyone thinking about the nature of computation. This concept is so fundamental that it is baffling why it is not universally part of computer science curricula. When the notion is introduced into a domain unfamiliar with it, such as web servers, parallel computation, operating systems, etc., it often leads to dramatic simplifications and draws connections between subfields that have been previously unrecognized.

## 9. Term Rewriting Systems

Term rewriting systems, as they are used in defining the type systems and operational semantics of programming languages, really are essential to a proper understanding of computer science. Term rewriting systems are best thought of as a general mechanism for defining the behavior of any computer system. Indeed, many of the systems presented to undergraduates, such as Turing machines, digital logic circuits, finite automata, programming languages, type systems, logical calculi, networks, concurrent systems, etc., are naturally characterized as term rewriting systems. By presenting the formal tool of term rewriting to students, it is possible to provide a unifying framework in which students can understand many aspects of computer science.

One could unify what is taught in automata theory with programming languages through term rewriting systems; Turing machines, etc., can be modeled as term rewriting systems just as well as the lambda calculus. If students understand term rewriting and operational semantics, they can apply that knowledge to rigorously specify and reason about the behavior of many different kinds computer systems, including (of course) the

languages they write programs in. Just as importantly, they can draw connections between many systems that, on the surface, appear to be unrelated.

## 10. Conclusion

Many students are not aware of the nature of the material typically taught in a programming language course; a general misconception is that such courses provide merely a comparative study of various languages, rather than underlying concepts and ways of thinking about computation. When such a course is offered as an elective, many students understandably choose not to fit what seems to be a peripheral course into their schedules. In my own case, I did not take a programming languages course until I attended graduate school. Once exposed to the material, I soon grasped the fundamental relation this subject matter has to computer science, so much so that I ultimately decided to become a researcher in the field. Although in many other respects I believe I received a strong undergraduate education, I was amazed in retrospect that I was allowed to graduate with a computer science degree without having been exposed to a programming languages course. In my view, leaving a programming language course out of the core computer science curriculum is akin to leaving classical mechanics out of a core physics curriculum. Whether they realize it or not, our students are suffering because of this omission, as are the employers that hire them, and, ultimately, our field.