1. Introduction

This book is intended for a broad second course in computer science, one emphasizing principles wherever it seems possible at this level. While this course builds and amplifies what the student already knows about programming, it is not limited to programming. Instead, it attempts to use various programming models to explicate principles of computational systems. Before taking this course, the student should have had a solid one-semester course in computer programming and problem-solving, ideally using the Java™ language, since some of the presentation here uses Java. The philosophy taken in this course is that computer science topics, at an introductory level, are best approached in an integrated fashion (software, theory, and hardware) rather than as a series of individual isolated topics. Thus several threads are intertwined in this text.

1.1 The Purpose of Abstraction

This text touches, at least loosely, upon many of the most important *levels of abstraction* in computer systems. The term *abstract* may be most familiar to the student in the form of an adjective, as in *abstract art.* That association may, unfortunately, conjure a picture of being difficult to understand. In fact, the use we make of the term of *abstract* is to *simplify*, or *eliminate irrelevant detail*, as in the abstract of a published paper, which states the key ideas without details. In computer science, an abstraction is an intellectual device to simplify by eliminating factors that are irrelevant to the key idea. Much of the activity of computer science is concerned with inventing abstractions that simplify thought processes and system development.

The idea of levels of abstraction is central to managing complexity of computer systems, both software and hardware. Such systems typically consist of thousands to millions of very small components (words of memory, program statements, logic gates, etc.). To design all components as a single monolith is virtually impossible intellectually. Therefore, it is common instead to view a system as being comprised of a few interacting components, each of which can be understood in terms of *its* components, and so forth, until the most basic level is reached.

Thus we have the idea of *implementing* components on one level using components on the level below. The level below forms a set of abstractions used by the level being implemented. In turn, the components at this level may form a set of abstractions for the next level. For example, proposition logic (also called switching logic, or sometimes Boolean algebra) is an abstraction of what goes on at the gate-level of a computer. This logic is typically implemented using electronics, although other media are possible, for example mechanical logic or fluid logic. Switching logic, with the addition of memory components such as flip-flops, is the basis for implementing finite-state machines. Components such as registers and adders are built upon both logic and finite-state machines. These components implement the instruction set of the computer, another abstraction. The instruction set of the computer implements the programs that run on the

computer. A compiler or assembler is a program that translates a program written in a more user-friendly language into commands in the instruction set. The program might actually be an interpreter or "virtual machine" for a still higher level language, such as Java™.

We never can be sure how far these levels may go; what were once complete systems are now being fashioned into networks of computers, and those networks into networks, which themselves are replete with layers of abstractions (called "protocol stacks"). The same phenomenon occurs for software: compilers and interpreters for new languages may be built atop existing languages. Figure 1 is meant to summarize some of these important levels of abstraction.

The benefits of using abstraction are not unique to computer science; they occur in many disciplines and across disciplines. For example:

> *Chemistry is an abstraction of physics*: The purpose of chemistry is to understand molecular interactions without resorting to particle physics to explain every phenomenon.
>
> *Biology is an abstraction of chemistry*: The purpose of biology is to understand the growth and behavior of living things without resorting to molecular explanations for every aspect.
>
> *Genetics is an abstraction of biology*: The purpose of genetics is to understand the evolution of traits of living organisms. Genetics develops its own abstractions based on genes which don't require appealing to cells in every instance.

We could go on with this list. Note that we are saying *an abstraction* rather than *the abstraction*. It is not implied that the abstraction in question covers all aspects of the field being abstracted, nor that it is the only possible abstraction of that field.

Some of the specific advantages in treating systems by levels of abstraction are:

- Each level has its own definition and specification. This means that development using this level can proceed concurrently with development at the next level.

- A system can be developed by more than one individual, each a specialist in a particular aspect of construction. This is important since some systems are sufficiently ambitious that they would be impossible to develop by a single person in his or her lifetime.

- A system can evolve by evolving components separately; it is not necessary to re-implement the entire system when one component changes.

- A non-working system can be diagnosed by diagnosing components at their interfaces, rather than by exhaustively tracing the functions of all components.

Historically, implementations tended to come before abstractions. That is, a system got built, then abstractions were used to simplify its description. However, increasingly, we need to think of the abstractions first then go about implementing them. The abstractions provide a kind of specification of what is to be implemented.

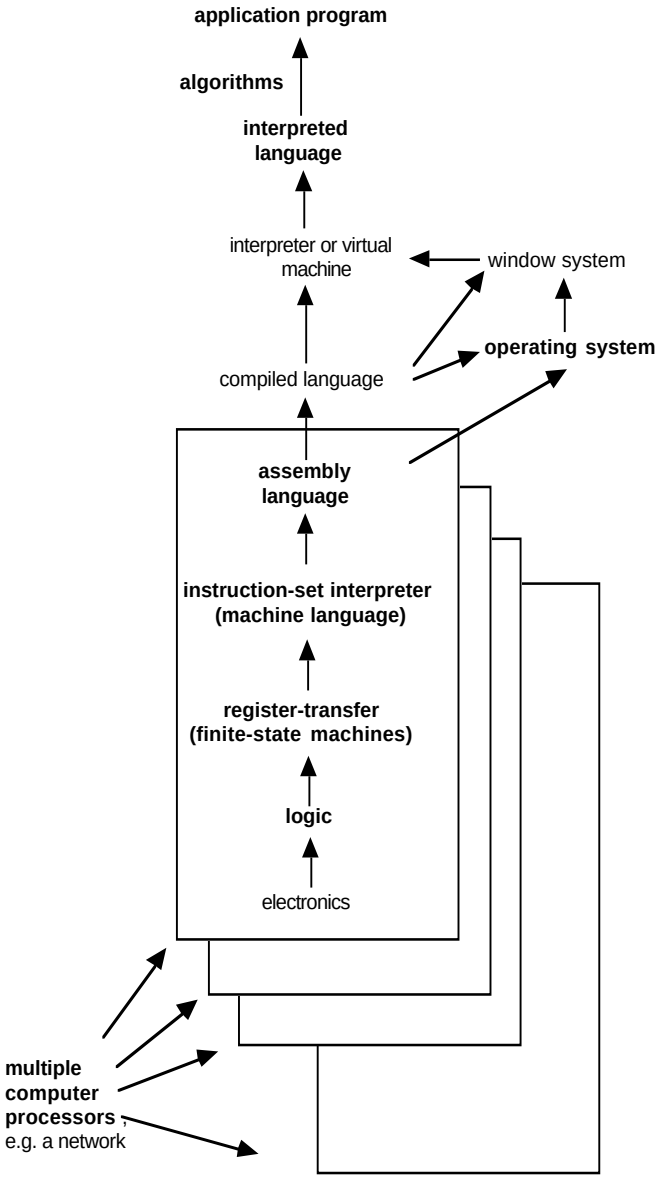

Figure 1: Typical levels of abstraction in computer science. The bolder items are ones given emphasis in this book.

Constructing abstractions is somewhat of an art, with or without a prior implementation. But understanding abstractions is rapidly becoming a way of life for those with any more than a casual relationship to computers. As new languages emerge, they are increasingly expressed using abstractions. A prime example is the object-oriented language Java™, which we use in part of this text to exemplify object-oriented concepts. The idea of inheritance is used heavily in the description of Java libraries. Inheritance hierarchies are miniature abstraction hierarchies in their own right.

At the same time, we are interested in how certain key abstractions are actually realized in hardware or in code, hence the word *implementation* in the title of the text. Having an understanding of implementation issues is important, to avoid making unreasonable assumptions or demands upon the implementor.

1.2 Principles

One might say that a science can be identified with its set of principles. Computer Science is relatively young as a discipline, and many of its principles are concerned with concepts lying at a depth which is beyond an introductory course such as this. Nevertheless, a conscious attempt is made to identify ideas as *named principles* wherever possible. Many of these principles are used routinely by computer scientists, programmers, and designers, but do not necessarily have standard names. By giving them names, we highlight the techniques and also provide more common threads for connecting the ideas. Although a modicum of theory is presented throughout the text, we are interested in imparting the ideas, rather than devoting attention to rigorous proofs.

While many of the points emphasized are most easily driven home by programming exercises, it is important to understand that the course is not *just* about programming, but rather about underlying conceptual continua that programming can best help illustrate.

1.3 Languages

The text is not oriented to a particular language, although a fair amount of time is spent on some language specifics. The educational "industry" has emerged from a point where Pascal was the most widely-taught introductory language. It was about ready to move on to C++ when Java™ appeared on the horizon. Java is a derivative of C++, which offers most of the object-oriented features of the latter, but omits some of the more confusing features. (As is usually the case, it introduces some new confusing features of its own.) For that reason, we start our discussion of object-orientation with Java. Another strong feature of Java, not heavily exploited in this text, is that working application programs, or “applets” as they are called, can be made available readily on the Internet. This text is not, in any way, to be regarded as a replacement for a handbook on any particular language, especially Java or C++. It is strongly advised that language handbooks be available for reference to specific language details that this text does not cover.

One purpose of a language is to give easy expression to a set of concepts. Thus, this book starts not with Java but rather a functional language *rex* of our own design. An interpreter for rex (implemented in C++) is provided. The rationale here is that there are many important concepts that, while they can be applied in many languages, are most cleanly illustrated using a functional language. To attempt to introduce them in Java would be to obscure the concepts with syntactic rubric. We later show how to transcribe the thinking and ideas into other languages. The current object-oriented bandwagon has much to recommend it, but it tends to overlook some of the important ideas in functional programming. In particular, functional programs are generally much easier to show *correct* than are object-oriented programs; there is no widely-accepted mathematical theory for the latter.

1.4 Learning Goals

The expected level of entry to this course is that students know basics of control-flow (*for* and *while* statements), are comfortable with procedures, know how and when to use arrays and structures, and understand the purposes of a type system. The student has probably been exposed to recursion, but might not be proficient at using it. The same is true for pointers. There may have been brief exposure to considerations behind choices of data structures, the analysis of program run-time, and the relationship between language constructs and their execution on physical processors. These things, as well as the structure of processors and relation to logic design, are likely to be gray areas and so we cover them from the beginning.

The student at this point is thus ready to tackle concepts addressed by this book, such as:

- information structures (lists, trees, directed graphs) from an abstract viewpoint, independent of particular data structure implementations
- recursion as a natural problem-solving technique
- functional programming as an elegant and succinct way to express certain specifications in an executable form
- objects and classes for expressing abstract data types
- underlying theoretical models that form the basis for computation
- inductive definitions and grammars for expressing language syntax and properties of sequences, and their application to the construction of simple parsers and interpreters from grammatical specifications
- proposition logic in specifying and implementing hardware systems and in program optimization
- predicate logic in specifying and verifying systems, and directly in programming

- advanced computing paradigms such as backtracking, caching, and breadth-first search
- use of techniques for analysis of program run-time complexity and the relationship to data-structure selection
- structure of finite-state machines how they extend to full processors
- assembly language, including how recursion is implemented at the assembly language level
- introduction to parallel processing, multithreading, and networking
- introduction to theoretical limitations of computing, such as problems of incomputability

These are among the topics covered in this book.

1.5 Structure of the Chapters

The chapters are described briefly as follows. There is more than adequate material for a one-semester course, depending on the depth of coverage.

1. ***Introduction*** is the current chapter.

2. ***Information Structures*** discusses various types of information, such as lists, trees, and directed graphs. We focus on the structure, and intentionally avoid getting into much programming until the next chapter.

3. ***High-Level Functional Programming*** discusses functions on the information structures used previously. The emphasis here is on thinking about high-level, wholesale, operations which can be performed on data.

4. ***Low-Level Functional Programming*** shows how to construct programs which carry out the high-level ideas introduced in the previous chapter. A rule-based approach is used, where each rule tries to express a thought about the construction of a function. We go into simple graph-processing notions, such as shortest path and transitive closure.

5. ***Implementing Information Structures*** presents methods of implementing a variety of information and structures in Java, including many of the structures discussed in earlier chapters.

6. ***States and Transitions*** discusses the basis of state-oriented computation, which is a prolog to object-oriented computation. It shows how state can be modeled using the

framework introduced in previous chapters. We show how conventional imperative programs can be easily transformed into functional ones. We illustrate the idea of a Turing machine and discuss its acceptance as a universal basis for computation.

7 . ***Object-Oriented Programming*** introduces object-oriented concepts for data abstraction, using Java as the vehicle. We explore ideas of polymorphism, and construct a model of polymorphic lists, matching the kind of generality available in functional programming systems. We describe the implementation of higher-order functions. We include a discussion of the uses of inheritance for normalizing software designs.

8 . ***Grammars and Parsing*** introduces the concept of grammars for specifying languages. It also shows the construction of simple parsers for such languages. The idea here is that in many cases we have to solve not just one problem but rather an entire family of problems. Indeed, we may need to provide such a language to a community of users who do not wish to get involved with a general purpose programming language. Inventing a language in which to express a family of problems, and being able to construct an interpreter for that language, is viewed as a helpful skill.

9. ***Proposition Logic*** begins with basic ideas of proposition logic from a functional point of view. We the show role these ideas play in hardware design, and go into some of the theory of simplification of logical expressions. Physical bases for computing are mentioned briefly

10. ***Predicate Logic*** introduces predicate logic and demonstrate its use in specifying and proving programs. We also show how predicate logic can be used for direct programming of databases and other applications, using the Prolog language.

11. ***Complexity*** introduces the idea of measuring the running time of a program across a wide spectrum of inputs. We use the "O" notation, defining it in a simplified way appropriate to the application at hand, analyzing programs. We show how programs can be analyzed when they are decomposed into sequential compositions, loops, and recursion. We use sorting and searching applications for many of the examples. We also mention hashing and related techniques.

12. ***Finite-State Machines*** introduces various finite-state machine models and how they are implemented. We work our way into the implementation of simple digital sub-systems based on finite-state machines. We conclude with a discussion of data communication issues, such as the use of 3-state buffer devices.

13. ***Stored-Program Computing*** talks about the structure and programming of stored-program computers from a fairly low level. This ties together programming concepts and concepts from finite-state computing. We present a simulated computer, the ISC, and its assembly language.

14. ***Parallel Computing*** discusses issues related to performing multiple computations at the same time. We review cellular automata, data-parallel computing, process-oriented approaches, suitable for multiple-instruction-stream/multiple-data-stream computers, and discuss the importance of this emerging area in the network-based computers of the future.

15. ***Limitations of Computing*** mentions some of the logical and physical limitations of computing. We discuss algorithmic lower bounds, the limitations of finite-state machines, the notions of incomputability and intractability, and the glitch phenomenon.

Figure 2 gives an approximate dependence among chapters.

1.6 How the Problems are Rated

We use a dot notation to visually suggest a problem's difficulty:

- G: Intended to be workable based on just an understanding of the prior readings.

•• PG: Intended to be workable based on an understanding of readings plus a little effort.

••• PG13: Workable with a little effort and perhaps a hint or two.

•••• R: Intended for mature audiences; requires substantial effort and possibly extra insight or perseverance.

••••• NC17: Obscenely difficult; might be the subject of a research paper, past or future. Intended for perspective, not necessarily to be done in the mainstream of the course.

Exercises

1. •• Cite an example from your own experience of an abstraction and its implementation.

2. ••• Identify some areas outside of computer science, such as in chemistry, music, dance, etc. where abstractions are heavily used. Give details.

3. •••• Identify some areas outside of computer science where several *levels* of abstraction are used.

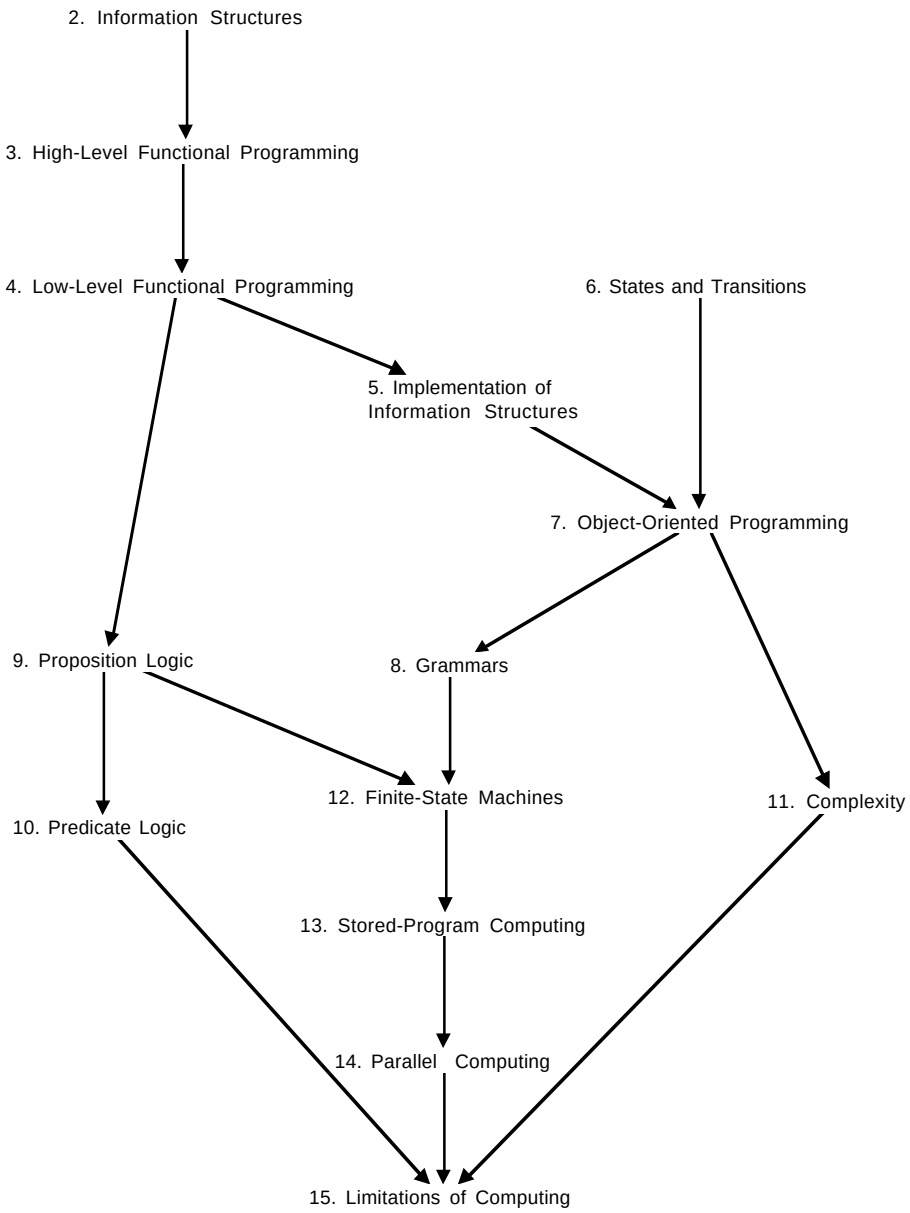

Figure 2: Chapter Dependence

1.7 Further Reading

Each chapter lists relevant further reading. In many cases, original sources are cited, which are sometimes not very light reading. We try to provide a qualitative estimate of difficulty at the end of each annotation. The following tend to lighter surveys, which cover some of the ideas in the text (as well as others) from different perspectives, but still at a more-or-less less technical level.

Alan W. Biermann, *Great Ideas in Computer Science*, M.I.T. Press, 1990. [A textbook approach to introductory examples. Algorithms are presented in Pascal. Easy to moderate.]

Glenn Brookshear, *Computer Science – An Overview*, Third Edition, Benjamin/Cummings, 1991. [Easy.]

Richard P. Feynman, Feynman Lectures on Computation, Edited by J.G. Hey and Robin W. Allen, Addison-Wesley, 1996. [A famous physicist talks about computation and computer science; moderate.]

A.K. Dewdney, *The (New) Turing Omnibus – 66 Excursions in Computer Science*, Computer Science Press, 1993. [Short (3-4 page) articles on a wide variety of computer science topics. Algorithms are presented in pseudo-code. Easy to moderate.]

David Harel, *Algorithmics – The Spirit of Computing*, Addison-Wesley, 1987. [Moderate.]

Anthony Ralston and Edwin D. Reilly, *Encyclopedia of Computer Science*, Van Nostrand Reinhold, 1993.

1.8 Acknowledgment

The following is a partial list of individuals whom the author wishes to thank for comments and corrections: Jeff Allen, James Benham, Jason Brudvik, Andrew Cosand, Dan Darcy, Jason Dorsett, Zachary Dodds, Elecia Engelmann, Kris Jurka, Ran Libeskind-Hadas, Eran Karmon, Geoffrey Kuenning, Stephen Rayhawk, David Rudel, Chuck Scheid, Virginia Stoll, Chris Stone, Ian Weiner, and Wynn Yin.