1 Introduction

1.1 Overview

This chapter briefly describes:

- what this book is about
- what this book tries to do
- what this book tries not to do
- a useful feature of the book: the exercises.

1.2 What this Book is About

This book is about two key topics of computer science, namely *computable languages* and *abstract machines*.

Computable languages are related to what are usually known as "formal languages". I avoid using the latter phrase here because later on in the book I distinguish between *formal* languages and *computable* languages. In fact, computable languages are a special type of formal languages that can be processed, in ways considered in this book, by computers, or rather *abstract machines* that *represent* computers.

Abstract machines are formal computing devices that we use to investigate properties of *real* computing devices. The term that is sometimes used to describe

A.P. Parkes, *A Concise Introduction to Languages and Machines*,
DOI: 10.1007/978-1-84800-121-3_1, © Springer-Verlag London Limited 2008

abstract machines is *automata*, but that sounds too much like real machines, in particular the type of machines we call *robots*.

This book assumes that you are a layperson, in terms of computer *science*. If you are a computer science undergraduate, as you might be if you are reading this book, you may by now have written many programs. So, in this introduction we will draw on your experience of programming to illustrate some of the issues related to formal languages that are introduced in this book.

The programs that you write are written in a formal language. They are expressed in text which is presented as input to another program, a *compiler* or perhaps an *interpreter*. To the compiler, your program text represents a *code*, which the compiler knows exactly how to handle, as the rules by which that code is constructed are completely specified inside the compiler. The type of language in which we write our programs (the *programming language*), has such a well-defined syntax (rules for forming acceptable programs) that a machine can decide whether or not the program you enter has the right even to be considered as a proper program. This is only part of the task of a compiler, but it's the part in which we are most interested.

For whoever wrote the compiler, and whoever uses it, it is very important that the compiler does its job properly in terms of deciding that your program is syntactically valid. The compiler writer would also like to be sure that the compiler will always *reject* syntactically *in*valid programs as well as accepting those that are syntactically valid. Finally, the compiler writer would like to know that his or her compiler is not wasteful in terms of precious resources: if the compiler is more complex than it needs to be, if it carries out many tasks that are actually unnecessary, and so on. In this book we see that the solutions to such problems depend on the type of language being considered.

Now, because your program is written in a formal language that is such that another program can decide if your program *is* a program, the programming language is a computable language. A *computable* language then, is a *formal* language that is such that a computer can understand its syntax. Note that we have not discussed what your program will actually *do*, when it's run: the compiler doesn't really understand that at all. The compiler is just as happy to compile a program that doesn't do what you intended as it is to compile one that does (as you'll know, if you've ever done any programming).

The book is in two parts. Part 1: Languages and Machines is concerned with the relationship between different types of formal languages and different types of theoretical machines (abstract machines) that can serve to process those languages, in ways we consider later. So, the book isn't really *directly* concerned with *programming* languages. However, much of the material in the first part of the book is highly relevant to programming languages, especially in terms of providing a useful theoretical background for compiler writing, and even programming language design. For that reason, some of the examples used in the book are from programming languages (particularly Pascal in fact, but you need not be familiar with the language to appreciate the examples).

In Part 1, we study four different types of formal languages and see that each of these types of formal language is associated with a particular type of abstract machine. The four types of language we consider were actually defined by the American linguist Noam Chomsky; the classification of formal languages he defined has come to be called the *Chomsky hierarchy*. It *is* a hierarchy, since it defines a general type of language (type 0), then a restricted version of that general type (type 1), then a restricted version of *that* type (type 2), and then finally the most restricted type of all (type 3).

The types of language considered are very simple to define, and even the most complex of abstract machines is also quite simple. What's more, all of the types of abstract machine we consider in this book can be represented in diagrammatic form. The most complex abstract machine we look at is called the Turing machine (TM), after Alan Turing, the mathematician who designed it. The TM is, in some senses, the most powerful computational device possible, as you will see in Part 2 of the book.

The chapters of Part 1 are shown in Figure 1.1.

By the end of Part 1 of the book you should have an intuitive appreciation of computable languages. Part 2: Machines and Computation investigates computation in a wider sense. We see that we can discuss the types of computation carried out by *real* computers in terms of our abstract machines. We see that a machine, called a finite state transducer, which appears to share some properties with real computers, is not really a suitable general model of real computers. Part of the reason for this is that this machine cannot do multiplication or division. We see that the *Turing machine* is capable of multiplication and division, and any other tasks that real computers can do. We even see that the TM can run programs. What is more, since the TM effectively has unlimited storage capacity, it turns out to be *more* powerful than any real computer could ever be.

One interesting property of TMs is that we cannot make them any more powerful than they are by adding extra computational facilities. Nevertheless, TMs use only one data structure, a potentially infinite one-dimensional array of symbols, called a tape, and only one type of instruction. A TM can simply read a symbol from its tape, replace that symbol by another, and then move to the next symbol on the right or left in the tape. We find that if a TM is designed so that it carries out many processes simultaneously, or uses many additional tapes, it cannot perform any more computational tasks than the basic serial one-tape version. In fact, it has been established that no other formalisms we might create

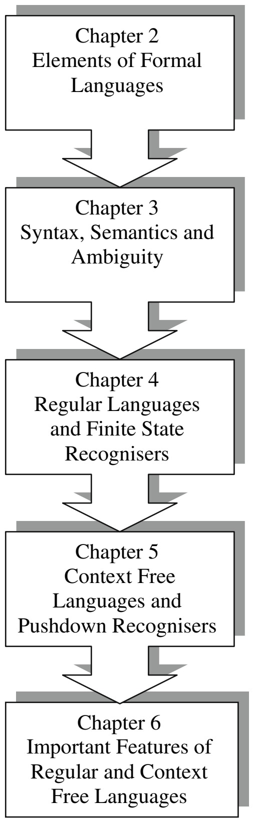
Chapter 2
Elements of Formal Languages
Chapter 3
Syntax, Semantics and Ambiguity
Chapter 4
Regular Languages and Finite State Recognisers
Chapter 5
Context Free Languages and Pushdown Recognisers
Chapter 6
Important Features of Regular and Context Free Languages

Figure 1.1 The chapters of Part 1: Languages and Machines.

for representing computation give us any more functional power than does a TM. This assertion is known as *Turing's thesis*.

We see that we can actually take any TM, code it and its data structure (tape) as a sequence of zeros and ones, and then let another TM run the original TM as if it were a program. This TM can carry out the computation described by any TM. We thus call it the universal Turing machine (UTM). However, it is simply a standard TM, which, because of the coding scheme used, only needs to expect either a zero or a one every time it examines a symbol on its tape. The UTM is an abstract counterpart of the real computer.

The UTM has unlimited storage, so no real computer can exceed its power. We use this fact as the basis of some extremely important results of computer science, one of which is to see that the following problem cannot be solved, in the general case:

Given any program, and appropriate input values to that program, will that program terminate when run on that input?

In terms of TMs, rather than programs, this problem is known as the *halting problem*. We see that the halting problem is unsolvable,[1] which has implications both for the real world of computing and for the nature of formal languages. We also see that the halting problem can be used to convince us that there are formal languages that cannot be processed by any TM, and thus by any program. This enables us to finally define the relationship between computable languages and abstract machines.

At the end of Part 2, our discussion about abstract machines leads us on to a topic of computer science, algorithm complexity, that is very relevant to

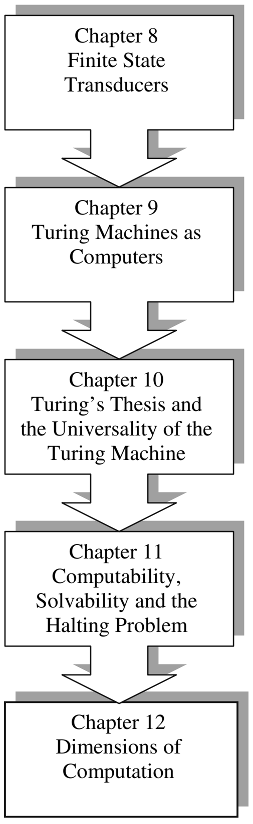
Chapter 8
Finite State Transducers
Chapter 9
Turing Machines as Computers
Chapter 10
Turing's Thesis and the Universality of the Turing Machine
Chapter 11
Computability, Solvability and the Halting Problem
Chapter 12
Dimensions of Computation

Figure 1.2 The chapters of Part 2: Machines and Computation.

[1] You have probably realised that the halting problem is solvable in certain cases.

programming. In particular, we discuss techniques that enable us to predict the running time of algorithms, and we see that dramatic savings in running time can be made by certain cleverly defined algorithms.

Figure 1.2 shows the chapters of Part 2 of the book.

1.3 What this Book Tries to Do

This book attempts to present formal computer science in a way that you, the student, can understand. This book does not assume that you are a mathematician. It assumes that you are not particularly familiar with formal notation, set theory, and so on. As far as is possible, excessive formal notation is avoided. When formal notation or jargon is unavoidable, the formal definitions are usually accompanied by short explanations, at least for the first few times they appear.

Overall, this book represents an understanding of formal languages and abstract machines which lies somewhere beyond that of a layperson, but is considerably less than that of a mathematician. There is a certain core of material in the subject that any student of computer science, or related disciplines, should be aware of. Nevertheless, many students do not become aware of this important and often useful material. Sometimes they are discouraged by the way it is presented. Sometimes, books and lecturers try to cover too much material, and the fundamentals get lost along the way.

Above all, this book tries to highlight the connections between what might initially appear to be distinct topics within computer science.

1.4 What this Book Tries Not to Do

This book tries not to be too formal. References to a small number of formal books are included in the section “Further Reading”, at the end of the book. In these books you will find many theorems, usually proved conclusively by logical proof techniques. The “proofs” appearing in this book are included usually to establish something absolutely central, or that we need to assume for subsequent discussion. Where possible, such “proofs” are presented in an intuitive way.

This is not to say that proofs are unimportant. I assume that, like me, you are the type of person who has to convince themselves that something is right before you really accept it. However, the proofs in this book are presented in a different way from the way that proofs are usually presented. Many results are established “beyond reasonable doubt” by presenting particular examples and encouraging the reader to appreciate ways in which the examples can be generalised. Of course, this is not really sound logic, as it does not argue throughout in terms of the general case. Some of the end-of-chapter exercises present an opportunity to practise or complete proofs. Such exercises often include hints and/or sample answers.

1.5 The Exercises

At the end of each of Chapters 2 to 12, you will find a small number of exercises to test and develop your knowledge of the material covered in that chapter. Most of the exercises are of the "pencil and paper" type, though some of them are medium-scale programming problems. Any exercise marked with a dagger (†) has a sample solution in the "Solutions to Selected Exercises" section near the end of the book. You do not need to attempt any of the exercises to fully understand the book. However, although the book attempts to make the subject matter as informal as possible, in one very important respect it is very much like maths: *you need to practise applying the knowledge and skills you learn or you do not retain them.*

Finally, some of the exercises give you an opportunity to investigate additional material that is not covered in the chapters themselves.

1.6 Further Reading

A small section called "Further Reading" appears towards the end of the book. This is not meant to be an exhaustive list of reading material. There are many other books on formal computer science than are cited here. The further reading list also refers to books concerned with other fields of computer science (e.g. computer networks) where certain of the formal techniques in this book have been applied. Brief notes accompany each title cited.

1.7 Some Advice

Most of the material in this book is very straightforward, though some requires a little thought the first time it is encountered. Students of limited formal mathematical ability should find most of the subject matter of the book reasonably accessible. You should use the opportunity to practise provided by the exercises, if possible. If you find a section really difficult, ignore it and go on to the next. You will probably find that an appreciation of the overall result of a section will enable you to follow the subsequent material. Sections you omit on first reading may become more comprehensible when studied again later.

You should not allow yourself to be put off if you cannot see immediate applications of the subject matter. There have been many applications of formal languages and abstract machines in computing and related disciplines, some of which are referred to by books in the "Further Reading" section.

This book should be interesting and relevant to any intelligent reader who has an interest in computer science and approaches the subject matter with an open mind. Such a reader may then see the subject of languages and machines as an explanation of the simple yet powerful and profound abstract computational processes beneath the surface of the digital computer.