Chapter 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 three key topics of computer science, namely *computable languages*, *abstract machines*, and *logic*.

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

The logic part of the book considers using different types of formal logic to represent things and reason about them. The logics we consider all play a very important role in computing. They are *Boolean* logic, *propositional* logic, and *first order predicate* logic (FOPL).

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 is 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 is run: the compiler does not really understand that at all. The compiler is just as happy to compile a program that does not do what you intended as it is to compile one that does (as you will know, if you have ever done any programming).

The book is in three 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 is not 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 languages we consider were actually defined by the American linguist Noam Chomsky, and 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 is 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 sense, 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.

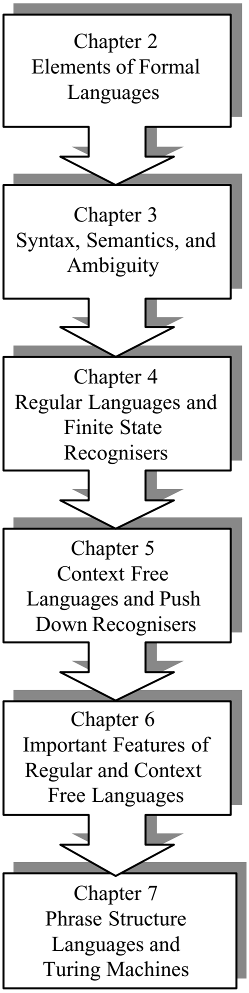

Chapter 2
Elements of Formal Languages
Chapter 3
Syntax, Semantics, and Ambiguity
Chapter 4
Regular Languages and Finite State Recognisers
Chapter 5
Context Free Languages and Push Down Recognisers
Chapter 6
Important Features of Regular and Context Free Languages
Chapter 7
Phrase Structure Languages and Turing Machines

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

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 TM 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 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 TM (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 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.

The final part of the book, Part 3: Computation and Logic considers formal logical systems as tools for representation, reasoning and computation. We first consider the system known as *Boolean logic*. This form of logic is important because it forms the foundation of digital computer circuitry. We show how Boolean logic can be used to solve problems, and we introduce the *truth table* as a tool for checking the validity of logical statements.

In the same chapter, we consider *propositional logic*. This logic includes the basic operators of Boolean logic, along with an additional operator that represents implication ("if x is true then so is y"). We consider logical rules of inference, which enable us to solve problems by applying the rules to a propositional representation of the problem. We discover that propositional logic is not sufficiently powerful to represent certain things, which leads us on to what is known as FOPL.

FOPL occupies the final two chapters of the book. We see how FOPL can be used to represent and reason about properties that apply to all objects in a domain, and properties that apply to one or more objects in that domain. We consider what is called *classical* FOPL reasoning, when we represent a domain as a set of FOPL

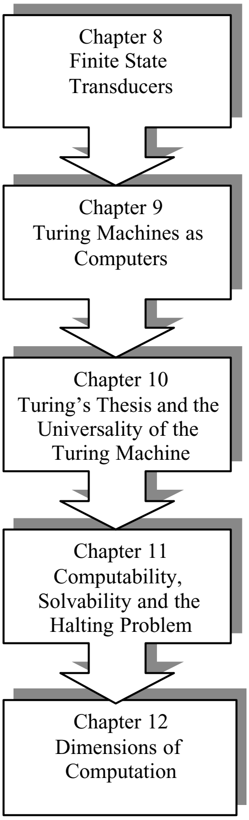


Figure 1.2. The chapters of Part 2: Machines and Computation

statements and then use rules of inference to attempt to derive a statement we believe to be true, and are thus trying to prove. In the final chapter we consider a computational technique for reasoning in FOPL known as *resolution*. We also relate the discussion of resolution to the real world of programming, by briefly considering the logic programming language PROLOG.

Throughout Part 3, we relate our discussions to issues from the first two parts of the book. We discuss the linguistic nature of logic and the time and space requirements of logical reasoning. We end the book by discussing the relationship between two key results of twentieth century mathematics (Gödel's theorem and Turing's thesis) and their implications for computer science.

Figure 1.3 shows the titles of the chapters of Part 3.

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 possible, excessive formal notation is avoided. When formal

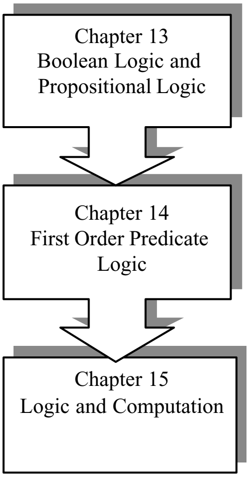


Figure 1.3. The chapters of Part 3: Computation and Logic

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, abstract machines and logic 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 convinced 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 a 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 practice or complete proofs. Such exercises often include hints, and/or sample answers.

1.5 The Exercises

At the end of each of Chapters 2–15, 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 practice 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 (for example, 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 practice 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 grammar, abstract machines and logic 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, machines and logic as an explanation of the simple yet powerful and profound abstract computational processes beneath the surface of the digital computer.

Notes

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

Part 1

Languages and Machines

Chapter 2

Elements of Formal Languages

2.1 Overview

In this chapter, you learn about:

- the building blocks of formal languages: *alphabets* and *strings*
- *grammars* and *languages*
- a way of classifying grammars and languages: the *Chomsky hierarchy*
- how formal languages relate to the definition of programming languages.

… and you are introduced to:

- writing definitions of sets of strings
- producing *sentences* from grammars
- using the notation of formal languages.

2.2 Alphabets

An alphabet is a finite collection (or set) of symbols. The symbols in the alphabet are entities which cannot be taken apart in any meaningful way, a property which leads to them being sometimes referred to as *atomic*. The symbols of an alphabet are simply the "characters", from which we build our "words". As already said, an alphabet is *finite*. That means we could define a program that would print out its elements (or members) one by one, and (this last part is very important) the program would terminate sometime, having printed out each and every element.

For example, the small letters you use to form words of your own language (for example, English) could be regarded as an alphabet, in the formal sense, if written down as follows:

$$\{a, b, c, d, e, \ldots, x, y, z\}.$$

The digits of the (base 10) number system we use can also be presented as an alphabet:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

2.3 Strings

A string is a finite sequence of zero or more symbols taken from a formal alphabet. We write down strings just as we write the words of this sentence, so the word “strings” itself could be regarded as a *string* taken from the alphabet of letters, above. Mathematicians sometimes say that a string taken from a given alphabet is a string *over* that alphabet, but we will say that the string is *taken from* the alphabet. Let us consider some more examples. The string *abc* is one of the many strings which can be taken from the alphabet $\{a, b, c, d\}$. So is *aabacab*. Note that duplicate symbols are allowed in strings (unlike in sets). If there are no symbols in a string it is called the *empty string*, and we write it as ε (the Greek letter *epsilon*), though some write it as λ (the Greek letter *lambda*).

2.3.1 Functions that Apply to Strings

We now know enough about strings to describe some important functions that we can use to manipulate strings or obtain information about them. Table 2.1 shows the basic string operations (note that x and y stand for *any* strings).

You may have noticed that strings have certain features in common with *arrays* in programming languages such as Pascal, in that we can index them. To index a string, we use the notation x_i, as opposed to something like $x[i]$. However, strings actually have more in common with the list data structures of programming languages such as LISP or PROLOG, in that we can concatenate two strings together, creating a new string. This is like the *append* function in LISP, with strings corresponding to lists, and the empty string corresponding to the empty list. It is only possible to perform such operations on arrays if the programming language allows

Table 2.1. The basic operations on strings

Operation	Written as	Meaning	Examples and comments
Length	$\|x\|$	The number of symbols in the string x	$\|abcabca\| = 7$ $\|a\| = 1$ $\|\varepsilon\| = 0$
Concatenation	xy	The string formed by writing down the string x followed immediately by the string y concatenating the empty string to any string makes no difference	let $x = abca$ let $y = ca$ then: $xy = abcaca$ let $x = <any\ string>$ then: $x\varepsilon = x$ $\varepsilon x = x$
Power	x^n, where n is a whole number $\geqslant 0$	The string formed by writing down n copies of the string x	let $x = abca$ then: $x^3 = abcaabcaabca$ $x^1 = x$ *Note*: $x^0 = \varepsilon$
Index	x_i, where i is a whole number	The ith symbol in the string x (i.e. treats the string as if it were an array of symbols)	let $x = abca$ then: $x_1 = a$ $x_2 = b$ $x_3 = c$ $x_4 = a$

arrays to be of dynamic size (which Pascal, for example, does not). However, many versions of Pascal now provide a special dynamic "string" data type, on which operations such as concatenation can be carried out.

2.3.2 Useful Notation for Describing Strings

As described above, a string is a sequence of symbols taken from some alphabet. Later, we will need to say such things as:

> suppose x stands for some string taken from the alphabet A.

This is a rather clumsy phrase to have to use. A more accurate, though even clumsier, way of saying it is to say:

> x is an element of the set of all strings which can be formed using zero or more symbols of the alphabet A.

There is a convenient and simple notational device to say this. We represent the latter statement as follows:

$$x \in A^*,$$

which relates to the English version as shown in Figure 2.1.

On other occasions, we may wish to say something like:

> x is an element of the set of all strings which can be formed using *one* or more symbols of the alphabet A,

for which we write:

$$x \in A^+,$$

which relates to the associated verbal description as shown in Figure 2.2.

Suppose we have the alphabet $\{a, b, c\}$. Then $\{a, b, c\}^*$ is the set

$$\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, aba, abb, abc, \ldots\}.$$

Clearly, for any non-empty alphabet (i.e. an alphabet consisting of one or more symbols), the set so defined will be *infinite*.

Earlier in the chapter, we discussed the notion of a program printing out the elements of a *finite* set, one by one, terminating when all of the elements of the set had been printed. If A is some alphabet, we could write a program to print out all the strings in A^*, one by one, such that each string only gets printed out once. Obviously, such a program would *never* terminate (because A^* is an *infinite* set), but we could design the program so that any string in A^* would appear within a finite period of time. Table 2.2 shows a possible method for doing this (as an

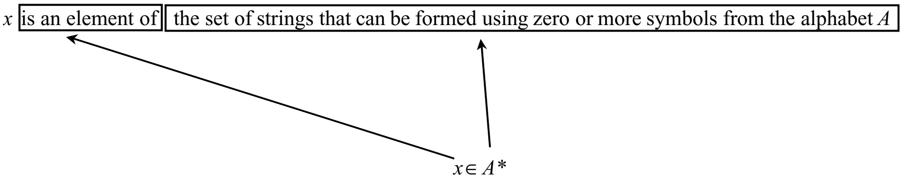

x
is an element of
the set of strings that can be formed using zero or more symbols from the alphabet A
x∈ A*

Figure 2.1. How we specify an unknown, possibly empty, string

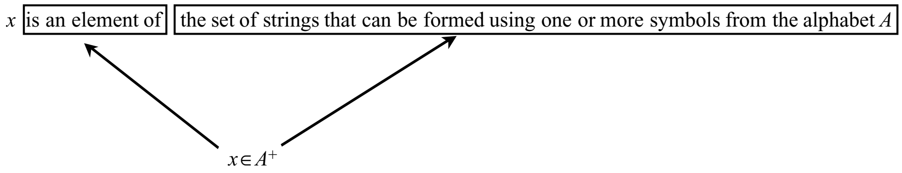

x
is an element of
the set of strings that can be formed using one or more symbols from the alphabet A
$x \in A^+$

Figure 2.2. How we specify an unknown, non-empty, string

exercise, you might like to develop the method into a program in your favourite programming language). The method is suggested by the way the first few elements of the set A^*, for $A = \{a, b, c\}$ were written down, above.

An infinite set for which we can print out any given element within a finite time of starting the program is known as a *countably infinite* set. I suggest you think carefully about the program in Table 2.2, as it may help you to appreciate just what is meant by the terms "infinite" and "finite". Clearly, the program specified in Table 2.2 would never terminate. However, on each iteration of the loop, *i* would have a finite value, and so any string printed out would be finite in length (a necessary condition for a string). Moreover, any string in A^* would appear after a finite period of time.

Table 2.2. Systematically printing out all strings in A^*

```
begin
  <print some symbol to represent the empty string>
  i := 1
  while i >= 0 do
    <print each of the strings of length i>
    i := i + 1
  endwhile
end
```

2.4 Formal Languages

Now we know how to express the notion of all of the strings that can be formed by using symbols from an alphabet, we are in a position to describe what is meant by the term *formal language*. Essentially, a formal language is simply any set of strings formed using the symbols from any alphabet. In set parlance, given some alphabet A,

> a *formal language* is "any (proper or non-proper) subset of the set of all strings which can be formed using zero or more symbols of the alphabet A".

The formal expression of the above statement can be seen in Figure 2.3.

A *proper* subset of a set is not allowed to be the *whole* of a given set. For example, the set $\{a, b, c\}$ is a proper subset of the set $\{a, b, c, d\}$, but the set $\{a, b, c, d\}$ is not.

A *non-proper* subset is a subset that is allowed to be the whole of a set.

So, the above definition says that, for a given alphabet, A, A^* is a formal language, and so is any subset of A^*. Note that this also means that the empty set, written "{ }" (sometimes it is written as $\emptyset$) is also a formal language, since it is a subset of A^* (the empty set is a subset of *any* set).

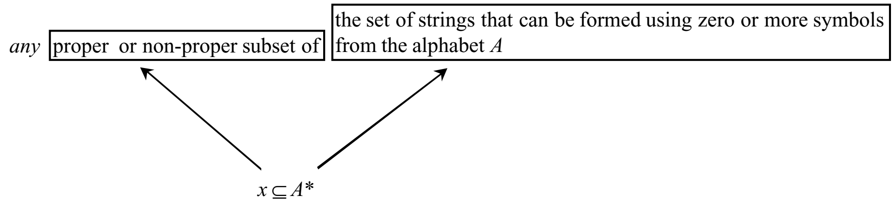

any
proper or non-proper subset of
the set of strings that can be formed using zero or more symbols from the alphabet A
x ⊆ A*

Figure 2.3. The definition of a *formal language*

A formal language, then, is any set of strings. To indicate that the strings are part of a language, we usually call them *sentences*. In some books, sentences are called *words*. However, while the strings we have seen so far are similar to English words, in that they are unbroken sequences of alphabetic symbols (for example, *abca*), later we will see strings that are statements in a programming language, such as

```
if i > 1 then x := x + 1.
```

It seems peculiar to call a statement such as this a "word".

2.5 Methods for Defining Formal Languages

Our definition of a formal language as being a set of strings that are called *sentences* is extremely simple. However, it does not allow us to say anything about the form of sentences in a particular language. For example, in terms of our definition, the Pascal programming language, by which we mean "the set of all syntactically correct Pascal programs", is a subset of the set of all strings which can be formed using symbols found in the character set of a typical computer. This definition, though true, is not particularly helpful if we want to write Pascal programs. It tells us nothing about what makes one string a Pascal program, and another string not a Pascal program, except in the trivial sense that we can immediately rule out any strings containing symbols that are not in the character set of the computer. You would be most displeased if, in attempting to learn to program in Pascal, you opened the Pascal manual to find that it consisted entirely of one statement which said: "Let *C* be the set of all characters available on the computer. Then the set of compilable Pascal programs, *P*, is a subset of *C**".

One way of informing you what constitutes "proper" Pascal programs would be to write all the proper ones out for you. However, this would also be unhelpful, albeit in a different way, since such a manual would be infinite, and thus could never be completed. Moreover, it would be a rather tedious process to find the particular program you required.

In this section we discover three approaches to defining a formal language. Following this, every formal language we meet in this book will be defined according to one or more of these approaches.

2.5.1 Set Definitions of Languages

Since a language is a *set* of strings, the obvious way to describe some language is by providing a set definition. Set definitions of the formal languages in which we are interested are of three different types, as now discussed.

The first type of set definition we consider is only used for the smallest finite languages, and consists of writing the language out in its entirety. For example,

$$\{\varepsilon, abc, abbba, abca\}$$

is a language consisting of exactly four strings.

The second method is used for infinite languages, but those in which there is some obvious pattern in all of the strings that we can assume the reader will induce when presented with sufficient instances of that pattern. In this case, we write out sufficient sentences for the pattern to be made clear, then indicate that the pattern should be allowed to continue indefinitely, by using three dots "...". For example,

$$\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$$

suggests the infinite language consisting of all strings which consist of one or more *a*s followed by one or more *b*s and in which the number of *a*s equals the number of *b*s.

The final method, used for many *finite* and *infinite* languages, is to use a set definition to specify how to construct the sentences in the language, i.e. provide a function to deliver the sentences as its output. In addition to the function itself, we must provide a specification of how many strings should be constructed. Such set definitions have the format shown in Figure 2.4.

For the "function to produce strings", of Figure 2.4, we use combinations of the string functions we considered earlier (*index*, *power*, and *concatenation*). A language that was defined immediately above,

> all strings which consist of one or more as followed by one or more *b*s and in which the number of *a*s equals the number of *b*s

can be defined using our latest method as:

$$\{a^i b^i : i \geqslant 1\}.$$

The above definition is explained in Table 2.3.
From Table 2.3 we can see that $\{a^i b^i : i \geqslant 1\}$ means:

> the set of all strings consisting of *i* copies of *a* followed by *i* copies of *b* such that *i* is is allowed to take on the value of each and every whole number value greater than or equal to 1.

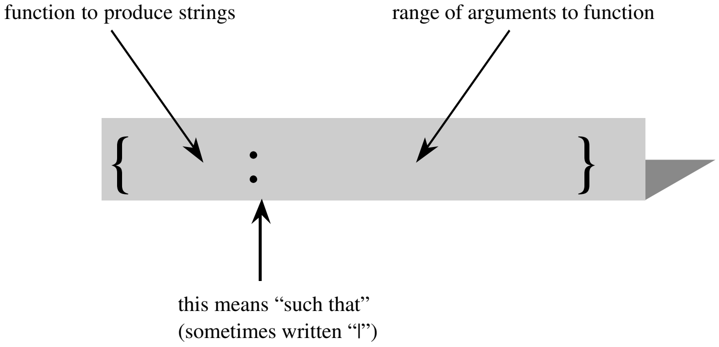


Figure 2.4. Understanding a set definition of a formal language

Table 2.3. What the set definition $\{a^ib^i: i \geqslant 1\}$ means

Notation	String function	Meaning
a	N/A	The string a
b	N/A	The string b
a^i	*Power*	The string formed by writing down i copies of the string a
b^i	*Power*	The string formed by writing down i copies of the string b
a^ib^i	*Concatenation*	The string formed by writing down i copies of a followed by i copies of b
$: i \geqslant 1$	N/A	Such that i is allowed to take on the value of each and every whole number value greater than or equal to 1 (we could have written $i > 0$)

Changing the right-hand side of the set definition can change the language defined. For example, $\{a^ib^i: i \geqslant 0\}$ defines:

> the set of all strings consisting of i copies of a followed by i copies of b such that i is is allowed to take on the value of each and every whole number value greater than or equal to 0.

This latter set is our original set, along with the empty string (since $a^0 = \varepsilon$, $b^0 = \varepsilon$, and therefore $a^0b^0 = \varepsilon\varepsilon = \varepsilon$). In set parlance, $\{a^ib^i: i \geqslant 0\}$ is the *union* of the set $\{a^ib^i: i \geqslant 1\}$ with the set $\{\varepsilon\}$, which can be written as:

$$\{a^ib^i: i \geqslant 0\} = \{a^ib^i: i \geqslant 1\} \cup \{\varepsilon\}.$$

The immediately preceding example illustrates a further useful feature of sets. We can often simplify the definition of a language by creating several sets and using the union, intersection and set difference operators to combine them into one. This sometimes removes the need for a complicated expression in the right-hand side of our set definition. For example, the definition

$$\{a^ib^jc^k: i \geqslant 1,\ j \geqslant 0,\ k \geqslant 0,\quad \mathit{if}\ i \geqslant 3\ \ \mathit{then}\ j = 0\ \ \mathit{else}\ k = 0\},$$

is probably better represented as

$$\{a^ic^j: i \geqslant 3,\ j \geqslant 0\} \cup \{a^ib^j: 1 \leqslant i < 3,\ j \geqslant 0\},$$

which means

> the set of strings consisting of three or more *a*s followed by zero or more *c*s, or consisting of one or two *a*s followed by zero or more *b*s.

2.5.2 Decision Programs for Languages

We have seen how to define a language by using a formal set definition. Another way of describing a language is to provide a program that tells us whether or not any given string of symbols is one of its sentences. Such a program is called a *decision program.* If the program always tells us, for any string, whether or not the string is a sentence, then the program in an implicit sense defines the language, in that the language is the set containing each and every string that the program would tell us is a sentence. That is why we use a special term, "sentence", to describe *a string that belongs to a language.* A *string* input to the program may or may not be a *sentence* of the language; the program should tell us. For an alphabet A, a

Table 2.4. A decision program for a formal language

```
1:  read(sym)          {assume read just gives us the next symbol in the string being examined}
      case sym of
         eos:  goto N  {assume read returns special symbol "eos" if at end of string}
         "a":  goto 2
         "b":  goto N
         "c":  goto N
      endcase          {case statement selects between alternatives as in Pascal}
2:  read(sym)
      case sym of
         eos:  goto Y  {if we get here we have a string of one a which is ok}
         "a":  goto 3
         "b":  goto 6  {we can have a b after one a}
         "c":  goto N  {any cs must follow three or more as – here we've only had one}
      endcase
3:  read(sym)
      case sym of
         eos:  goto Y  {if we get here we've read a string of two as which is OK}
         "a":  goto 4
         "b":  goto 6  {we can have a b after two as}
         "c":  goto N  {any cs must follow three or more as – here we've only had two}
      endcase
4:  read(sym)
      case sym of
         eos:  goto Y  {if we get here we've read a string of three or more as which is OK}
         "a":  goto 4  {we loop here because we allow any number of as ⩾ 3}
         "b":  goto N  {b can only follow one or two as}
         "c":  goto 5  {cs are OK after three or more as}
      endcase
5:  read(sym)
    case sym of
         eos:  goto Y  {if we get here we've read ⩾3 as followed by ⩾1 cs which is OK}
         "a":  goto N  {as after cs are not allowed}
         "b":  goto N  {bs are only allowed after one or two as}
         "c":  goto 5  {we loop here because we allow any number of cs after ⩾3 as}
      endcase
6:  read(sym)
      case sym of
         eos:  goto Y  {we get here if we've read 1 or 2 as followed by ⩾1 bs – OK}
         "a":  goto N  {no as allowed after bs}
         "b":  goto 6  {we loop here because we allow any number of bs after 1 or 2 as}
         "c":  goto N  {no cs are allowed after bs}
      endcase
Y:  write("yes")
      goto E
N:  write("no")
      goto E
E:  {end of program}
```

language is any *subset* of A^*. For any interesting language, then, there will be many strings in A^* that are not sentences.

Later in this book we will be more precise about the form these decision programs take, and what can actually be achieved with them. For now, however, we will consider an example to show the basic idea.

If you have done any programming at all, you will have used a decision program on numerous occasions. The decision program you have used is a component of the *compiler*. If you write programs in a language such as Pascal, you submit your program text to a compiler, and the compiler tells you if the text is a syntactically correct Pascal program. Of course, the compiler does a lot more than this, but a very important part of its job is to tell us if the source text (string) is a syntactically correct Pascal program, i.e. a *sentence* of the *language* called "Pascal".

Consider again the language

$$\{a^i c^j : i \geqslant 3,\ j \geqslant 0\} \cup \{a^i b^j : 1 \leqslant i < 3,\ j \geqslant 0\},$$

i.e.

> the set of strings consisting of three or more *a*s followed by zero or more *c*s, or consisting of one or two *a*s followed by zero or more *b*s.

Table 2.4 shows a decision program for the language.

The program of Table 2.4 is purely for illustration. In the next chapter we consider formal languages for which the above type of decision program can be created automatically. For now, examine the program to convince yourself that it correctly meets its specification, which can be stated as follows:

> given any string in $\{a, b, c\}^*$, tell us whether or not that string is a sentence of the language

$$\{a^i c^j : i \geqslant 3,\ j \geqslant 0\} \cup \{a^i b^j : 1 \leqslant i < 3,\ j \geqslant 0\}.$$

2.5.3 Rules for Generating Languages

We have seen how to describe formal languages by providing set definitions, and we have encountered the notion of a decision program for a language. The third method, which is the basis for the remainder of this chapter, defines a language by providing a set of rules to generate sentences of a language. We require that such rules are able to generate every one of the sentences of a language, and no others. Analogously, a set definition describes every one of the sentences, and no others, and a decision program says "yes" to every one of the sentences, and to no others.

There are several ways of specifying rules to generate sentences of a language. One popular form is the *syntax diagram*. Such diagrams are often used to show the structure of programming languages, and thus inform you how to write syntactically correct programs (*syntax* is considered in more detail in Chapter 3).

Figure 2.5 shows a syntax diagram for the top level syntax of the Pascal "program" construct.

The diagram in Figure 2.5 tells us that the syntactic element called a "program" consists of

> the string "PROGRAM" (entities in rounded boxes and circles represent actual strings that are required at a given point),

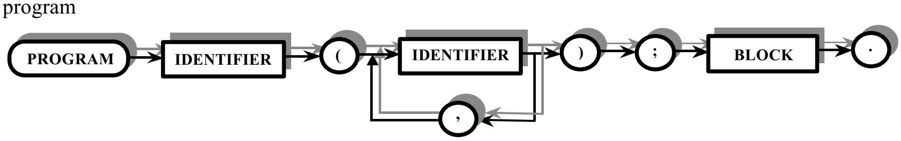


Figure 2.5. Syntax diagram for the Pascal construct "program"

followed by something called

an “identifier” (entities in rectangles are those which need elaborating in some way that is specified in a further definition),

followed by

an open bracket “(”,

followed by

a list of one or more “identifiers”, in which *every one except the last* is followed by a comma, “,”, followed by a semicolon, “;”,

followed by

a close bracket, “)”,

followed by

something called a “block”,

followed by

a full stop, “.”.

In Figure 2.6 we see the syntax diagram for the entity “identifier”.

Figure 2.6 shows us that an “identifier” consists of a letter followed by zero or more letters and/or digits.

The following fragment of Pascal:

```
program calc(input, output, infile26, outfile23);
```

associates with the syntax diagram for “program” as shown in Figure 2.7.

Of course, the diagrams in Figures 2.5 and 2.6, together with all of the other diagrams defining the syntax of Pascal, cannot tell us how to write a program to solve a given problem. That is a *semantic* consideration, relating to the *meaning* of the program text, not only its *form*. The diagrams merely describe the *syntactic structure* of constructs belonging to the Pascal language.

An alternative method of specifying the syntax of a programming language is to use a notation called Backus–Naur form (BNF).[1] Table 2.5 presents a BNF version of our syntax diagrams from above.

The meaning of the notation in Table 2.5 should be reasonably clear when you see its correspondence with syntax diagrams, as shown in Figure 2.8.

Formalisms such as syntax diagrams and BNF are excellent ways of defining the syntax of a language. If you were taught to use a programming language, you may never have looked at a formal definition of its syntax. Analogously, you probably did not learn your own “natural” language by studying a book describing its grammar. However, many programming languages are similar to each other in

Table 2.5. BNF version of Figures 2.5 and 2.6

<program> ::= <program heading> <block>
<program heading> := program <identifier> (<identifier> { , <identifier>})
<identifier> ::= <letter> {<letter or digit>}
<letter or digit> ::= <letter> | <digit>

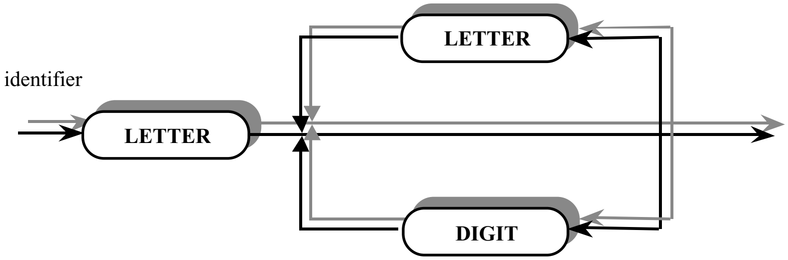
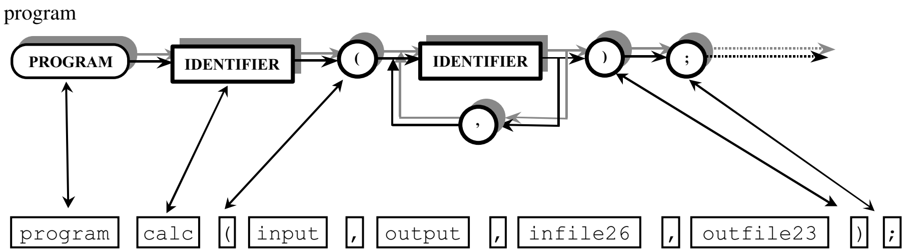


Figure 2.6. Syntax diagram for a Pascal "identifier"



```
program calc ( input , output , infile26 , outfile23 ) ;
```

Figure 2.7. How a syntax diagram describes a Pascal statement

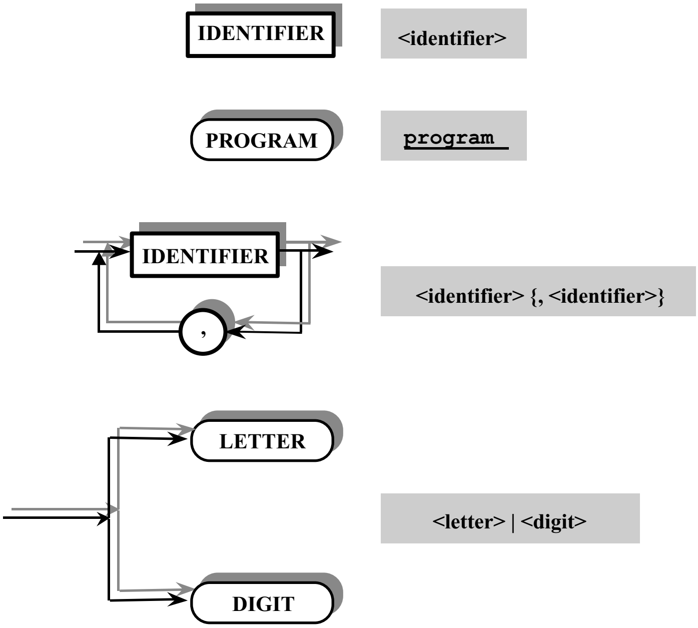


Figure 2.8. How syntax diagrams and BNF correspond

many respects, and learning a subsequent programming language is made easier if the syntax is clearly defined. Syntax descriptions can also be useful for refreshing your memory about the syntax of a programming language with which you are familiar, particularly for types of statements you rarely use.

If you want to see how concisely a whole programming language can be described in BNF, see the original definition of the Pascal language,[2] from where the above Pascal syntax diagrams and BNF descriptions were obtained. The BNF definitions for the whole Pascal language are presented in only *five* pages.

2.6 Formal Grammars

A *grammar* is a set of rules for generating strings. The grammars we will use in the remainder of this book are known as *phrase structure grammars* (PSGs). Here, our formal definitions will be illustrated by reference to the following grammar:

$$S \rightarrow aS \mid bB$$
$$B \rightarrow bB \mid bC \mid cC$$
$$C \rightarrow cC \mid c.$$

In order to use our grammar, we need to know something about the status of the symbols that we have used. Table 2.6 provides an informal description of the symbols that appear in grammars such as the one above.

2.6.1 Grammars, Derivations, and Languages

Table 2.7 presents an informal description, supported by examples using our grammar above, of how we use a grammar to generate a sentence.

Table 2.6. The symbols that make up the PSG:

$$S \rightarrow aS \mid bB$$
$$B \rightarrow bB \mid bC \mid cC$$
$$C \rightarrow cC \mid c$$

Symbols	Name and meaning
S, B, C	*Non-terminal* symbols [BNF: things in angled brackets, for example <identifier>]
S	Special non-terminal, called a *start*, or *sentence*, symbol [BNF: in our example above, <program>]
a, b, c	*Terminal* symbols: only these symbols can appear in sentences [BNF: the underlined terms (for example program) and punctuation symbols (for example ";")]
$\rightarrow$	*Production arrow* [BNF: the symbol "::="]
$S \rightarrow aS$	*Production rule*, usually called simply a *production* (or sometimes we will just use the word *rule*). Means "*S produces aS*", or "*S can be replaced by aS*". The string to the left of → is called the *left-hand side* of the production, the string to the right of → is called the *right-hand side* [BNF: this rule would be written as <S> ::= a <S>]
|	"Or", so $B \rightarrow bB \mid bC \mid cC$ means "*B produces bB* or *bC* or *cC*". Note that this means that $B \rightarrow bB \mid bC \mid cC$ is really *three* production rules, i.e. $B \rightarrow bB$, $B \rightarrow bC$, and $B \rightarrow cC$. So there are *seven* production rules altogether in the example grammar above [BNF: exactly the same]

Table 2.7. Using a PSG

Action taken	Resulting string	Production applied
Start with S, the *start* symbol	S	
If a substring of the resulting string matches the left-hand side of one or more productions, replace that substring by the right-hand side of any one of those productions	aS	$S \rightarrow aS$
Same as above	aaS	$S \rightarrow aS$
Same as above	$aabB$	$S \rightarrow bB$
Same as above	$aabcC$	$B \rightarrow cC$
Same as above	$aabccC$	$C \rightarrow cC$
Same as above	$aabccc$	$C \rightarrow c$
If the result string consists entirely of terminals, then stop		

As you can see from Table 2.7, there is often a choice as to which rule to apply at a given stage. For example, when the resulting string was aaS, we could have applied the rule $S \rightarrow aS$ as many times as we wished (adding another a each time). A similar observation can be made for the applicability of the $C \rightarrow cC$ rule when the resulting string was $aabcC$, for example.

Here are some other strings we could create, by applying the rules in various ways:

$$abcc, \quad bbbbc, \quad \text{and} \quad a^3b^2c^5.$$

You may like to see if you can apply the rules yourself to create the above strings. You must always begin with a rule that has S on its left-hand side (that is why S is called the *start* symbol).

We write down the S symbol to start the process, and we merely repeat the process described in Table 2.7 as

> *if a substring of the resulting string matches the left-hand side of one or more productions, replace that substring by the right-hand side of any one of those productions,*

until the following becomes true:

> *if the result string consists entirely of terminals,*

at which point we:

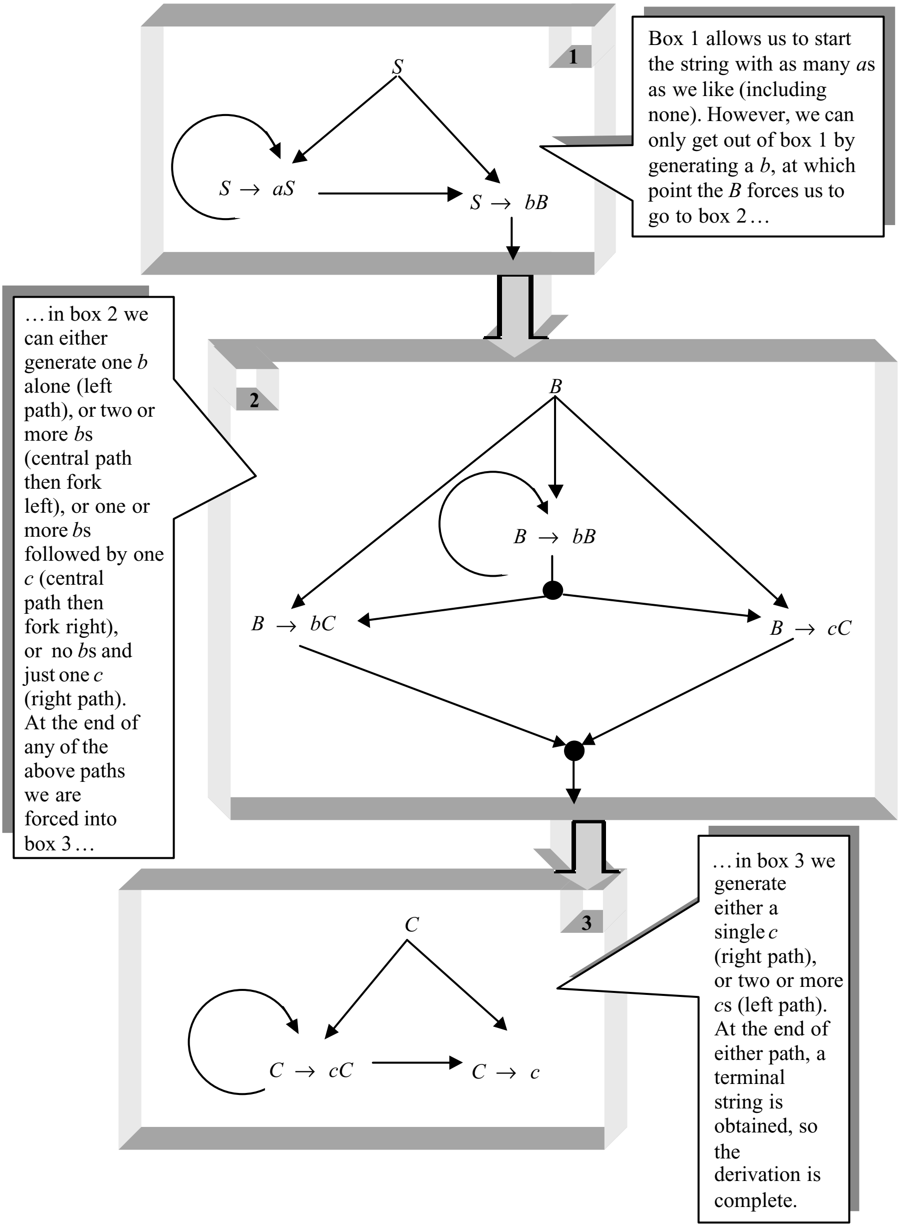
> *stop.*

You may wonder why the process of matching the substring was not presented as:

> *if a* non-terminal symbol *in the resulting string matches the left-hand side of one or more productions, replace that* non-terminal symbol *by the right-hand side of any one of those productions.*

This would clearly work for the example grammar given. However, as discussed in the next section, grammars are not necessarily restricted to having single non-terminals on the left-hand sides of their productions.

The process of creating strings using a grammar is called *deriving* them, so when we show how we have used the grammar to *derive* a string (as was done in Table 2.7), we are showing a *derivation* for (or *of*) that string.



Box 1 allows us to start the string with as many *a*s as we like (including none). However, we can only get out of box 1 by generating a *b*, at which point the *B* forces us to go to box 2…

…in box 2 we can either generate one *b* alone (left path), or two or more *b*s (central path then fork left), or one or more *b*s followed by one *c* (central path then fork right), or no *b*s and just one *c* (right path). At the end of any of the above paths we are forced into box 3…

…in box 3 we generate either a single *c* (right path), or two or more *c*s (left path). At the end of either path, a terminal string is obtained, so the derivation is complete.

Figure 2.9. Working out all of the terminal strings that a grammar can generate

Let us now consider all of the "terminal strings" – strings consisting entirely of terminal symbols, also known as *sentences* – we can use the example grammar to derive. As this is a simple grammar, it is not too difficult to work out what they are. Figure 2.9 shows the choice of rules possible for deriving terminal strings from the example grammar.

Table 2.8. The language generated by a grammar

Box in diagram of Figure 2.9	Informal description of derived strings	Formal description of derived strings
$\boxed{1}$ i.e. productions $S \rightarrow aS \mid S \rightarrow bB$	"Any non-zero number of *a*s followed by *bB*" or "just *bB*" which is the same as saying "zero or more *a*s followed by *bB*"	$a^ibB,\ i \geqslant 0$
... is expanded in box 2 ...		... the *B* at the end ...
$\boxed{2}$ i.e. *productions* $B \rightarrow bB \mid bC \mid cC$	"Any non-zero number of *b*s followed by either *bC* or *cC*" or "just *bC*" or "just *cC*"	$b^jC,\ j \geqslant 1$ or $b^jcC,\ j \geqslant 0$
... is expanded in box 3 ...		... the *C* at the end ...
$\boxed{3}$ i.e. productions $C \rightarrow cC \mid c$	"Any non-zero number of *c*s followed by one *c*" or "just one *c*" which is the same as saying "one or more *c*s"	$c^k,\ k \geqslant 1$

Any "legal" application of our production rules, starting with *S*, the start symbol, alone, and resulting in a *terminal string*, would involve us in following a path through the diagram in Figure 2.9, starting in box 1, passing through box 2, and ending up in box 3. The boxes in Figure 2.9 are annotated with the strings produced by taking given options in applying the rules. Table 2.8 summarises the strings described in Figure 2.9.

We now define a set that contains all of the terminal strings (and only those strings) that can be derived from the example grammar. The set will contain all strings defined as follows:

> A string taken from the set $\{a^ib: i \geqslant 0\}$ concatenated with a string taken from the set $\{b^j: j \geqslant 1\} \cup \{b^jc: j \geqslant 0\}$ concatenated with a string taken from the set $\{c^k: k \geqslant 1\}$.

The above can be written as:

$$\{a^ibb^jc^k: i \geqslant 0,\ j \geqslant 1,\ k \geqslant 1\} \cup \{a^ibb^jcc^k: i \geqslant 0,\ j \geqslant 0,\ k \geqslant 1\}.$$

Observe that bb^j, $j \geqslant 1$ is the same as b^j, $j \geqslant 2$, and bb^j, $j \geqslant 0$ is the same as b^j, $j \geqslant 1$, and cc^k, $k \geqslant 1$ is the same as c^k, $k \geqslant 2$ so we could write:

$$\{a^ib^jc^k: i \geqslant 0,\ j \geqslant 2,\ k \geqslant 1\} \cup \{a^ib^jc^k: i \geqslant 0,\ j \geqslant 1,\ k \geqslant 2\}.$$

This looks rather complicated, but essentially there is only one awkward case, which is that if there is only one *b* then there must be two or more *c*s (any more than one *b* and we can have 1 or more *c*s). So we could have written:

$$\{a^ib^jc^k: i \geqslant 0,\ j \geqslant 1,\ k \geqslant 1,\ \mathit{if}\, j = 1\ \ \mathit{then}\, k \geqslant 2\ \ \mathit{else}\, k \geqslant 1\}.$$

Whichever way we write the set, one point should be made clear: the set is a set of strings formed from symbols in the alphabet $\{a, b, c\}$, that is to say, the set is a *formal language*.

2.6.2 The Relationship Between Grammars and Languages

We are now ready to give an intuitive definition of the relationship between grammars and languages:

> The *language generated by a grammar* is the set of all *terminal strings* that can be *derived* using the *productions* of that grammar, each derivation beginning with the *start symbol* of that grammar.

Our example grammar, when written like this:

$$\begin{aligned} S &\to aS \mid bB \\ B &\to bB \mid bC \mid cC \\ C &\to cC \mid c \end{aligned}$$

is not fully defined. A grammar is fully defined when we know which symbols are terminals, which are non-terminals, and which of the non-terminals is the start symbol. In this book, we will usually see only the productions of a grammar, and we will assume the following:

- capitalised letters are *non-terminal symbols*
- non-capitalised letters are *terminal symbols*
- the capital letter S is the *start symbol.*

The above will always be the case unless explicitly stated otherwise.

2.7 Phrase Structure Grammars and the Chomsky Hierarchy

The production rules of the example grammar from the preceding section are simple in format. For example, the left-hand sides of all the productions consist of lone non-terminals. As we see later in the book, restricting the form of productions allowed in a grammar in certain ways simplifies certain language processing tasks, but it also reduces the sophistication of the languages that such grammars can generate. For now, we will define a scheme for classifying grammars according to the "shape" of their productions which will form the basis of our subsequent discussion of grammars and languages. The classification scheme is called the Chomsky hierarchy, named after Noam Chomsky, an influential American linguist.

2.7.1 Formal Definition of PSGs

To prepare for specifying the Chomsky hierarchy, we first need to precisely define the term PSG. Table 2.9 does this.

Formally, then, a PSG, G, is specified as (N, T, P, S). This is what mathematicians call a "tuple" (of four elements).

The definition in Table 2.9 makes it clear that the empty string, ε, cannot appear alone on the left-hand side of any of the productions of a PSG. Moreover, the definition tells us that ε *is* allowed on the right-hand side. Otherwise, any strings of terminals and/or non-terminals can appear on either side of productions. However, in most grammars we usually find that there are one or more non-terminals on the *left-hand side* of each production.

Table 2.9. The formal definition of a PSG

Any PSG, G, consists of the following:	
N, a set of *non-terminal* symbols	An alphabet, containing no symbols that can appear in sentences
T, a set of *terminal* symbols	Also an alphabet, containing only symbols that *can* appear in sentences
P, a set of *production rule*s of the form	This specification uses the notation for specifying strings from an alphabet we looked at earlier
$x \rightarrow y$, where $x \in (N \cup T)^+$, and $y \in (N \cup T)^*$	x is the left-hand side of a production, y the right-hand side
	The definition of y means: *the right-hand side of each production is a possibly empty string of terminals and/or non-terminals*
	The only difference between the specification above and the one for x (the left-hand side) is that the one for x uses "+" rather than "*"
	So the specification for x means: *the left-hand side of each production is a non-empty string of terminals and/or non-terminals*
S, a member of N, designated as the *start*, or *sentence* symbol	The non-terminal symbol with which we always begin a derivation

As we always start a derivation with a lone S (the *start* symbol), for a grammar to derive anything it must have at least one production with S alone on its left-hand side. This last piece of information is not specified in the definition above, as there is nothing in the formal definition of PSGs that says they *must* generate anything. To refer back to our earlier example grammar, its full formal description would be as shown in Table 2.10.

Table 2.10. The (N, T, P, S) form of a grammar

Productions	(N, T, P, S)	
	({S, B, C},	---N
$S \rightarrow aS \mid bB$	{a, b, c},	---T
$B \rightarrow bB \mid bC \mid cC$	{$S \rightarrow aS$, $S \rightarrow bB$, $B \rightarrow bB$, $B \rightarrow bC$,	---P
$C \rightarrow cC \mid c$	$B \rightarrow cC$, $C \rightarrow cC$, $C \rightarrow c$},	
	S	---S
	)	

2.7.2 Derivations, Sentential Forms, Sentences, and "*L*(*G*)"

We have formalised the definition of a PSG. We now formalise our notion of derivation, and introduce some useful terminology to support subsequent discussion. To do this, we consider a new grammar:

$$\begin{aligned} S &\rightarrow aB \mid bA \mid \varepsilon \\ A &\rightarrow aS \mid bAA \\ B &\rightarrow bS \mid aBB. \end{aligned}$$

Using the conventions outlined earlier, we know that S is the start symbol, $\{S, A, B\}$ is the set of non-terminals (N), and $\{a, b\}$ is the set of terminals (T). So we need not provide the full (N, T, P, S) definition of the grammar.

As in our earlier example, the left-hand sides of the above productions all consist of single non-terminals. We see an example grammar that differs from this later in the chapter.

Here is a string in $(N \cup T)^+$ that the above productions can be used to derive, as you might like to verify for yourself:

$$abbbaSA.$$

This is not a terminal string, since it contains non-terminals (S and A). Therefore it is not a sentence. The next step could be, say, to apply the production $A \rightarrow bAA$, which would give us

$$abbbaSbAA,$$

which is also not a sentence.

We now have two strings, *abbbaSA* and *abbbaSbAA* that are such that the *former can be used as a basis for the derivation of the latter by the application of one production rule of the grammar*. This is rather a mouthful, even if we replace *"by the application of one production rule of the grammar"* by the phrase *"in one step"*, so we introduce a symbol to represent this relationship. We write:

$$abbbaSA \Rightarrow abbbaSbAA.$$

To be absolutely correct, we should give our grammar a name, say G, and write

$$abbbaSA \Rightarrow^G abbbaSbAA$$

to denote which particular grammar is being used. Since it is usually clear in our examples which grammar is being used, we will simply use $\Rightarrow$. We now use this symbol to show how our example grammar derives the string *abbbaSbAA*:

$$S \Rightarrow aB$$
$$aB \Rightarrow abS$$
$$abS \Rightarrow abbA$$
$$abbA \Rightarrow abbbAA$$
$$abbbAA \Rightarrow abbbaSA$$
$$abbbaSA \Rightarrow abbbaSbAA.$$

As it is tedious to write out each intermediate stage twice, apart from the first (S) and the last (*abbbaSbAA*), we allow an abbreviated form of such a derivation as follows:

$$S \Rightarrow aB \Rightarrow abS \Rightarrow abbA \Rightarrow abbbAA \Rightarrow abbbaSA \Rightarrow abbbaSbAA.$$

We now use our new symbol as the basis of some additional useful notation, as shown in Table 2.11.

A new term is now introduced, to simplify references to the intermediate stages in a derivation. We call these intermediate stages *sentential forms*. Formally, given any grammar, G, a *sentential form* is any string that can be derived in zero or more steps from the start symbol, S. By "any string", we mean exactly that; not only terminal strings, but any string of terminals and/or non-terminals. Thus, *a sentence is a sentential form*, but *a sentential form is not necessarily a sentence*. Given the simple grammar

$$S \rightarrow aS \mid a,$$

some sentential forms are: S, *aaaaaaS*, and a^{10}. Only one of these sentential forms (a^{10}) is a sentence, as it is the only one that consists entirely of *terminal symbols*.

Table 2.11. Useful notation for discussing derivations, and some example true statements for grammar:

$$S \to aB \mid bA \mid \varepsilon$$
$$A \to aS \mid bAA$$
$$B \to bS \mid aBB$$

Notation	Meaning	Example true statements
$x \Rightarrow y$	The application of one production rule results in the string x becoming the string y Also expressed as "*x generates y* in one step", or "*x produces y* in one step", or "*y is derived from x* in one step"	$aB \Rightarrow abS$ $S \Rightarrow \varepsilon$ $abbbaSA \Rightarrow abbbaSbAA$
$x \Rightarrow^* y$	x generates y in zero or more steps, or just "*x generates y*", or "*x produces y*", or "*y is derived from x*"	$S \Rightarrow^* S$ $S \Rightarrow^* abbbaSA$ $aB \Rightarrow^* abbbaa$
$x \Rightarrow^+ y$	x generates y in one or more steps, or just "*x generates y*", or "*x produces y*", or "*y is derived from x*"	$S \Rightarrow^+ abbbaSA$ $abbbaSbAA \Rightarrow^+ abbbabaa$

Formally, using our new notation,

if $S \Rightarrow^* x$, then x is a sentential form;
if $S \Rightarrow^* x$, and x is a terminal string, then x is a *sentence*.

We now formalise a definition given earlier, this being the statement that

> the language generated by a grammar is the set of all terminal strings that can be derived using the productions of that grammar, each derivation beginning with the start symbol of that grammar.

Using various aspects of the notation introduced in this chapter, this becomes:

given a PSG, G, $L(G) = \{x: x \in T^* \text{ and } S \Rightarrow {}^* x\}$.

(Note that the definition assumes that we have specified the set of terminals and the start symbol of the grammar, which as we said earlier is done implicitly in our examples.)

So, if G is some PSG, $L(G)$ means *the language generated by G*. As the set definition of $L(G)$ clearly states, the set $L(G)$ contains all of the terminal strings generated by G, but only the strings that G generates. It is very important to realise that this is what it means when we say *the language generated by the grammar*.

We now consider three examples, to reinforce these notions. The first is an example grammar encountered above, now labelled G_1:

$$S \to aS \mid bB$$
$$B \to bB \mid bC \mid cC$$
$$C \to cC \mid c.$$

We have already provided a set definition of $L(G_1)$; it was:

$$L(G_1) = \{a^i b^j c^k: i \geqslant 0,\ j \geqslant 1,\ k \geqslant 1,\ \textit{if } j = 1\ \textit{ then } k \geqslant 2\ \textit{ else } k \geqslant 1\}.$$

Another grammar we have already encountered, which we now call G_2, is:

$$S \rightarrow aB \mid bA \mid \varepsilon$$
$$A \rightarrow aS \mid bAA$$
$$B \rightarrow bS \mid aBB.$$

This is more complex than G_1, in the sense that some of G_2's productions have more than one non-terminal on their right-hand sides:

$$L(G_2) = \{x\colon x \in \{a, b\}^* \text{ and the number of } a\text{s in } x \text{ equals the number of } b\text{s}\}.$$

I leave it to you to establish that the above statement is true.

Note that $L(G_2)$ is not the same as a set that we came across earlier, i.e.

$$\{a^i b^i\colon i \geqslant 1\},$$

which we will call set A. In fact, set A is a proper subset of $L(G_2)$. G_2 can generate all of the strings in A, but it generates many more besides (such as ε, *bbabbbaaaaab*, and so on). A grammar, G_3, such that $L(G_3) = A$ is:

$$S \rightarrow ab \mid aSb.$$

2.7.3 The Chomsky Hierarchy

This section describes a classification scheme for PSGs, and the corresponding phrase structure languages (PSLs) that they generate, which is of the utmost importance in determining certain of their computational features. PSGs can be classified in a hierarchy, the location of a PSG in that hierarchy being an indicator of certain characteristics required by a decision program for the corresponding language. We saw above how one example language could be processed by an extremely simple decision program. Much of this book is devoted to investigating the computational nature of formal languages. We use as the basis of our investigation the classification scheme for PSGs and PSLs called the *Chomsky hierarchy*.

Classifying a grammar according to the Chomsky hierarchy is based solely on the presence of certain patterns in the productions. Table 2.12 shows how to make the classification. The types of grammar in the Chomsky hierarchy are named types 0 to 3, with 0 as the most general type. Each type from 1 to 3 is defined according to one or more restrictions on the definition of the type numerically preceding it, which is why the scheme qualifies as a *hierarchy*.

If you are observant, you may have noticed an anomaly in Table 2.12. Context sensitive grammars are not allowed to have the empty string on the right-hand side of productions, whereas all of the other types are. This means that, for example, our grammar G_2, which can be classified as unrestricted and as context free (but not as regular), cannot be classified as context sensitive. However, every grammar that can be classified as regular can be classified as context free, and every grammar that can be classified as context free can be classified as unrestricted.

When classifying a grammar according to the Chomsky hierarchy, you should remember the following:

> For a grammar to be classified as being of a certain type, *each and every production of that grammar must match the pattern specified for productions of that type.*

Table 2.12. The Chomsky hierarchy

Type no.	Type name	Patterns to which ALL productions must conform	Informal description and examples
0	Unrestricted	$x \rightarrow y,\ x \in (N \cup T)^+,$ $y \in (N \cup T)^*$	The definition of PSGs we have already seen. Anything allowed on the left-hand side (except for ε), anything allowed on the right. All of our example grammars considered so far conform to this. Example type 0 production: $aXYpq \rightarrow aZpq$ (all productions of G_1, G_2 *and* G_3 conform – but see below).
1	Context sensitive	$x \rightarrow y,\ x \in (N \cup T)^+,$ $y \in (N \cup T)^+,\ \|x\| \leqslant \|y\|$	As for *type* 0, but we are not allowed to have ε on the left *or the right-hand-sides*. Note that the example production given for *type* 0 is not a context sensitive production, as the length of the right-hand side is less than the length of the left. Example type 1 production: $aXYpq \rightarrow aZwpq$ (all productions of G_1 and G_3 conform, but not all of those of G_2 do).
2	Context free	$\mathrm{x} \rightarrow \mathrm{y},\ x \in N,$ $\mathrm{y} \in (N \cup T)^*$	Single non-terminal on left, any mixture of terminals and/or non-terminals on the right. Also, ε is allowed on the right. Example type 2 production: $X \rightarrow XapZQ$ (all productions of G_1, G_2, and G_3 conform).
3	Regular	$w \rightarrow x$, or $w \rightarrow yz$ $w \in N,\ x \in T \cup \{\varepsilon\},$ $y \in T,\ z \in N$	Single non-terminal on left, and either ε or a single terminal or a single terminal followed by a single non-terminal, on the right. Example type 3 productions: $P \rightarrow pQ,$ $F \rightarrow a$ (all of the productions of G_1 conform to this, but G_2 and G_3 do not).

Which means that the following grammar:

$$S \rightarrow aS \mid aA \mid AA$$
$$A \rightarrow aA \mid a$$

is classified as *context free*, since the production $S \rightarrow AA$ does not conform to the pattern for regular productions, even though all of the other productions do.

Table 2.13. The order in which to attempt the classification of a grammar, *G*, in the Chomsky hierarchy

```
if G is regular then
   return("regular")
else
   if G is context free then
      return("context free")
   else
      if G is context sensitive then
         return("context sensitive")
      else
         return("unrestricted")
      endif
   endif
endif
```

So, given the above rule that all productions must conform to the pattern, you classify a grammar, *G*, according to the procedure in Table 2.13.

Table 2.13 tells us to begin by attempting to classify *G* according to the most restricted type in the hierarchy. This means that, as indicated by Table 2.12, G_1 is a *regular* grammar, and G_2 and G_3 are *context free* grammars (CFGs). Of course, we know that as *all* regular grammars are CFGs, G_1 is also context free. Similarly, we know that they can *all* be classified as unrestricted. But we make the classification as specific as possible.

From the above, it can be seen that classifying a PSG is done simply by seeing if its productions match a given pattern. As we already know, grammars generate languages. In terms of the Chomsky hierarchy, *a language is of a given type if it is generated by a grammar of that type*. So, for example,

$$\{a^ib^i: i \geqslant 1\} \quad \text{(set } A \text{ mentioned above)}$$

is a context free language (CFL), since it is generated by G_3, which is classified as a CFG. However, how can we be sure that there is not a *regular grammar* that could generate *A*? We see later on that the more restricted the language (in the Chomsky hierarchy), the simpler the decision program for the language. It is therefore useful to be able to define the simplest possible type of grammar for a given language. In the meantime, you might like to see if you can create a *regular* grammar to generate set *A* (clue: do not devote too much time to this!).

From a theoretical perspective, the immediately preceding discussion is very important. If we can establish that there are languages that can be generated by grammars at some level of the hierarchy and cannot be generated by more restricted grammars, then we are sure that we do indeed have a genuine *hierarchy*. However, there are also *practical* issues at stake, for as mentioned above, and discussed in more detail in Chapters 4, 5 and 7, each type of grammar has associated with it a type of decision program, in the form of an abstract machine. The more restricted a language is, the simpler the type of decision program we need to write for that language.

In terms of the Chomsky hierarchy, our main interest is in CFLs, as it turns out that the syntactic structure of most programming languages is represented by CFGs. The grammars and languages we have looked at so far in this book have all been context free (remember that any *regular* grammar or language is, by definition, also context free).

2.8 A Type 0 Grammar: Computation as Symbol Manipulation

We close this chapter by considering a grammar that is more complex than our previous examples. The grammar, which we label G_4, has productions as follows (each row of productions has been numbered, to help us to refer to them later):

$$S \rightarrow AS \mid AB \tag{2.1}$$
$$B \rightarrow BB \mid C \tag{2.2}$$
$$AB \rightarrow HXNB \tag{2.3}$$
$$NB \rightarrow BN \tag{2.4}$$
$$BM \rightarrow MB \tag{2.5}$$
$$NC \rightarrow Mc \tag{2.6}$$
$$Nc \rightarrow Mcc \tag{2.7}$$
$$XMBB \rightarrow BXNB \tag{2.8}$$
$$XBMc \rightarrow Bc \tag{2.9}$$
$$AH \rightarrow HA \tag{2.10}$$
$$H \rightarrow a \tag{2.11}$$
$$B \rightarrow b \tag{2.12}$$

G_4 is a type 0, or unrestricted grammar. It would be context sensitive, but for the production $XBMc \rightarrow Bc$, which is the only production with a right-hand side shorter than its left-hand side.

Table 2.14 represents the derivation of a particular sentence using this grammar. It is presented step by step. Each sentential form, apart from the *sentence* itself, is followed by the number of the row in G_4 from which the production used to achieve the next step was taken. Table 2.14 should be read row by row, left to right.

The sentence derived is $a^2b^3c^6$. Notice how, in Table 2.14, the grammar replaces each A in the sentential form $AABBBC$ by H, and each time it does this it places one

Table 2.14. A type 0 grammar is used to derive a sentence

Stage	Row	Stage	Row	Stage	Row
S	(1)	$A\underline{S}$	(1)	$AA\underline{B}$	(2)
$AAB\underline{B}$	(2)	$AABB\underline{B}$	(2)	$A\underline{AB}BBC$	(3)
$AHX\underline{NB}BBC$	(4)	$AHXB\underline{NB}BC$	(4)	$AHXBB\underline{NB}C$	(4)
$AHXBBB\underline{NC}$	(6)	$AHXBB\underline{BM}c$	(5)	$AHXB\underline{BM}Bc$	(5)
$AHX\underline{BM}BBc$	(5)	$AH\underline{XMBB}Bc$	(8)	$AHBX\underline{NB}Bc$	(4)
$AHBXB\underline{NB}c$	(4)	$AHBXBB\underline{Nc}$	(7)	$AHBXB\underline{BM}cc$	(5)
$AHBX\underline{BM}Bcc$	(5)	$AHB\underline{XMBB}cc$	(8)	$AHBBX\underline{NB}cc$	(4)
$AHBBXB\underline{Nc}c$	(7)	$AHBB\underline{XBMc}cc$	(9)	$\underline{AH}BBBccc$	(10)
$H\underline{AB}BBccc$	(3)	$HHX\underline{NB}BBccc$	(4)	$HHXB\underline{NB}Bccc$	(4)
$HHXBB\underline{NB}ccc$	(4)	$HHXBBB\underline{Nc}cc$	(7)	$HHXBB\underline{BM}cccc$	(5)
$HHXB\underline{BM}Bcccc$	(5)	$HHX\underline{BM}BBcccc$	(5)	$HH\underline{XMBB}Bcccc$	(8)
$HHBX\underline{NB}Bcccc$	(4)	$HHBXB\underline{NB}cccc$	(4)	$HHBXBB\underline{Nc}ccc$	(7)
$HHBXB\underline{BM}ccccc$	(5)	$HHBX\underline{BM}Bccccc$	(5)	$HHB\underline{XMBB}ccccc$	(8)
$HHBBX\underline{NB}ccccc$	(4)	$HHBBXB\underline{Nc}cccc$	(7)	$HHBB\underline{XBMc}ccccc$	(9)
$\underline{H}HBBBcccccc$	(11)	$a\underline{H}BBBcccccc$	(11)	$aa\underline{B}BBcccccc$	(12)
$aab\underline{B}Bcccccc$	(12)	$aabb\underline{B}cccccc$	(12)	$aabbbcccccc$	

c at the rightmost end for each B. Note also how the grammar uses non-terminals as "markers" of various types:

- H is used to replace the As that have been accounted for
- X is used to indicate how far along the Bs we have reached
- N is used to move right along the Bs, each time ending in a c being added to the end of the sentential form
- M is used to move left back along the Bs.

You may also notice that at many points in the derivation several productions are applicable. However, many of these productions lead eventually to "dead ends", i.e. sentential forms that cannot lead eventually to sentences.

The language generated by G_4, i.e. $L(G_4)$, is $\{a^i b^j c^{i \times j}: i,\ j \geqslant 1\}$. This is the set:

> all strings of the form one or more as followed by one or more bs followed by cs in which the number of cs is the number of as *multiplied* by the number of bs.

You may wish to convince yourself that this is the case.

G_4 is rather a complicated grammar compared to our earlier examples. You may be wondering if there is a simpler *type* of grammar, perhaps a CFG, that can do the same job. In fact there is not. However, while the grammar is comparatively complex, the method it embodies in the generation of the sentences is quite simple. Essentially, like all grammars, it simply replaces one string by another at each stage in the derivation.

An interesting way of thinking about G_4 is in terms of it performing a kind of *computation*. Once a sentential form like $A^i B^j C$ is reached, the productions then ensure that $i \times j$ cs are appended to the end by essentially modelling the simple algorithm in Table 2.15.

The question that arises is: what range of computational tasks can we carry out using such purely syntactic transformations? We see from our example that the type 0 grammar simply specifies string substitutions. If we take our strings of as and bs as representing numbers, so that, say, a^6 represents the *number* 6, we see that G_4 is essentially a model of a process for multiplying together two arbitrary length numbers.

Later in this book, we encounter an abstract machine, called a *Turing machine* (TM), that specifies string operations, each operation involving the replacing of only one symbol by another, and we see that the machine is actually as powerful as the type 0 grammars. Indeed, the machine is capable of performing a wider range of computational tasks than even the most powerful real computer.

However, we will not concern ourselves with these issues until later. In the next chapter, we encounter more of the fundamental concepts of formal languages: *syntax*, *semantics*, and *ambiguity*.

Table 2.15. The "multiplication" algorithm embodied in grammar G_4

```
for each $A$ do
  for each $B$ do
    put a $c$ at the end of the sentential form
  endfor
endfor
```

2.9 Exercises

For exercises marked †, solutions, partial solutions, or hints to get you started appear in "Solutions to Selected Exercises" at the rear of the book.

2.1. Classify the following grammars according to the Chomsky hierarchy. In all cases, briefly justify your answer:

(a)† $S \rightarrow aA$
$A \rightarrow aS \mid aB$
$B \rightarrow bC$
$C \rightarrow bD$
$D \rightarrow b \mid bB.$

(b)† $S \rightarrow aS \mid aAbb$
$A \rightarrow \varepsilon \mid aAbb.$

(c) $S \rightarrow XYZ \mid aB$
$B \rightarrow PQ \mid S$
$Z \rightarrow aS.$

(d) $S \rightarrow \varepsilon.$

2.2†. Construct set definitions of each of the languages generated by the four grammars in Exercise 2.1.

Hint: the language generated by 2.1(c) *is not the same as that generated by* 2.1(d)*, as one of them contains no strings at all, whereas the other contains exactly one string.*

2.3†. It was pointed out above that we usually insist that one or more non-terminals must be included in the left-hand side of type 0 productions. Write down a formal expression representing this constraint. Assume that N is the set of non-terminals, and T the set of terminals.

2.4. Construct regular grammars, G_v, G_w, and G_x, such that

(a) $L(G_v) = \{c^j : j > 0$, and j does not divide exactly by 3$\}$.

(b) $L(G_w) = \{a^i b^j [cd]^k : i, k \geqslant 0,\ 0 \leqslant j \leqslant 1\}$

Note: as we are dealing only with whole numbers, the expression $0 \leqslant j \leqslant 1$*, which is short for* $0 \leqslant j$ *and* $j \leqslant 1$*, is the same as writing:* $j = 0$ *or* $j = 1$.

(c) $L(G_x) = \{a, b, c\}^*$.

2.5†. Use your answer to 2.4(c) as the basis for sketching out an intuitive justification that A^* is a regular language, for any alphabet, A.

2.6. Use the symbol $\Rightarrow$ in showing the step by step derivation of the string c^5 using (a) G_v and (b) G_x from Exercise 2.4.

2.7. Construct CFGs, G_y and G_z, such that

(a) $L(G_y) = \{a^{2i+1}c^j\mathrm{b}^{2i+1}: i \geqslant 0,\ 0 \leqslant j \leqslant 1\}$

Note: if $i \geqslant 0$, a^{2i+1} means "all odd numbers of as".

(b)† $L(G_z) =$ all Boolean expressions in your favourite programming language (Boolean expressions are discussed in Chapter 13).

2.8. Use the symbol ⇒ in showing the step-by-step derivation of a^3b^3 using

(a) G_y from Exercise 2.7,

and the grammar

(b) G_3 from Chapter 2, i.e. $S \rightarrow ab \mid aSb$

2.9. Provide a regular grammar to generate the language {*ab*, *abc*, *cd*}.

Hint: make sure your grammar generates only the three given strings, and no others.

2.10†. Use your answer to Exercise 2.9 as the basis for sketching out an intuitive justification that any finite language is regular.

Note: the converse, i.e. that every regular language is finite, is certainly not true. Consider, for example, the languages specified in Exercise 2.4.

Notes

[1] The formalism we describe here is actually *Extended* BNF (EBNF). The original BNF did not include the repetition construct found in Table 2.5.

[2] Jensen and Wirth (1975), see Further Reading section.