FORMAL LANGUAGES AND AUTOMATA

CM FL&A

Abstract: This manuscript gives detailed discussion of the lecture course on Formal Languages and Automata.

Keywords: Automata, Grammars.

Communication regarding this document should be directed to:

Antonia J. Jones

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WALES, CARDIFF
PO BOX 916
Cardiff CF2 3XF

Telephone: +44-122-287-4812
Telefax: +44-122-287-4598

Copyright: © University of Wales, Cardiff 1999.
Recommended texts:


D. Hopkins and B. Moss. Remarks: Not much practical discussion, covers almost all the course, goes further on theory.

Hopcroft and Ullman. Remarks: Without doubt one of the classic definitive theoretical texts on this subject.

Other useful references.


Background and Overview. Basic to all computing is an appreciation of what can and cannot be done with computers. This course aims to give the student a firm grasp of the fundamental models of computing - the basic models of abstract automata and the languages they accept. Automata types are discussed with particular reference to the capabilities and limitations of each type.

Premises. That knowledge of the classification of automata and representational tools is a fundamental pre-requisite in the software/hardware design process.

Attitudes and values.

_Away from:_ Developing automata theory as a branch of pure mathematics.

_Towards:_ Developing understanding by example.

_Knowledge of:_ The hierarchical classification of automata and the languages they accept.

_Understanding of capabilities and limitations of Turing machines._

_Skills:_ Design of FSM and Push-down acceptors

Method of Teaching. 20 hours lectures, plus seminar time.

Methods of Assessment: Students answer 3 questions out of 5 in a 3-hour written examination.
Formal Languages and Automata

CONTENTS

CHAPTER 1 The finite state machine ................................................................. 5
  Introduction ......................................................................................... 5
  Reality versus Abstraction ................................................................. 7
    Example - Levels of representation ................................................. 7
  Formal definition of a (deterministic) FSM .......................................... 8
  Other possible automata types ............................................................ 9
  Non-deterministic FSMS .................................................................... 10
  Making Automata ............................................................................... 12
  Similarities and differences between FSM's ......................................... 13
  The substitution property .................................................................. 15
  Chapter references ............................................................................. 18

CHAPTER 2 Automata and languages .............................................................. 19
  Introduction ......................................................................................... 19
  Defining a language ............................................................................ 19
  Defining a language by means of a Grammar ....................................... 21
  Regular expressions and Finite-state automata ...................................... 22
  Regular expressions / Finite-state languages ........................................ 24
  Deriving FSM acceptors from regular grammars .................................... 25
  From non-deterministic to deterministic ............................................. 26
  Other properties of regular expressions .............................................. 28
  The Pumping lemma .......................................................................... 29
  An application of the Pumping lemma ................................................. 31
  Applications of FSM's in software ..................................................... 32
  Chapter references ............................................................................. 32
  Exercises for Chapter 2 ....................................................................... 32

Chapter 3 Context-free languages and Push-down automata ....................... 35
  Introduction ......................................................................................... 35
  Context-free languages ....................................................................... 35
  Stack automata .................................................................................. 36
  Definition of Stack Automata .............................................................. 38
  NDPDSA's and (deterministic)PDA's are not equivalent ....................... 39
  Context free grammars and NDPDSA's ............................................. 39
  Derivation trees and CFG's .................................................................. 41
  The Pumping Lemma for CFL's ......................................................... 42
  Chapter references ............................................................................. 43
  Exercises for Chapter 3 ....................................................................... 43

Chapter 4 Turing machines: Context-sensitive and Phrase structured languages ........................................................................................................... 44
  Introduction ......................................................................................... 44
  Context-sensitive languages ............................................................... 45
  Turing machines .................................................................................. 46
  Turing's Hypothesis ............................................................................ 49
  Universal Turing machines ................................................................. 49
  Linear bounded automata and context sensitive languages ................. 50
  The halting problem ........................................................................... 50
List of Figures

Figure 1-1 Three representations of one FSM. .................................................. 7
Figure 1-2 A probabilistic automaton. .............................................................. 9
Figure 1-3 FSM version of the probabilistic automaton. ................................. 10
Figure 1-4 A Non-deterministic FSM ............................................................. 11
Figure 1-5 FSM version of the Non-deterministic automaton. ......................... 11
Figure 1-6 Schematic for an automaton ....................................................... 12
Figure 1-7 Difficulties in i/o measurements .................................................. 13
Figure 1-8 Building feasible machines from test results .................................. 14
Figure 1-9 Example for next state-vectors ............................................... 15
Figure 1-10 Five state FSM ................................................................. 17
Figure 2-1 NDFSM for (A#)(B#)* .......................................................... 23
Figure 2-2 NDFSM for $a + ((b + ac)(bc)*b)$ ........................................... 23
Figure 2-3 (a) State. (b) Box. (c) Full state diagram. ................................... 25
Figure 2-4 Acceptor for modified language .................................................. 25
Figure 2-5 Example of a NDFSM ............................................................. 26
Figure 2-6 Equivalent deterministic FSM ................................................... 27
Figure 2-7 The pumping lemma .............................................................. 29
Figure 2-8 Figure for Q2.1 ................................................................. 32
Figure 2-9 Figure for Q2.2 ................................................................. 33
Figure 3-1 Push down stack automata ....................................................... 36
Figure 3-2 Operation of the FSM controller .............................................. 37
Figure 3-3 Derivation tree for $aabbaa$ ...................................................... 42
Figure 5-1 Solution to Q2.1(iii) ............................................................. 53
Figure 5-2 Solution to Q2.2(ii) ............................................................... 53
Figure 5-3 Solution to Q2.3(a) ............................................................... 55
Figure 5-4 Solution to Q2.3(b) ............................................................... 55
Figure 5-5 Solution to Q2.3(c) ............................................................... 55
Figure 5-6 Solution to Q2.5(a) ............................................................... 57
Figure 5-7 Solution to Q2.5(b) ............................................................... 57
Figure 5-8 Solution to Q2.5(c) ............................................................... 58
Figure 5-9 Solution to Q2.8(i) ............................................................... 60
Figure 5-10 Solution to Q2.8(ii) ............................................................. 60
Introduction.

It is an interesting exercise to follow the history of automata theory as a separate concept from that of digital computing. Indeed, this history may be seen as starting in 1937, ten years before the birth of the digital computer as we now know it (Burks, Goldstine and von Neumann, 1947). Automata theory, it can be plausible argued, really began in the mind of the British mathematician Alan Turing as a cross fertilisation of two ideas: he had an interest in computing machinery and was well aware of the potential of electronics. Also as a pure mathematician he was interested in the subject of effective computability. The value of a function, or the solution to some problem, is said to be effectively computable if there exists an unambiguous set of rules which, if applied relentlessly, will terminate after a finite number of steps and give the value. (It is a fairly easy matter for pure mathematicians to construct functions which are not effectively computable). If the rules can be carried out by some very simple machine then there is no doubt that the specification is complete and that we have an ‘effective procedure’. Turing’s thesis, which at first sight may seem extreme, is a sort of converse to this observation:

- Any procedure which can naturally be called effective can be realised by a (simple) machine.

In his 1936 paper Turing defined the class of abstract machines which now bear his name. They are exceedingly simple. Yet Turing discovered that he could set these machines up to perform very complex calculations. His thesis that these machines could perform any effective procedure is closely related to the work of Alonzo Church, Emil Post and S.C. Kleene. An excellent defence of his position can be found in his brilliant article Computing Machines and Intelligence, 1950. We shall touch upon the consequences of his argument later in the course.

Both Church and Kleene, like Turing himself, were concerned with ideas relating to ‘effective computability’. The notion of an effective procedure can be implemented with a system that includes:

- A language for describing rules of behaviour.
- A machine that obeys statements in the language.

Let us turn, for a moment, to the language in which the rules can be expressed. The idea of Post (1943) is that expressions or statements in a logical system or language, whatever else they may seem to be, are in the final analysis nothing but strings of symbols written in some finite alphabet. Even the most powerful mathematical or logical system is ultimately, nothing but a set of rules that tell how some string of symbols may be transformed into another string of symbols. Just as Turing was able to show the equivalence of his broadest notion of a computing machine with the very sharply restricted idea of a Turing machine, so Post was able to reduce his broadest concept of a string transformation system to a family of astoundingly special and simple symbol manipulation operations. We can summarise Post’s view as: any system for manipulation of symbols which could naturally be called a formal or logical system can be realised as one of Post’s ‘canonical systems’.

From the work of Turing and his contemporaries the gestation period lasted some twenty-one years and saw many supporting developments take place. Shannon’s theories of Switching Logic (1938) and Statistical Communication theory (1948) being two examples. Indeed, the period saw the coming of Weiner’s classic book on Cybernetics where, although he mentions meeting Turing (“I spent a total of three weeks in England...I had an excellent chance... to talk over the fundamental ideas of Cybernetics with Mr. Turing.”), there is no further mention of Turing’s ideas on abstract automata. The major events of this period, however, were still to come.

In September 1948 John von Neumann gave a lecture entitled The General and Logical Theory of Automata, which is reprinted in Volume 5 of his collected works. Because he spoke in general terms, there is very little in it that is dated. Von Neumann’s automata are a conceptual generalisation of the electronic computers whose revolutionary implications he was the first to see. He suggested models closely related to the abstract definition of a
Finite-State-Machine (FSM) which we shall examine shortly. The main theme of the 1948 lecture is an abstract analysis of the structure of an automaton which is of sufficient complexity to be able to reproduce itself. He shows that a self-reproducing automaton must have four separate components:

- **Component A.** Is an automaton which collects raw materials and processes them into an output specified by a written instruction which must be supplied from outside. In effect an automatic factory.
- **Component B.** Is a duplicator, an automaton which takes a written instruction and copies it.
- **Component C.** Is a controller, an automaton hooked up to both A and B. When C is given an instruction it first passes the instruction to B for duplication, then passes it to A for action, and finally supplies the copied instruction to the output of A whilst keeping the original for itself.
- **Component D.** Is a written instruction containing the complete specifications which cause A to manufacture the combined system, A plus B plus C.

Von Neumann’s analysis showed that a structure of this kind was logically necessary and sufficient for a self-reproducing automaton, and he conjectured that it must also exist in living cells. Whilst von Neumann did not live long enough to bring his theory of automata into existence he did live long enough to see his insight into the functioning of living cells brilliantly confirmed by the biologists. Five years later Crick and Watson discovered the structure of DNA, and now every child in high school learns the biological identification of von Neumann’s four components.

- **Component A.** Is the ribosomes.
- **Component B.** Is the enzymes RNA and DNA polymerase.
- **Component C.** Is the repressor and derepressor molecules and other items whose functioning is still imperfectly understood.
- **Component D.** Is the genetic materials, RNA and DNA.

So far as we know the basic design of every microorganism larger than a virus is precisely as von Neumann said it should be. Viruses are not self-reproducing in von Neumann’s sense since they borrow the ribosomes from the cells they invade.

Von Neumann’s first main conclusion was that self-reproducing automata with these characteristics can in principle be built. His second main conclusion follows from the work of Turing and, whilst it is less well known, goes deeper into the heart of automata theory. Turing showed that there exists in theory a universal automaton, that is to say a machine of a certain definite size and complication, which, if you give it the correct written instructions, will do anything that any other machine can do! So beyond a certain point, you don’t need to make your machine any bigger or more complicated to get more complicated jobs done. All you need is to give it longer and more elaborate instructions. You can also make the universal automaton self-reproducing by including it within the factory unit (Component A). Von Neumann believed that the possibility of a universal automaton was ultimately responsible for the possibility of indefinitely continued biological evolution. In evolving from simpler to more complex organisms you do not have to redesign the basic biochemical machinery as you go along. You have only to modify and extend the genetic instructions. Everything we have learned about evolution since 1948 tends to confirm that von Neumann was right.

In 1956 Claude Shannon and John McCarthy organised a symposium at Princeton. Shannon of course is a name well known to electronic engineers and McCarthy is one of the main figures in the development of Artificial Intelligence. This was an extraordinarily creative period at Princeton and saw the blossoming of several interrelated subjects including Automata Theory, Artificial Intelligence, and Game Theory. In addition to his other achievements von Neumann, together with Morgenstern, co-founded the Theory of N-person Games. The title of the Princeton symposium was Automata Studies and contained much of the classical work in the field, some of which will be discussed in these lectures.

However, the subject of Automata Theory surely came of age in 1957, with the publication of Chomsky’s influential analysis of language called Syntactic Structures. Chomsky showed that the syntax of a language could be expressed...
as a set of production rules for 'surface' and 'deep' symbols. Although Chomsky aimed at providing a rigorous background to an understanding of natural language, his influence on automata theory was different. It pointed to the fact that one could define artificial languages, and these in turn defined automata which accept such languages. Indeed this is probably much more fundamental in automata theory than in psycholinguistics, mainly because in psycholinguistics one needs to take a rigorous view of meaning (semantics) in order to have a full view of the subject. In automata theory, however, Chomsky's description of grammars led to a classification of automata in terms of the complexities of the language which they accept. We shall pursue this further later in these lectures.

**Reality versus Abstraction.**

In discussing automata theory it is well to understand that there is a clear distinction between the implementation of a particular model in hardware, be it logic circuits or beer cans, and the model itself. The theory of automata is concerned not so much with the practical details of implementation, although engineers must eventually reach this stage, but rather with abstract representations of automata from which we can deduce the properties and limitations of all automata, regardless of their practical implementation.

The following example shows three ways of describing a simple automaton. The first representation is a block diagram of a sequential machine, the second is a state diagram with binary assignment, the third is an abstract state diagram representation.

*Example - Levels of representation.*

![Diagrams](image)

**Figure 1-1** Three representations of one FSM.

In Figure 1.1(a) the Boolean function of the hardware determines a state diagram Figure 1.1(b), which is labelled in terms of logical 0 or 1 values on the input (A, B), output (C) and feedback Q connections and AB/C on the arcs.
It should be noted that $Q'$ is the ‘next’ value of the D flip-flop and $Q$ the ‘current value'.

The state diagram is derived as follows. We have two internal states: the 0 and 1 state of the flip-flop. Suppose that $Q = 0$, $A = 0$, and $B = 0$; the logic function for $Q'$ is then

$$Q' = \neg A \cdot B + \neg A \cdot Q + B \cdot Q = 1.0 + 1.0 + 0.0 = 0.$$ 

Here $\neg A = \text{not}(A)$, $+ = \text{OR}$ and $.$ = $\text{AND}$. Therefore the next state will again be 0 and we can draw an arrow from state 0 to itself labelling it with the input values $A = 0$, $B = 0$. The corresponding output value $C$ is

$$C = \neg A \cdot Q + B \cdot Q = 1.0 + 0.0 = 0$$

We label the arrow just drawn with 00/0, meaning that for the input combination 00 at the given state the output is 0. By repeating this procedure for all combinations of $Q$, $A$ and $B$ we obtain the state diagram of Figure 1.1(b).

To the state diagram of Figure 1.1(b) there corresponds another, shown in Figure 1.1(c), which has the following labelling:

- AB = 00 is called $a$
- AB = 01 is called $b$
- AB = 11 is called $c$
- AB = 10 is called $d$
- $Q = 0$ is called $q$
- $Q = 1$ is called $q'$
- $C = 0$ is called $o$
- $C = 1$ is called $p$

The point is that one could assign totally different binary values to the quantities \{a, b, c, d\}, \{q, q'\}, and \{o, p\}, in which case the corresponding sequential machine would be different but the abstract representation in Figure 1.1(c) would be the same. Thus Figure 1.1(c) describes an equivalence class of distinct sequential machines.

- The basic objects of Automata theory are diagrams of the type shown in Figure 1.1(c).

**Formal definition of a (deterministic) FSM.**

**Definition:** A Finite State Machine is an ordered 5-tuple $<I, Z, Q, d, w>$ where

- $I$ is a finite set of discrete input messages.
- $Z$ is a finite set of discrete output messages.
- $Q$ is a finite set of internal states.
- $d$ is a rule which given the current state $q(t) \in Q$ and an input $i(t) \in I$, defines the next state $q(t+1)$ in $Q$. i.e. $q(t+1) = d(q(t), i(t))$. 
- $w$ is a rule which given the current state $q(t)$ in $Q$ and an input $i(t)$ in $I$, defines the corresponding output $z(t)$ in $Z$. i.e. $z(t) = w(q(t), i(t))$. 

A subtle point may need stressing here, namely that the output $z(t)$ is measured immediately, but by the time one measures the state it has changed to $q(t+1)$.

**Example.** In order to illustrate this rather abstract definition consider the example given in the last section. Here

- $I = \{a, b, c, d\}$ four possible input messages,
- $Z = \{o, p\}$ two possible output messages, and

---

1 A Flip-Flop remembers its current state until pulsed by a clock when it may change its current state depending on the current input.
\( Q = \{ q, q' \} \) two possible internal states.

The defining table for the function \( d \) is

\[
\begin{array}{cccc|c}
  i(t) & a & b & c & d & \text{resulting state} \\
  q(t) & q & q & q' & q & q(t+1) \\
  q' & q' & q' & q' & q & \\
\end{array}
\]

Similarly the defining table for \( w \) is

\[
\begin{array}{cccc|c}
  i(t) & a & b & c & d & \text{resulting output} \\
  q(t) & q & o & o & o & z(t) \\
  q' & p & p & p & o & \\
\end{array}
\]

This representation of an automaton is called the Mealy model. There are other, essentially equivalent, representations (such as the Moore model discussed, for example, in Hopkins and Moss) but when we deal with larger numbers of states, these table descriptions become awkward and it is more convenient to use state diagrams.

The assumption that the number of possible internal states is finite is fundamental to the theory. It implies that, by its present and future behaviour, the machine can only distinguish a finite set of ‘histories’ from the much much larger class of all possible histories. The histories which, from the machine’s viewpoint, are indistinguishable form an equivalence class, in the technical sense, which in effect defines the corresponding state. Because all quantities processed, the i/o messages and the internal states themselves, are finite in number this model is known as a Finite-State-Machine (FSM).

Exercise. 1.1 Prove for a deterministic FSM that given any input string \( xy \) (\( x \) is read first then \( y \)) we have

\[ d(q, xy) = d(d(q, x), y). \]

[Hint: Use induction on the length of \( y \).]

(Remark: Do it now - we will need it later.)

Other possible automata types.

Other possible automata models which it is natural to consider are

- Continuous state automata,
- Probabilistic automata,

We shall briefly consider each of these in turn before returning to our discussion of FSM’s.

A continuous state automaton is one in which a continuum of possible states occur, as opposed to the finite discrete states of a FSM. Practically any analog device is a continuous state machine. It is possible to approximate the behaviour of any continuous state machine by a FSM with a sufficiently large number of states. This observation is of some interest since, for example, it indicates a relation between FSM’s and the state space representation of continuous control theory (see for example Elgerd 1967).

\[ \text{Figure 1-2 A probabilistic automaton.} \]
A probabilistic automaton is a machine whose behaviour is not entirely determined by its state because it includes random variables not considered to be part of its state. More precisely a probabilistic automaton is one whose definition must state the probabilities of transitions from one state to the next for all possible inputs. Again we should ask how this idea relates to the definition of a FSM given above. Consider the following example.

Example. In Figure 1.2 we see there are two states A, B but now, for example, when we say that $i_1$ causes a transition between state A and state B we say additionally that it does so with probability 0.5 (this is written in brackets next to the transition-causing input label). Readers familiar with statistics will recognise this as an alternative description of a Markov chain.

Now consider Figure 1.3, where we see a FSM with four states and the same input parameters $i_1$ and $i_2$ as the probabilistic automaton. The additional features are:

- A new input labelled C.
- The splitting of the original two states.

If we imagine that C is an input which is applied sufficiently often, but in a way uncorrelated with inputs $i_1$ and $i_2$, we see that this introduces an uncertainty regarding which pair of states, $(A_1, A_2)$ or $(B_1, B_2)$, is the current one. Thus the second figure can be regarded as a simulation or approximation of the first. One sees that, as with the continuous case, it is possible to approximate a probabilistic automaton with a FSM having a much larger number of states. In fact if the probabilities of state transition from A attached to the input $i_1$ were (0.01, 0.99), as opposed to (0.5, 0.5), then we would need 100 states in the FSM to represent one state in the probabilistic automaton.

Non-deterministic FSMs.

We next introduce the notion of a non-deterministic FSM (NDFSM). The NDFSMs are mostly used in the discussion of 'acceptors', automata which decide whether a given input string is syntactic or not. As such the outputs of a NDFSM are usually rather simple. The NDFSM either 'accepts' or 'rejects' the whole of the input string. Consequently in our discussion of NDFSMs we shall tend to ignore the question of intermediate outputs and concentrate on how changes of state take place.

Definition. The NDFSM is a generalisation of the idea of the (deterministic) FSM. We note that in the definition of an FSM the next state function $d : Q \times I \to Q$. In the definition of an NDFSM we replace this function by $d : Q \times I \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the set of all subsets of $Q$.

We further identify a special subset $A \subseteq Q$, where $A$ is the set of 'acceptor' or final states. By extending the mapping $d$ we intend the following interpretation. At any given state the NDFSM can move to one of a finite set of possible next states. If given the whole input string there exists a sequence of state transitions which leads to a state in $A$ then the string is accepted.

This definition may at first sight seem somewhat puzzling. One cannot help asking 'how does the machine decide which is the next state to go to?'. The point is that the definition of 'accept' is, in the first instance, an existence assertion, not a deterministic algorithm. Given an input string, we say it is accepted if there is SOME path through the state diagram which, when the last symbol is processed, leaves the machine in one of the accept states of $A$. There are three ways we can conceptualise this process.

Firstly, there is the 'cloning model'. Each time the NDFSM reaches a point where there are several alternative states it could go to we can imagine creating several copies of the machine, one for each alternative. We then allow each
of these machines to continue along its respective alternative path until the next ambiguous point is reached, when we once again clone each copy. In this way more and more machines are produced, but it is a finite process, and eventually the whole input string is processed. We can then say that the string is accepted if there is at least ONE of the machines which has reached a state in \( A \).

Secondly we can imagine the process of determining whether or not a string is to be accepted as equivalent to a tree search. The automaton will search for a sequence of state transitions which leads to a state in \( A \). This search is finite, since the input string and the total number of states is finite. Although the automaton may reach a ‘dead end’ and be forced to back up and try another route, eventually all routes can be tried and a decision as to ‘success’ or ‘failure’ reached.

The problem with this model is that, at first sight, it appears to require computational resources beyond the capability inherent in the original definition of an FSM.

The natural machine to perform the ‘back tracking’ is a Push Down Stack Automaton (PDSA), which - as we shall see - is genuinely more powerful than an FSM. However this appearance is, as it turns out, illusory. Because the number of states and input messages is finite there is a definite upper bound on the depth of back tracking required. Thus to implement this model on a PDSA we should only require a PDSA with a bounded stack memory. It can be shown that such a special kind of PDSA is equivalent to an FSM so that no logical contradiction is involved after all.

Finally we could adopt the oracular model of non-determinism and simply say that at every decision point the machine makes the ‘right’ decision, i.e. it will end in an accept state if this is possible.

To summarise: a non-deterministic automaton is a generalisation of a FSM in which, for a given input, the next state is replaced by a finite set of possible next states. As we shall see, such automata play an important role in the analysis of languages. We shall also see that every NDFSM can be replaced by an equivalent FSM.

Thus the NDFSMs are quite distinct from probabilistic automata, despite any confusion which may seem to arise from the similarity in everyday usage of the words ‘probabilistic’and ‘non-deterministic’. In the description of a FSM we say - given this input a transition ‘will occur to such and such a state’.

With a probabilistic automaton the operative phrase is ‘will occur with probability \( p(i) \) to state \( y(i) \)’, where \( i \) runs over a finite set of values; the random variables are extra constructs attached to the model. In a non-deterministic automaton, although the next state is not immediately determined from the definition, there are no associated probability values and so the operative phrase is ‘may occur to one of the states \( y(i) \)’, where once again \( i \) ranges over a finite set.

Example. A testing system is required for strings of 0’s and 1’s such that it will pass all strings starting with a 0 and followed by any number, but at least one, of 1’s. Figure 1.4 summarises this task, and is an example of a
non-deterministic machine.

The states are labelled $S$ for Start state, $I$ for Intermediate state, $P$ for Pass state, and $X$ for Fail state. The symbol $R$ stands for a reset input. We interpret this diagram, in a slightly different manner from the state diagram of a FSM, as follows:

Suppose the system is reset to state $S$ and a sequence of inputs has occurred. Can one find a path through the NDFSM which follows the inputs and ends in state $P$? If one can, the string is accepted. We note that strings that are not accepted generate state $X$.

Thus, for example, we have:

<table>
<thead>
<tr>
<th>Accepted strings:</th>
<th>01</th>
<th>011111</th>
<th>0111111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not accepted strings:</td>
<td>101</td>
<td>01111110</td>
<td>010</td>
</tr>
</tbody>
</table>

We shall see later, that this is a very convenient way of describing the necessary action of some machines which check computer languages. We shall also see that every NDFSM may be transformed into a FSM by means of a very simple algorithm. In fact the application of this algorithm to the NDFSM of Figure 1.4 gives the FSM of Figure 1.5. The reader can easily check that the latter is functionally equivalent to the former.

Although we have only given a brief account of probabilistic and continuous automata one should mention that it is perfectly possible to write down formal definitions for these models analogous to that given for a FSM. Still, as will become clear, the FSM is a fundamental model for a vast range of information processing mechanisms, both natural and man-made. In the next section we shall see how these models can be made into real systems.

**Making Automata.**

Recalling the abstract 5-tuple definition of a FSM $<I, Z, Q, d, w>$, it becomes clear that the first step in making an automaton is to encode $I, Z$ and $Q$ into some concrete form. There is no problem if one assumes some form of digital, indeed binary, encoding. One imagines that at any moment in time the current elements of these sets are held in binary registers. Therefore we define three registers labelled $I, Z$ and $Q$, each register containing at least $[\log N]+1$ bits, where the log is to base 2, $[X]$ is the greatest integer $\leq X$, and $N$ is the number of distinct messages in the set in question (i.e. $I, Z$, or $Q$). The automaton is ‘made’ by setting up a logical transformation circuit, which first relates the present contents of $I$ and $Q$ to the next contents of $Q$ according to $d$, and the value of $Z$ according to $w$. This arrangement is shown in Figure 1.6.

If the automaton were made by an electronics engineer, he would use either logic gates (AND, OR, NAND) etc. to implement $d$ and $w$, or fixed memory circuits acting as look-up tables where $I$ and $Q$ together act as an address, and the values of $Z$ and next $Q$ are stored at this address.

A computer scientist might do things a little differently.

The following is an example of a BASIC program which emulates a general purpose FSM.
Program Comment

10 INPUT K Number of states.
20 INPUT L Number of input symbols.
30 DIM D(K,L) Dimension next state function.
40 DIM W(K,L) Dimension output function.
50 FOR X=1 TO K For each state.
60 FOR Y=1 TO L For each input symbol.
70 INPUT D(X,Y) Input D value.
80 INPUT W(X,Y) Input W value.
90 NEXT Y Next input symbol.
100 NEXT X Next state.

The functions \( d \) and \( w \) are now defined from data supplied by the programmer. To use the automaton the program goes on as follows.

110 INPUT S Input initial state.
120 INPUT I Input input message.
130 PRINT W(S,I) Print output message.
140 S=D(S,I) Compute next state.
150 GOTO 120 Get next input message.

Clearly, in both these examples there is an implicit clocking mechanism. In the first case some clocking arrangements for the \( Q \) register are required. In the case of the BASIC program the clocking is provided in one sense because the program is sequential and in another by the hardware which actually runs it.

**Similarities and differences between FSM's.**

Some important questions one might ask, both from the point of view of theory and of engineering, are:

Given two identical 5-tuple descriptions of FSM’s, to what extent can identical descriptions lead to different machines?

also In what way can different 5-tuple descriptions lead to identical input-output behaviour?

Theoretically, answers to these questions will tell us whether or not the FSM is a powerful tool for modelling general systems. On the engineering side, these questions bear on whether or not a machine has been efficiently made. For example, in this context a corollary to the second question is:

Does a machine exist which has the same i/o

---

**Figure 1-7** Difficulties in i/o measurements
CHAPTER 1 The finite state machine

description as the machine I have designed, but which can be made more cheaply?

By ‘cheaper’ one usually means simpler circuitry or a lower component count.

One sad fact that becomes clear from the start is that no amount of i/o experimentation will guarantee the revelation of details of the internal state structure. Consider the FSM’s in Figure 1.7.

Recall that the labels on the transitions are of the form i/o, where i is the input causing the transition and o is the output resulting from it. The reader can verify that, if only measurements of i/o are available, there is no experiment that will distinguish between the two automata in Figure 1.7(a) and Figure 1.7(b). One would need a ‘can opener’ in order to determine the real state structure.

What then can be achieved by i/o measurements on an automaton? Before answering this question we need to look at another factor that cannot be revealed via i/o experiments. Examine the three FSM’s of Figure 1.7.

Consider the machine in Figure 1.7(c). Say this automaton starts in state A. Note that during an i/o experiment it could enter state B and then state C. Alas no manner of further experimentation will lead the system back to state A or B, and the only information that further experiments can reveal concerns states C and D. Such a FSM is said to be not strongly connected.

Definition: A strongly connected FSM is one for which given any two states there exists an input sequence which will cause the system to pass from one state to the other.

Plainly an i/o experiment can lead only to a minimal state model of the strongly connected part of an automaton. That is we can say ‘there are at least these states’, but the model is only as accurate as the experimental data permits. Of all the FSM’s that have the same i/o behaviour (which for a given behaviour pattern is infinitely many) some have fewer states than others. Consequently there is at least one which has fewest states (this deduction is obvious but mathematicians should note the appeal to well ordering). The i/o measurements determine the amount of state structure that has been explored, and therefore determines a minimal model.

To illustrate these points consider the following progression of i/o measurements for a machine with an unknown number of states:

<table>
<thead>
<tr>
<th>Input</th>
<th>1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
The possible configurations for this experiment are shown in Figure 1.8(a) (the symbol * indicates the state at the end of the experiment). We now continue with

\[
\begin{array}{ccc}
\text{Input} & 0 & 1 & 0 \\
\text{Output} & 0 & 0 & 1
\end{array}
\]

This yields possibilities, some of which are shown in Figure 1.8(b). If the experiment were to stop here, all of these models would be valid. If one insisted that the machine were strongly connected models such as $a\delta$ or $b\xi$ would remain valid. If one were looking for models completely characterised by the experiments so far then only models such as $a\delta$ or $b\xi$ would remain.

Plainly the question ‘which model is right?’ has no real answer. Our examples indicate that in order to answer the question, apart from knowing that the machine is strongly connected one must also know how many states it has. Then the completely characterised model which has the right number of states emerges as a good candidate for the ‘right model’.

**The substitution property.**

Given an i/o specification for a FSM the question arises as to how one might find a minimal state representation. Traditionally minimisation of the number of states was recommended to students of logic design on the basis that fewer states require simpler and cheaper circuits. In reality this is often not the case, but the problem of finding a minimal state representation is worth pursuing, albeit briefly.

Before we consider equivalence between machines we first look at the concept of the ‘next state vector’, or NSV. Consider the following FSM in Figure 1.9. Plainly a complete description of this machine is given by the two tables below, one for $d$ and one for $w$.

**Table d**

<table>
<thead>
<tr>
<th>Input</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>j</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

**Table w**

<table>
<thead>
<tr>
<th>Input</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>j</td>
<td>C</td>
<td>p</td>
<td>p</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next state Output

We can illustrate the NSV concept by using the first table. The NSV for $i$ is (B, C, C, D) and for $j$ is (A, B, D, A).

- Thus the NSV is a description of the function defined by a given input from the set of states to the set of next states.

Now consider the following situation. Suppose that for some FSM we have

- Set of states ($S$) $A B C D E F G$
- NSV under $i$ $D E F A B C A$
- NSV under $j$ $A B A F D G G$. 

15
We now arbitrarily partition the set of states $S$ into blocks. For example, we could break up $S$ into three blocks

<table>
<thead>
<tr>
<th>Partition 1:</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A, B)</td>
<td>(C, D)</td>
<td>(E, F, G),</td>
</tr>
</tbody>
</table>

or into two blocks

<table>
<thead>
<tr>
<th>Partition 2:</th>
<th>$b_1$</th>
<th>$b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A, B, C)</td>
<td>(D, E, F, G).</td>
</tr>
</tbody>
</table>

To discover equivalence between machines, the idea of a partition of states into blocks is used together with that of a NSV. Taking Partition 1 first, see how the NSV maps into the blocks.

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A, B)</td>
<td>(C, D)</td>
<td>(E, F, G)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NSV under $i$:</th>
<th>(D, E)</th>
<th>(F, A)</th>
<th>(B, C, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b2, b3)</td>
<td>(b3, b1)</td>
<td>(b1, b2, b1)</td>
<td></td>
</tr>
</tbody>
</table>

We repeat this exercise for Partition 2.

<table>
<thead>
<tr>
<th>Partition 2:</th>
<th>$b_1$</th>
<th>$b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A, B, C)</td>
<td>(D, E, F, G)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NSV under $i$:</th>
<th>(D, E, F)</th>
<th>(A, B, C, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b2, b2, b2)</td>
<td>(b1, b1, b1, b1)</td>
<td></td>
</tr>
</tbody>
</table>

The most noticeable difference is that for the given NSV every block in Partition 2 maps to a subset of a single block, which is not the case for Partition 1.

Now let us see how this is useful. Consider an example where A, B, C all give output 1 and D, E, F, G all give output 0. Then for input $i$ we have the NSV (D,E,F,A,B,C,A) as above, whilst for input $j$ we have the NSV (A,B,A,F,D,G,G). Taking both $i$ and $j$ into account the block map becomes:

<table>
<thead>
<tr>
<th>Partition 2:</th>
<th>$b_1$</th>
<th>$b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A, B, C)</td>
<td>(D, E, F, G)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NSV under $i$:</th>
<th>(D, E, F)</th>
<th>(A, B, C, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b2, b2, b2)</td>
<td>(b1, b1, b1, b1)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NSV under $j$:</th>
<th>(A, B, A)</th>
<th>(F, D, G, G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b1, b1, b1)</td>
<td>(b2, b2, b2)</td>
<td></td>
</tr>
</tbody>
</table>

Note that for Partition 2 the NSV under $j$ takes $b_1$ into $b_1$ and $b_2$ into $b_2$. In doing an i/o experiment it becomes clear that no amount of experimentation will distinguish between states $X = (A, B, C)$, with output 1, on the one hand or states $Y = (D, E, F, G)$, with output 0, on the other. In fact, the experimenter would report a simple two-state machine with the following mappings.
In terms of i/o behaviour this two-state machine is equivalent to the original seven-state machine. This has come about for two reasons.

(a) Each block maps into a single block under all possible inputs.
(b) Each block is output consistent.

Definition: A partition of the set of states of an automaton is said to have the substitution property if and only if (a) holds. Such a partition is often called an SP-partition.

The situation with regard to state reduction of an i/o specified automaton can now be summarised as follows.

- Given an SP-partition of the states of an automaton which satisfies (b) then every block may be replaced by a single state and the resulting FSM is the minimal state automaton.

Although of theoretical interest, this technique has rather limited practical application, since most systems have a very large number of states and the corresponding number of partitions is enormous. Moreover one may easily find that a large number of partitions have the substitution property, and some may well result in simpler machines than others.

Example - State Reduction/Minimisation. Design a five state FSM with a single channel binary input and a single channel binary output. The system is synchronous and outputs a logical 1 when the fourth successive input is a 1, it continues to output 1 for further successive inputs of 1. In all other cases the system outputs 0. Any occurrence of an input 0 resets the FSM to its initial state. Draw the state diagram and give the next-state and output tables.

List the state partitions of this FSM which have the Substitution Property and find one of these partitions which is output consistent. Hence deduce an equivalent four state representation for the FSM.

Using the state assignment

<table>
<thead>
<tr>
<th>Binary digits:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>q(0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>q(1)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q(2)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>q(3)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

for the 4-state FSM, where $q(0)$ is the initial state and $q(1), q(2), q(3)$ are the transition states for successive 1’s, deduce the logic expressions for the new values $x', y'$ in terms of the old values $x, y$ and the input $i$ whenever a state transition occurs. Also deduce a corresponding expression for the output.

Draw a circuit using AND gates, OR gates and inverters which implements this FSM and state the hardware requirements.

Solution.

The Five state FSM is shown in Figure 1.10.
CHAPTER 1 The finite state machine

Tables:

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present state</td>
<td>Next state</td>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q(0)</td>
<td>q(0)</td>
<td>q(1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>q(1)</td>
<td>q(0)</td>
<td>q(2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>q(2)</td>
<td>q(0)</td>
<td>q(3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>q(3)</td>
<td>q(0)</td>
<td>q(4)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q(4)</td>
<td>q(0)</td>
<td>q(4)</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Partitions with the SP property:

Partition 1. \{ (q(0)), (q(1), q(2), q(3), q(4)) \}
Partition 2. \{ (q(0)), (q(1)), (q(2), q(3), q(4)) \}
Partition 3. \{ (q(0)), (q(1)), (q(2)), (q(3), q(4)) \}

Of these three it is clear from the above table that only the third is output consistent. So we can identify states q(3) and q(4) and the resulting FSM has four states. The assignment:

<table>
<thead>
<tr>
<th>Binary digits:</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>q(0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>q(1)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q(2)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>q(3)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

leads to the logic expressions

Next value \( x' = i.(x + y) \)
Next value \( y' = i.\neg x \)
Output \( = i.x.\neg y \)

where \( i \) is the input and \( \neg x \) denotes the logical complement of \( x \).

This leads to a representation requiring four two-input AND gates, one two-input OR gate and two inverters.

As well as state minimisation the SP-property also has a role to play in algorithms for state assignment. The 'state assignment' problem may be roughly described as finding a binary label for each state in such a way that (given the particular FSM) the hardware required to implement the logic of state transitions and output is as simple as possible. To some extent these algorithms will depend on the particular technology used and a detailed discussion of optimal state assignment would lead us along a lengthy digression. The interested reader could begin by consulting Lewin's Design of Logic Systems.

Chapter references

The Collected works of John von Neuman.
CHAPTER 2 Automata and languages

Introduction.

An obvious limitation of a FSM is the finite number of states. This is a realistic and practical limitation, since any man-made machine must have finite limitations. However consider the example of a 'simple' home computer circa 1980. It provided its user with at least 16K of 8 bit words for a cost considerably less than a bicycle. The number of states provided by this amount of fast local storage is 2 to the power 16000 ×8 or approximately 10 to the power 43,000, which is indeed an astronomical figure. If one adds to this the amount of backing store that disk or tape may provide one can easily see that even such simple machines are more akin to infinite state machines, for which the FSM models we have used are not so much inadequate as inappropriate. There is no way one could handle FSM concepts such as NSV's or state diagrams because of their enormous size. Yet such machines exist and people work with them despite the enormity of the number of states. In what follows we shall look at automata models better suited to modelling such systems.

The central theme of such models is the idea that automata may be classified in terms of the type of language they accept. The classification scheme we shall discuss is known as the Chomsky hierarchy and involves four levels of language classification. We can summarise these as follows

<table>
<thead>
<tr>
<th>Automata type</th>
<th>Language type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>Regular expressions</td>
</tr>
<tr>
<td>Push down or Stack</td>
<td>Context free</td>
</tr>
<tr>
<td>Linear bounded</td>
<td>Context sensitive</td>
</tr>
<tr>
<td>Turing machine</td>
<td>Phrase structured</td>
</tr>
</tbody>
</table>

The languages are of course artificial as distinct from the natural languages used for written or spoken communication between people. The advantage of using artificial languages is that it avoids the pitfalls of more informal methods and provides a rigorous axiomatic foundation upon which a formal study may be based. In the sections which follow we shall define and discuss each of the language types mentioned above, but first we clarify the idea of an artificial language.

Defining a language.

In our discussion of languages we shall assume that the linguistic elements are strings of symbols, rather than any other arrangement (for example two dimensional patterns) that one may imagine. Given this assumption, the two essential ingredients of a language are:

(a) An alphabet or vocabulary. These are the basic units which constitute the symbols of the language. A string is a finite sequence of symbols from the alphabet. It should be noted that the null string, i.e. the string of length zero which contains no symbols, is always a permitted string.

Example. For example, the alphabet of a computer language might contain symbols such as +, -, :, LET, GOTO, END, etc. Indeed it becomes necessary to see words as the basic units of natural language, although it would not be impossible to think of letters as such symbols. In this case an example of a string might be LET X=Y+Z:GOTO100.

(b) A set of rules. These enable one to check whether or not combinations of symbols, i.e. strings, belong to the language.
These requirements are rather free and easy, so let us see if we can construct some arbitrary language.

Example. Language L1 consists of the following basic symbols:

- X meaning 'scissors',
- [ ] meaning 'paper',
- O meaning 'stone'.

The rule of L1 is that any symbol can follow another as long as it does not dominate in the sense of the children’s game. Thus

- scissors dominate paper (cutting),
- paper dominates stone (wrapping),
- stone dominates scissors (breaking).

In the game two players present symbols simultaneously and the player presenting the dominating symbol wins. Thus a sequence in L1 beginning with X could be

```
X X [ ] O O X [ ] O X [ ] [ ] etc.
```

On discovering sequences containing O [ ] one could decide that these do not belong to L1, since [ ] dominates O (paper wraps stone). Although it is quite feasible to imagine a FSM which, at the end of a given sequence, will signal whether or not the sequence belongs to L1, it is also easy to see that slight modification of the rules would make this less easy.

For example, if we define the rules as requiring that at the end of a sequence the majority (or some other proportion) of the consecutive pairs should not have a dominant second symbol, then the FSM design would depend on the length of the sequence one expects to encounter. However, it is quite possible to specify the rules of a language without reference to the length of strings (we did this in the previous sentence), so this creates a disparity between the language specification and the FSM model. In fact with the new rules the FSM is inappropriate due to its finiteness.

The checking model for a language has a specific name, it is called an acceptor. That is it accepts a string if it belongs to the given language, otherwise the string is rejected. We shall look at the classification of languages largely in terms of the implications they have on such acceptors.

Before proceeding it is best to clarify an important point. All the formal work done on languages in automata theory relates to syntax, i.e. the grammatical structure of the language. Very little of this relates to meaning or semantics, which is a different question and one that is addressed in the study of Artificial Intelligence. To illustrate these terms very briefly

'"The cheese ate the mouse.'

has the correct syntax, but its semantic content is incorrect. In general automata theory does not deal with semantics.

Note that the empty set $\emptyset$ and the set \{ $\lambda$ \}, consisting of the empty string, are languages and that they are distinct. The latter, \{ $\lambda$ \}, has a member, whilst $\emptyset$ does not. The set of palindromes (strings that read the same forwards and backwards) over the vocabulary $V =$ \{ 0 , 1 \} is an infinite language. Some members of this language are $\lambda$, 0, 1, 00, 010, and 1101011.

Note that the set of all palindromes over some infinite set of symbols $V$ is technically not a language because its strings are not built up from a finite vocabulary.

Another language is the set of all finite strings over a fixed vocabulary $V$. We denote this language by $V^*$. For
example, if
\[ V = \{ a \}, \text{ then } V^* = \{ \lambda, a, aa, aaa, \ldots \}. \]
If
\[ V = \{ 0, 1 \}, \text{ then } V^* = \{ \lambda, 0, 1, 00, 01, 11, 000, \ldots \}. \]

**Warning:** note the inclusion of the null string \( \lambda \)

**Defining a language by means of a Grammar.**

When one tries to classify different languages one mainly looks for differences between the rules. In this way one can categorise rules into classes and, consequently, languages into corresponding classes. To do this we need a precise form into which the rules of a language may be cast: this is called a (transformational) Grammar. The term 'grammar' is used by analogy with the way that the syntactic rules of natural languages, such as English or French, are usually specified.

**Definition.** A grammar consists of the following:

(a) A vocabulary \( V \) (as before), but this time split into two disjoint sets of symbols:

A finite set of terminal symbols \( V(T) \). The elements of \( V(T) \) will normally be denoted by lower case letters \( a, b, \) etc. Terminal symbols are the symbols that actually appear in the strings of the language.

A finite set of non-terminal symbols \( V(N) \). The elements of \( V(N) \) will normally be denoted by upper case letters \( A, B, \) etc. Non-terminal symbols do not actually appear in the strings of the language but are used by the production rules (see (b) below) to generate strings of terminals that are syntactically correct. Non-terminal symbols are often called variables.

(b) A finite set \( P \) of production rules. Each production is of the form \( x \rightarrow y \), where \( x \) and \( y \) are strings of symbols from \( V^* \), with \( x \) not equal to the empty string. Thus \( x \) and \( y \) are finite strings of variables or terminals. Elements of \( V^* \) will normally be denoted by lower case letters \( x, y, \) etc. The arrow symbol ‘\( \rightarrow \)’ is read as ‘may be replaced by’.

(c) A start symbol \( S \), which is usually one of the \( V(N) \).

In fact this definition covers the most general type of grammar we shall consider, the phrase structured grammars, but we shall begin by considering some restricted special cases.

**Example.** Consider a grammar \( G = \langle V(N), V(T), P, S \rangle \) where

\[
\begin{align*}
V(N) &= \{ S \}, \\
V(T) &= \{ a, b \}, \\
P &= \{ S \rightarrow aSb, S \rightarrow ab \}.
\end{align*}
\]

By applying the first production \( n - 1 \) times, followed by an application of the second production, we have

\[
S \rightarrow aSb - aaSbb - \ldots - a^{n-1}Sb^{n-1} - a^n b^n
\]

[in this context the \( a^n \) symbol means \( a \) repeated \( n \) times].

Furthermore, the only strings in the language generated by \( G \), \( L(G) \) say, are \( a^n b^n \) for integer \( n \) greater than zero.

To prove this we argue as follows. Each time \( S \rightarrow aSb \) is used, the number of \( S \)'s remains the same. After using the
production \( S \rightarrow ab \) we find that the number of \( S \)'s in the sentential form decreases by one. Thus after using \( S \rightarrow ab \), no \( S \)'s remain in the resulting string. Since both productions have an \( S \) on the left, the only order in which the productions can be applied is \( S \rightarrow aSb \) some number of times followed by one application of \( S \rightarrow ab \). Thus \( L(G) = \{ a^n b^n : n > 0 \} \) as required.

In this example we could interpret \( S \) as the symbol \( ( \) ', a as the symbol \( ( \) and \( b \) as the symbol \( ) \). In this case \( L(G) \) now consists of the set of all finite strings of well nested brackets, i.e. strings of \( L(G) \) are of the form (((())))) etc.

Exercise 2.1. Describe the typical strings of the language generated by adding the production rule \( S \rightarrow SS \) to \( P \) in the above example.

Plainly it is precisely the nature of the production rules which define the structure of a language. The Chomsky hierarchy, mentioned previously, classifies languages in terms of the production rules used by their grammars. The classification is of a kind where the more restricted language is always a special case of the next level. Thus, for example, all context sensitive languages are phrase structured, but not all phrase structured languages are context sensitive. We shall begin by studying the languages which correspond to FSM’s.

**Regular expressions and Finite-state automata.**

The languages accepted by FSM’s are easily described by means of simple expressions called regular expressions. We shall shortly give a formal definition of a regular expression, but loosely a regular expression is a means of describing a set of strings. The following examples should serve to illustrate the idea.

**Examples.**

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Typical element of set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^* )</td>
<td>( a ) repeated any number of times</td>
</tr>
<tr>
<td>( (a^<em>)(b^</em>) )</td>
<td>Any number of ( a )'s followed by any number of ( b )'s.</td>
</tr>
<tr>
<td>( (abc)(d^<em>)(ca)^</em> )</td>
<td>( abc ) followed by ( d ) any number of times followed by ( ca ) any number of times.</td>
</tr>
<tr>
<td>( (a + b)^<em>(aa)^</em> )</td>
<td>Any string of ( a )'s or ( b )'s followed by any even number of ( a )'s.</td>
</tr>
</tbody>
</table>

N.B.

1. Any number of times includes possibly zero times.

2. \( (a + b)^* = (a + b)(a + b)(a + b) \ldots (a + b) \).

We observe that these expressions can all be assembled from very few terms and connectives. We require in fact only the terminal symbols themselves and three connectives

- 'any number of times'
- 'followed by'
- 'or'

We now proceed to characterise the collection of all regular expressions.

(i) Any terminal symbol \( t \) is a regular expression. It describes the set consisting of precisely the (one letter) string \( t \).

(ii) If \( E \) and \( F \) are regular expressions then so is \( EF \) and this describes the set \( \{ x : x = ef \} \), where \( e \) is a string in the set described by \( E \) and \( f \) is a string in the set described by \( F \).

(iii) If \( E \) and \( F \) are regular expressions then so is \( E + F \) and this describes the set \( \{ x : x = e \text{ or } f \} \), where
e is a string in the set described by $E$ and $f$ is a string in the set described by $F$.

(iv) If $E$ is a regular expression then so is $E^*$.

**Definition.** The *regular expressions* are all those that can be constructed from the above four rules and no others.

Plainly a regular expression over a finite set of terminal symbols defines a language, namely the set of strings $L$ described by the regular expression. These languages are called *regular* or *Finite state languages*. A reasonable question to ask at this stage is:

‘What are the production rules which define this language?’

However we shall put aside this question for the moment and first briefly examine the connection between FSMs and regular expressions.

Given a regular expression it is possible to construct a NDFSM which is the acceptor for the corresponding set of strings.

*Example.* Consider the regular expressions over the terminal symbols $A, B, ..., Z, a, b, ..., z, 0, 1, ..., 9$

\[
A\# = A + B + \ldots + Z + a + b + \ldots + z
\]

\[
B\# = A + B + \ldots + Z + a + b + \ldots + z + 0 + 1 + \ldots + 9
\]

Thus $A\#$ can be any alphabetic symbol and $B\#$ can be any alpha-numeric symbol. Then the regular expression

\[
(A\#)(B\#)^* 
\]

represents, for example, any valid ISO Pascal identifier (e.g. a variable or procedure name). Figure 2.1 gives the equivalent NDFSM.

N.B. The diagram is incomplete. The convention in this and other similar figures is that if at any state an input is received which is not indicated on the diagram then the string fails.

*[Aside.]* If you want to experiment with regular expressions in UNIX try using a program called **grep** which searches files for lines containing specified regular expressions. For example to search for all Pascal identifiers in a file ‘foo’ (not very useful!) one would use the syntax

\[
grep \^[A-Za-z][A-Za-z0-9]*\] foo
\]

*Example.* Consider the following regular expression

\[
a + ((b + ac)(bc))^* b
\]

This is easily seen to be equivalent to the NDFSM of Figure 2.2

When checking these diagrams remember that for a given input string **if there is a path leading to the**
terminal state the string is accepted, if none of the possible paths generated by the string lead to the final state the string is rejected.

However we mentioned earlier (and will shortly prove) that any NDFSM can be replaced by an equivalent FSM. Thus the following theorem should come as no surprise.

**Theorem** (Kleene, 1956). The FSM’s are precisely the acceptors for regular expressions.

We are not about to attempt the proof here but the interested reader can find a more readable account than most in the book by Minsky (section 4.3). Another account of Kleene’s theorem can be found in Hopcroft and Ullman (section 2.5).

**Regular expressions / Finite-state languages.**

We now return to the question: which grammars characterise languages generated by regular expressions (and hence are accepted by FSM’s).

**Definition.** If all production rules of a language are of the form

\[ A \rightarrow tB \]  (Non-terminal on the right) or \[ A \rightarrow t \],

where \( A \) and \( B \) are variables and \( t \) is a (possibly empty) string of terminals, then we say the grammar is right linear.

If all productions are of the form

\[ A \rightarrow Bt \] (Non-terminal on the left) or \[ A \rightarrow t \]

we call the it left linear. A right or left linear grammar is called a regular grammar.

[Note that the definition of a regular expression and that of a regular grammar are quite distinct.]

**Definition.** If a language is generated by a regular grammar we call it a regular language.

**Example.** The language generated by the regular expression \( 0(10)^* \) is also generated by the right linear grammar

\[
V(N) = \{ S, A \} \\
V(T) = \{ 0, 1 \} \\
P = \{ S \rightarrow 0A, A \rightarrow 10A, A \rightarrow \lambda \}
\]

and by the left linear grammar

\[
V(N) = \{ S \} \\
V(T) = \{ 0, 1 \} \\
P = \{ S \rightarrow S10, S \rightarrow 0 \}.
\]

The relationship between left and right linear grammars is rather incestuous. If a language can be generated by one then it can also be generated by the other. More precisely we have:

**Theorem.**

(i) A language is regular if and only if it has a left linear grammar.

(ii) A language is regular if and only if it has a right linear grammar.

The regular grammars are so named because they characterise the languages generated by regular expressions. Thus we have the following theorems (of which the previous theorem is a corollary).
**Theorem.** If a language \( L \) has a regular grammar, then \( L \) is a regular set (i.e. describable by a regular expression).

**Theorem.** If \( L \) is a regular set, then \( L \) is generated by some left linear grammar and by some right linear grammar.

The proofs can be found in Hopcroft and Ullman (Theorems 9.1 and 9.2). Still, for our purpose the important conclusion is that FSM’s are precisely the acceptors for the languages generated by regular grammars.

### Deriving FSM acceptors from regular grammars

Consider the following example.

**Example.** \( L \) is the language of all strings of three symbols of the form  
\[
<\text{decimal digit}>.<\text{decimal digit}>
\]

For example, 4.5 or 7.9. The language may be defined by the following grammar.

\[
\begin{align*}
V(N) &= \{S, A, B\} \\
V(T) &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots\} \\
P &= \{S \rightarrow nA, A \rightarrow .B, B \rightarrow n: 0 \leq n \leq 9\}
\end{align*}
\]

Thus the sequence

\[
S \rightarrow <\text{d digit}>A \rightarrow <\text{d digit}>.B \rightarrow <\text{d digit}>.<\text{d digit}>
\]

leads to the desired string.

Now the above is a right linear grammar and so we can find a FSM which is the acceptor for the corresponding language. This derivation is shown in the state diagrams of Figure 2.3.

In Figure 2.3(a) the only element over and above those found in the defining production rules is a state marked PASS. This signifies that the checking process is ended, i.e. the string is accepted, and occurs whenever a production of the type \( B \rightarrow n \) is encountered.

The resulting state diagram is in fact the direct design for the acceptor shown in Figure 2.3(b). The only modification to the state diagram which is needed is that it must be logically completed. That is, one must lead undesirable steps into a FAIL state. This is shown in Figure 2.3(c), where the output labelling for the states is also shown. The diagram now contains all the information needed to actually build an acceptor, except for the decisions about state assignments.

Normally the construction of a FSM which is the acceptor for a language, defined by a given left or right linear grammar, will not be quite as simple as the last example might suggest. The problem is that the acceptor design derived from the production rules will often be that of a NDFSM. Consider the following example.

**Example.** We modify the production rules of the previous example to be
The partial state diagram which corresponds to this new grammar is shown in Figure 2.4(a). This FSM is non-deterministic, since state B has an exit both to itself and to the END state. What is needed here is an algorithm for transforming a NDFSM into a FSM. A standard way of transforming NDFSM into FSM involves transformations such as shown in Figure 2.4(a) to and Figure 2.4(b).

**From non-deterministic to deterministic**

Consider the grammar

\[ V(N) = \{ S = A, B, C \} \]
\[ V(T) = \{ 0, 1 \} \]
\[ P = \{ A \rightarrow 1A, A \rightarrow 1B, B \rightarrow 0A, B \rightarrow 0C, C \rightarrow 1B, C \rightarrow 1 \} \]

It is clear that any string produced by this grammar is accepted by the NDFSM of Figure 2.5.

[Note: 1*((10)*(01)*0)*101]

The non-deterministic machine in Figure 2.5 is used as an example to illustrate the algorithm for converting a NDFSM to a FSM which we call \( M \).

**Step 1.** From the starting state, \( (S) \), find all the next states for all inputs. In \( M \) allow one state for \( (S) \) and one following state for each input. If the NDFSM goes from \( (S) \) to more than one state for any input, label that state in \( M \) by the names of all these states in the NDFSM (this does happen in the example).

**Step 2.** Repeat Step 1 for all (but only) new states that have been added to \( M \) in Step 1.

**Step 3.** Go back to Step 2 and repeat the procedure of Step 1 for all new states, and so on until no new states can be added to \( M \).

**Step 4.** Label all states in \( M \) that contain a PASS state with the output that indicates a pass.

If the NDFSM has \( N \) states it is not hard to see that the resulting FSM will have no more than \( 2^N \) states. A formal proof that this algorithm terminates, producing an equivalent FSM, proceeds by induction on the length of the input string and can be found in Hopcroft and Ullman (Theorem 2.1). A more readable, but less rigorous account, can be found in Aleksander and Hanna (section 7.2). However if we trace the effect of the algorithm on our particular example it should become clear why the method works.

**Exercise 2.2.** Verify that the string 110101 is accepted by the NDFSM illustrated in Figure 2.5. Construct a grammar which the NDFSM models. Prove that no string beginning 1 n 0 ( \( n > 0 \) ) will be accepted.

**Partial solution.** It is easy to see that the string 110101 drives the NDFSM through the states \( A \rightarrow A \rightarrow B \rightarrow (A \text{ or } C) \rightarrow B \rightarrow C \rightarrow X \). Hence the string is accepted.
The grammar \( G = \langle V(N), V(T), P, S, > \), where

\[
V(N) = \{ S = A, B, C \} \\
V(T) = \{ 0, 1 \} \\
P = \{ A \rightarrow 1A, A \rightarrow 1B, B \rightarrow 0A, B \rightarrow 0C, C \rightarrow 1B, C \rightarrow 1 \},
\]

has a flow graph identical to the NDFSM of the Figure 2.5

With internal states \( Q = \{ A, B, C \} \) the subsets of \( Q \) are

\[
\emptyset, \{ A \}, \{ B \}, \{ C \}, \{ A, B \}, \{ A, C \}, \{ B, C \}, \{ A, B, C \}
\]

and there are \( 2^3 \), namely 8, possible subsets. In the example \( Q = \{ A, B, C, X \} \) and so there are 16 possible subsets, but some of these are never reached (e.g. \( \{ A, C, X \} \), or \( \{ B \} \)).

The crucial step in the construction of the equivalent FSM is to let the states of \( M \), the FSM we are trying to construct, correspond to subsets of the states of the NDFSM. This is motivated by the observation that, in the NDFSM at a given state, the set of next states for a given input can then be mapped to the corresponding single state in \( M \). Thus if the NDFSM has \( N \) states, the equivalent FSM can have at most \( 2^N \) states.

**Step 1.** In the present example

\( S = A \) and \( (A, 0) \rightarrow \emptyset, (A, 1) \rightarrow \{ A, B \} \)

so we begin by creating states called \( \emptyset \) and \( \{ A, B \} \) for \( M \).

**Step 2.** In the previous step we added 2 states. So repeating the process for these states we find

\[
(\{ \}, 0) \rightarrow \{ \} \text{ and } (\{ \}, 1) \rightarrow \{ \}
\]

\[
(\{ A, B \}, 0) \rightarrow (\text{Image } (A, 0)) \cup (\text{Image } (B, 0)) \rightarrow \{ \} \cup \{ A, C \} = \{ A, C \}
\]

\[
(\{ A, B \}, 1) \rightarrow (\text{Image } (A, 1)) \cup (\text{Image } (B, 1)) \rightarrow \{ A, B \} \cup \{ \} = \{ A, B \}.
\]

Thus at this stage we add a state labelled \( \{ A, C \} \) to \( M \).

**Step 3.** The last state added was \( \{ A, C \} \) and

\[
(\{ A, C \}, 0) \rightarrow (\text{Image } (A, 0)) \cup (\text{Image } (C, 0)) \rightarrow \{ \} \cup \{ \} = \{ \}
\]

\[
(\{ A, C \}, 1) \rightarrow (\text{Image } (A, 1)) \cup (\text{Image } (C, 1)) \rightarrow \{ A, B \} \cup \{ B, X \} = \{ A, B, X \}.
\]

Thus at this stage we add a state labelled \( \{ A, B, X \} \) to \( M \).

**Step 3.1.** The last state added was \( \{ A, B, X \} \) and

\[
(\{ A, B, X \}, 0) \rightarrow (\text{Image } (A, 0)) \cup (\text{Image } (B, 0)) \cup (\text{Image } (X, 0)) \rightarrow \{ \} \cup \{ A, C \} \cup \{ \} = \{ A, C \}.
\]

\[
(\{ A, B, X \}, 1) \rightarrow (\text{Image } (A, 1)) \cup (\text{Image } (B, 1)) \cup (\text{Image } (X, 1)) \rightarrow \{ A, B \} \cup \{ \} \cup \{ \} = \{ A, B \}.
\]
The algorithm now terminates, since no new states have been added.

Exercise 2.3. Draw the state diagram for this deterministic FSM (see Figure 2.6).

[Note: 1(1*0(10)*11)*01]

Finally, the NDFSM accepts a string if it is able to finish in one of the accept states. The corresponding condition for the equivalent deterministic FSM is that it must finish in a state whose label contains one of the accept states. In the above example the only accept state of the NDFSM was $X$ and so the only accept state of the equivalent deterministic FSM is $\{A, B, X\}$.

Other properties of regular expressions.

Another useful set of properties possessed by regular expressions concerns their closure under Boolean operations. The term closure is used here in a precise technical sense.

Definition. We say a set $R$ is closed under some binary operation $\odot$ if, for any pair $x, y$ of elements in $R$, $x \odot y$ is also in $R$.

Example. The set of even integers is closed under addition, that is the sum of any pair of even integers is also an even integer. However the set of odd integers is not closed under addition since $3 + 5$ is $8$, which is not odd.

Theorem. The regular sets are closed under complement. That is if $L$ is a regular set and $L^c$ is the set of strings accepted by a particular FSM, then the complement of $L$ (i.e. the set of strings in $V^*$ but not in $L$) will necessarily be accepted by some FSM.

Proof. Here we are stating that if $V$ is an alphabet (or vocabulary) and $L$ is the set of strings accepted by a particular FSM, then the complement of $L$ (i.e. the set of strings in $V^*$ but not in $L$) will necessarily be accepted by some FSM. Suppose

$$M = \langle I, Z, Q, d, w \rangle$$

is the FSM which accepts $L$.

We modify $M$ by first exchanging the set of acceptor states for the set of non-acceptor states. Secondly, if necessary, we augment the input set $I$ of $M$ to allow the full alphabet $V$. Finally, realising that any string containing one of the previously unaccepted symbols from $V - I$ must now be unconditionally accepted, we add a new state, $i$, to the set...
of states \( Q \), and extend \( d \) to `trap' any such string. The result is the required FSM.

The above pairs of theorems lead to the following results.

**Theorem.** The regular sets are closed under all Boolean operations.

**Theorem.** The class of languages accepted by FSM’s is closed under all Boolean operations.

**Proof** (of either).

This follows immediately since any Boolean function can be expressed in terms of union and complement. For example

\[
L(1) \text{ AND } L(2) = \neg(\neg L(1) \cup \neg L(2))
\]

\[
L(1) \text{ EOR } L(2) = \neg(\neg L(1) \cup L(2)) \cup \neg(L(1) \cup \neg L(2)).
\]

There are other closure properties of regular sets which are of some interest. For example the operator \( * \) (`any number of times')acting on an alphabet \( V \) can be regarded as a kind of `set-closure' operator. Indeed it is plain from axiom (iv) for regular expressions that the class of regular sets is closed under the operator \( * \). This leads to the following equivalent result for FSM’s.

**Theorem.** The class of languages accepted by FSM’s is closed under the operator \( * \).

**Exercise 2.4.** Construct an FSM which accepts strings of the form

\[0^{2n+1}1\]

and no others, call this the language \( L \). Illustrate by examples the language \( L^* \). Construct a FSM acceptor for \( L^* \).

**Solution.** See the questions and solutions Appendix or p124 Aleksander and Hanna.

It is important to have algorithms to answer various questions concerning regular sets. The type of question we may be concerned with include: is a given language empty, finite or infinite? Is one regular set equivalent to another? And so on. The reader may feel at this stage that it is obvious that we can determine whether a regular set is empty. However, for many interesting classes of languages the question cannot be answered (see Hopcroft and Ullman Chapter 8). Before discussing this and related questions we shall first introduce a useful theoretical tool.

**The Pumping lemma.**

In this section we prove a basic result, called the *Pumping Lemma*, which is a powerful tool for proving certain languages non-regular. It is also useful in the development of algorithms to answer certain questions concerning finite automata, such as whether the language accepted by a given FSM is finite or infinite.

If a language is regular, it is accepted by a deterministic FSM \( M \) with states \( Q \), say \( n \) in number, and initial state \( q(0) \). Let \( D \)
be the function which maps to the resulting state $q'$ when $M$ is supplied with a start state $q$ and an input string $a(1)a(2)...a(i)$, so $D(q(0), a(1)a(2)...a(i)) = q'$.

Now consider an input of $n$ or more symbols $a(1)a(2)...a(m)$, where $m \geq n$, and for $i = 1, 2, ..., m$ let $D(q(0), a(1)a(2)...a(i)) = q(i)$. It is not possible for each of the $n + 1$ states $q(0), q(1), ..., q(n)$ to be distinct, since there are only $n$ different states. Thus there are two integers $j$ and $k$, $0 \leq j < k \leq n$, such that $q(j) = q(k)$. Since $j < k$, the string $a(j+1)...a(k)$ is of length at least 1, and since $k \leq n$, its length is no more than $n$, see Figure 2.7.

If $q(m)$ is an accept state of $M$, i.e. $a(1)a(2)...a(m) \in L(M)$, then $a(1)a(2)...a(j)a(k+1)a(k+2)...a(m)$ is also in $L(M)$, since this string is constructed by omitting the loop labelled $a(j+1)...a(k)$ from $a(1)a(2)...a(m)$. Formally

$$D(q(0), a(1)...a(j)a(k+1)...a(m)) = D(D(q(0), a(1)...a(j)), a(k+1)...a(m)) = D(q(j), a(k+1)...a(m)) = D(q(k), a(k+1)...a(m)) = q(m)$$

Thus we can either not go around the loop at all or we can go round it once, either way the string is accepted. Similarly, we could go around the loop more than once, in fact as many times as we like. Thus

$$a(1)...a(j) (a(j+1)...a(k))^i a(k+1)...a(m) \in L(M) \forall i \geq 0.$$

What we have proved is that given any sufficiently long string accepted by a FSM, we can find a substring (within the first $n+1$ symbols) that may be 'pumped', i.e. repeated as many times as we like, and the resulting string will still be accepted by the FSM. The formal statement of this result is as follows.

**Lemma (Pumping lemma for regular sets).** Let $L$ be a regular set. Then there is a constant $n$ such that if $z$ is any word in $L$ and $l(z) \geq n$, we may write $z = uvw$ in such a way that $l(uv) \leq n$, $l(v) \geq 1$, and for all $i \geq 0$, $uv^i w$ is in $L$. Moreover, $n$ is no greater than the number of states in the smallest FSM accepting $L$.

**Proof.** In the above discussion take

$$z = a(1)a(2)...a(m)$$
$$u = a(1)a(2)...a(j)$$
$$v = a(j+1)...a(k)$$
$$w = a(k+1)...a(m).$$

Example. Consider the language $L$ of well formed brackets. Strings of $L$ are of the form $a^k b^k$, $k > 0$. Now apply the above lemma. If $L$ is regular then we may choose $m > n$ so that

$$z = a^m b^m \in L.$$

By the lemma there exist sub-strings $u, v, w$ of $z$ with the properties stated. There are three cases.

(i) $v$ is composed entirely of $a$'s. In this case taking $i = 2$ the lemma asserts that $uv^2w$ is in $L$. This is false since we have increased the number of $a$'s in the string without increasing the number of $b$'s by the same number.

(ii) $v$ is composed entirely of $b$'s. This case can be dealt with similarly to (i), and again leads to a contradiction.

(iii) $v$ is composed of both $a$'s and $b$'s, i.e. $v$ bridges the interface between $a$'s and $b$'s. Again apply the lemma with $i = 2$. Now the string $uvvw$ has (in $vv$) at least one $a$ following a $b$, so again we have a contradiction.
Theorem. The set of strings accepted by a FSM is not regular. Another application of the pumping lemma for regular sets is the proof of the following theorem.

Solution. Assume $L$ is regular and let $n$ be the integer in the pumping lemma. Let $s = n^2$ and $z = 0^s$. By the pumping lemma, $0^s$ may be written as $uvw$, where $1 \leq l(v) \leq n$ and $uv^i w$ is in $L$ for all $i$. In particular for $i = 2$. However,

$$n^2 = l(uvw) < l(uvw) = l(uvw) + l(v) \leq n + n < (n+1)(n+1).$$

Thus $l(uvw)$ cannot be a perfect square and so $uvw$ is not in $L$, a contradiction. Hence $L$ is not regular. ■

An application of the Pumping lemma.

Another application of the pumping lemma for regular sets is the proof of the following theorem.

Theorem. The set of strings accepted by a FSM $M$ with $n$ states is:

1. Non-empty if and only if the FSM accepts a string of length less than $n$.
2. Infinite if and only if the FSM accepts some sentence of length $l$ where $n \leq l < 2n$.

Proof. (1) The ’if’ part is obvious. Suppose $M$ accepts a non-empty set. Let $w$ be a word as short as any other word accepted. By the pumping lemma, $l(w) < n$, for if $w$ were a shortest word and $l(w) \geq n$, then $w = uvy$, and $uy$ is a shorter word in the language.

(2) If $w$ is in $L(M)$ and $n \leq l(w) < 2n$, then by the Pumping lemma, $L(M)$ is infinite. That is, $w = w(1)w(2)w(3)$, and for all $i$, $w(1)w(2)^i w(3)$ is in $L$. Conversely if $L(M)$ is infinite, then there exists $w$ in $L(M)$, where $l(w) \geq n$. If $l(w) < 2n$ we are done. If no string is of length between $n$ and $2n - 1$, let $w$ be of length at least $2n$, but as short as any string in $L(M)$ whose length is $\geq 2n$. Again by the Pumping lemma, we can write $w = w(1)w(2)w(3)$ with $1 \leq l(w(2)) < n$ and $w(1)w(3)$ in $L(M)$. Either $w$ was not a shortest word of length $2n$ or more, or $l(w(1)w(3))$ is between $n$ and $2n - 1$, a contradiction in either case. ■

Thus from part (1) an algorithm to determine whether $L(M)$ is empty is: see if any string of length up to $N$ is in $L(M)$. Clearly there is such a procedure that is guaranteed to halt. In part (2), the algorithm to decide whether or not $L(M)$ is infinite is: see if any string of length between $N$ and $2N - 1$ is in $L(M)$. Again, clearly there is such a procedure that is guaranteed to halt. It should be noted that the algorithms suggested by the theorem are highly inefficient, but this form of the result is easy to prove.

An easier way to check whether an FSM accepts only the empty set is to take its state diagram and delete all states that are not reachable on any input from the start state. If one or more states remain, the language is non-empty. Then, without changing the language accepted, we may delete all states that are not final and from which one cannot reach a final state. The FSM accepts an infinite language if and only if the resulting state diagram has a cycle.

Theorem. There is an algorithm to determine if two FSM’s accept the same set.

Proof. Let $M(1)$ and $M(2)$ be two FSM’s which accept $L(1)$ and $L(2)$ respectively. Since the class of languages accepted by FSM’s is closed under Boolean operations there is an FSM, $M$ say, which accepts

$$(L(1) \cap \neg L(2)) \cup (\neg L(1) \cap L(2))$$

Theorem. There is an algorithm to determine if two FSM’s accept the same set.
It is easy to see that $M$ accepts some string, i.e. $L(M)$ is not empty, if and only if $L(1) = L(2)$. Thus by the last theorem there is an algorithm to determine if $L(1) = L(2)$.

**Applications of FSM's in software.**

The notions of FSM's and regular expressions were originally developed with neural nets and switching circuits in mind. Later they served as useful tools in the design of lexical analysers, the part of a compiler that groups characters into tokens (indivisible units such as variable names and keywords), see the aside earlier concerning grep under UNIX. A number of compiler-compilers automatically transform regular expressions into FSM's for use as lexical analysers.

Certain text editors and similar programs permit the substitution of a string for any string matching a given regular expression. For example the UNIX text editor `ed` allows commands such as

```
:s/*/*/l
```

(the first string is three spaces followed by * and is a regular expression, the second string is a single space) that substitutes a single space for the first string of two or more spaces found in a given line. Define the regular expression

$$A# = a(1) + a(2) + \ldots + a(n)$$

where the $a(i)$'s are all of a standard character set except 'newline'. We could convert a regular expression $R$ to a FSM which accepts $(A#)^* R$. Note the presence of $(A#)^*$ allows us to recognise a member of $L(R)$ beginning anywhere in the line. However the conversion of a regular expression to a FSM takes far more time than it takes to scan a single short line using the FSM, and the FSM could have a number of states that is an exponential function of the length of the regular expression.

What actually happens in `ed` is that the regular expression $(A#)^* R$ is converted to a NDFSM and this is then simulated directly by exploring all possible paths generated by the input.

**Chapter references**


**Exercises for Chapter 2**

1.(i) Define the term *Finite State Machine* (FSM). Figure 2.8 is the state diagram of a simple FSM. Write down the set of states, input and output alphabets, and defining functions for this machine and give an informal description of its operation.

[Description of Figure 2.8. States q(0)]
(initial), q(1), q(2) Arcs q(0) to q(0) i=1, o=0 q(0) to q(1) i=0, o=1 q(1) to q(0) i=1, o=0 q(1) to q(2) i=0, o=0 also i=1, o=0 ]

(ii) Define the term 'regular expression' and explain the relationship between regular expressions and FSM's.

(iii) Construct a state diagram for an FSM that accepts any string of the alphabet $a$, $b$, $c$ that either begins with $a$ or $b$ or, if it begins with $c$, contains not more than one $a$. (Note: indicate the FAIL state, and for simplicity assume that when the last input symbol is read, if the current state is not FAIL then the string will be accepted.) What is the regular expression which describes the set of strings accepted by this FSM?

2(i) Define the terms Deterministic Finite State Machine (DFSM) and Non-deterministic Finite State Machine (NDFSM). What feature in an FSM state diagram characterises a NDFSM?

(ii) The NDFSM in Figure 2.9 accepts an input string of $0$'s or $1$'s if, given the string, there is at least one sequence of possible moves that leaves the machine in state q(1). Construct an equivalent DFSM, indicating the accept states and assuming for simplicity that if the machine is not in an accept state when the last symbol is read then the string is rejected.

Write down a regular expression which describes the set of strings which this automaton accepts.

(Note: In Figure 2.9 edges are labelled with inputs only and the initial state is assumed to be q(0).)

[Description of Figure 2.9. States q(0) (initial), q(1) (accept state). Arcs
From To Labelled/Input
q(0) q(0) 0
q(0) q(1) 1
q(1) q(0) 0+1
q(1) q(1) 0
]

Figure 2-9 Figure for Q2.2.

3. Define the term 'regular expression' and explain the relationship between regular expressions and Finite State Machines (FSMs).

For each of the three cases listed below design an acceptor for input strings composed of $0$'s and $1$'s. Give the FSM state diagram and clearly indicate the 'Accept' states. (You may include a specific FAIL state if this seems appropriate.) Also write down an equivalent regular expression.

(a) The number of $1$'s in the input string is divisible by three.

(b) All $1$'s in the input string are in blocks of at least three.

(c) The $3n$ th (all $n \geq 1$) and the $3n + 2$ th (all $n \geq 0$) characters in the input string are not $1$'s.

4. Define the term 'Finite State Machine' (FSM) and briefly describe what is meant by a 'regular expression'. Explain the relationship between FSM acceptors and regular expressions.

Which of the following strings composed of $0$'s and $1$'s, can be recognised by an FSM acceptor. In each case give a few words of explanation or the relevant regular expression.

(a) The set of all non-empty strings of $0$'s and $1$'s, $0$, $1$, $00$, $01$, $11$, $000$, etc.

(b) The numbers $1$, $2$, $4$, $8$, ..., $2^n$ ...etc, written in binary notation. (Input the most significant digit first.)
(c) The same set as in (b) written in unary 1, 11, 111, etc.
(d) The set of strings in which the number of 0’s is equal to the number of 1’s.
(e) The strings 0, 101, 1101, ..., 1^n0^n, etc.
(f) If \( E \) is the set of strings recognised by an FSM \( M \), is there another FSM that recognises any string ending with one that \( M \) recognises?
(g) If \( E \) is the set of strings recognised by an FSM \( M \), is there another FSM that recognises any string containing one that \( M \) recognises?
(h) If \( E \) is the set of strings recognised by an FSM \( M \), is there another FSM that recognises any two consecutive occurrences of the same string of \( E \)?

5. Define the term ‘Grammar’ as used in automata theory. In particular explain the expressions ‘left-linear’ and ‘right-linear’ as applied to grammars. What is the relationship between left and right linear grammars and FSM acceptors?

Give regular expressions which describe the sets of strings generated by each of the following two grammars, and give a state diagram for an acceptor in each case.

(a) \( S \rightarrow 0S, S \rightarrow 1A, A \rightarrow 1A, A \rightarrow 0S, S \rightarrow \lambda (\lambda = \text{empty string}) \).

(b) \( S \rightarrow bA, A \rightarrow bB, B \rightarrow bB, B \rightarrow aS, B \rightarrow a \)

6. State and prove the Pumping Lemma for regular sets. Hence or otherwise show that the set

\[ \{ 0^p : \text{where } p \text{ varies over all primes} \} \]

is not a regular set.

7. State the Pumping Lemma for regular sets. Hence or otherwise prove the following theorem.

**Theorem.** The set of strings accepted by a FSM \( M \) with \( n \) states is

1. Non-empty if and only if the FSM accepts a string of length less than \( n \).
2. Infinite if and only if the FSM accepts some sentence of length \( l \) where \( n \leq l < 2n \).

Briefly discuss a simple but more efficient method of deciding if a FSM

(a) Accepts only the empty set.
(b) Accepts an infinite set of strings.

8. Define the term ‘regular expression’ and explain the relationship between regular expressions and FSM’s. Deduce that the class of languages accepted by FSM’s is closed under the operator \(*\).

Construct a FSM which accepts only strings of the form \( 2n+1 \) 0’s followed by a single 1, where \( n \geq 0 \). Call this the language \( L \). Illustrate by examples the language \( L^* \). Construct a FSM acceptor for \( L^* \).
Introduction.

The power of FSM’s is limited by their finiteness. Thus we may design an FSM to multiply two specific numbers, or any two numbers up to some pre-specified size, but not to multiply ANY two numbers. To overcome this restriction, in the first instance we augment the FSM with an additional memory unit having unlimited capacity - the stack. The resulting device is called a Push Down Stack automata. In practice, of course, such an automaton is unrealisable but nevertheless it is helpful to consider the abstract notion.

Just as FSM’s have their linguistic counterpart, the regular languages, so Push Down Stack automata have their linguistic parallel in Context-free languages. Here the parallel is somewhat less satisfactory since deterministic and non-deterministic Stack automata turn out not to be equivalent and the Context-free languages correspond to the non-deterministic Stack automata.

Context-free languages

Consider a grammar \( G = \langle V(N), V(T), P, S \rangle \) where

\[
\begin{align*}
V(N) & = \{ S \}, \\
V(T) & = \{ a, b \}, \\
P & = \{ S \rightarrow SS, S \rightarrow aSb, S \rightarrow ab \}.
\end{align*}
\]

This is a modification of the language \( a^n b^n \) which we considered earlier, the language of well nested brackets. With the extra production \( S \rightarrow SS \) this becomes the language of well formed brackets. For example

\[ (((()))))()()() \]

is a valid string in this language, and compilers frequently need to check such expressions.

First, note that this is not a regular grammar, since the productions are not of the required type. Moreover the language it represents is not regular, since the production \( S \rightarrow aSb \) is a self embedding property (recursion) and this is not characteristic of regular languages. Informally it is easy to see why we cannot design a FSM which accepts this language. The acceptance of the string \( a^n b^n \) can only occur if the FSM keeps track of the number of times the symbol \( a \) has occurred so that this can later be compared with the number of times the symbol \( b \) occurs. Plainly if the FSM has \( N \) states, then by making \( n \) sufficiently large we can find a string which the FSM cannot be guaranteed to check successfully. We shall shortly consider in some detail an automata type, the Stack automata, which does have the required property. Even so, we shall have to suppose that such an automaton has at its disposal an arbitrarily large memory called a ’stack’.

Definition. A grammar \( G = \langle V(N), V(T), P, S \rangle \) is context free if all productions in \( P \) are of the form \( A \rightarrow x \), where \( A \) is a variable and \( x \) a string of symbols from \( (V(N) \cup V(T))^* \).

Note that the context free grammars include the regular grammars, since the productions of a right or left linear grammar \( (A \rightarrow tB \text{ or } Bt, \text{ and } A \rightarrow t) \) are included in the above.

The language defined by a grammar can be defined as the set of strings \( L(G) \) such that:

(i) The string consists solely of terminals,

and

(ii) The string can be derived from \( S \) using the production rules of the grammar.
We call \( L \) a context free language if it is \( L(G) \) for some context free grammar. We can define two grammars \( G(1) \) and \( G(2) \) to be equivalent if \( L(G(1)) = L(G(2)) \).

The language of well formed brackets is easily seen to be a context free language, another example is given below.

**Example.** Define \( G = \langle \{ E \}, \{ +, \cdot, (, ) \}, P, E > \), where \( P \) consists of

\[
E \rightarrow E + E \\
E \rightarrow E \cdot E \\
E \rightarrow (E) \\
E \rightarrow id
\]

If we interpret \( id \) to be any arithmetic variable (chosen from some finite set), then it can be quickly verified that \( G \) generates the language of arithmetic expressions.

While linguists were studying context free grammars, computer scientists began to study programming languages by a notation called Backus-Naur Form (BNF), which is the context free grammar notation with minor changes in format and some shorthand. The use of context free grammars has greatly simplified the definition of programming languages and the construction of compilers. The reason for this is undoubtedly due in part to the natural way most programming constructs are described by grammars.

### Stack automata

The idea of a stack automaton is that it is a FSM with an additional feature, an arbitrarily large memory organised in a particular way. We say ‘arbitrarily large’ rather than ‘infinite’ so as to imply that we can still, in some sense, actually construct such an automaton. We can imagine starting with some amount of store and, whenever the need arises, just adding more and more. Thus, at any point in time, only a finite amount of store would be deployed.

In the first method we shall study the store will be configured as a ‘push down stack’. Readers familiar with the structure of contemporary computers will already be familiar with this type of organisation. A push down stack is organised on a ‘last in first out’ basis. The operation of this system is analogous to the ‘push down’ stacks of plates said (mostly by computer scientists) to be found in some cafeterias. When a plate is put on the top of the stack all the remaining plates move down. At any given time only the top plate is accessible. When this plate is removed all the remaining plates move up, and a new top plate is exposed.

We shall call a device using this type of store a push down automaton (PDSA).

Before giving a formal definition let us consider an example.

**Example.** To illustrate such a machine let us consider the language of well formed brackets described above. The ploy adopted is to place all opening brackets onto the stack as they occur and to remove them as the closing brackets occur. As the string

\(((()())())\)
is processed the stack develops as shown.

```
Input      Stack content
0 (empty stack)
(          (          
(          (          
(          (          
)          )          
0          )          
)          )          
)          )          
)          )          
)          )          
)          )          
)          )          
)
```

When an acceptable string has been seen at the input the top of stack is 0. What would the FSM part of Figure 3.1 look like in this case? Firstly, we note that the FSM takes a pair of input symbols at each step, one being the input to the PDSA and the other is the top-of-stack symbol. Thus the set $I$ of input messages for this FSM is

$$
I = \{[\lambda, 0], [\lambda, (], [\lambda, )], [(, 0], [), 0], [(, (], [), (], [(, )], [), )]\}
$$

where the left symbol of a pair $[ , ]$ is the incoming symbol ($\lambda$ is the empty string) and the right symbol is the top-of-stack. We can label these combinations $i_1, i_2, ..., i_9$ respectively. The output of the FSM affects the stack and also signals pass or fail according to the following table.

<table>
<thead>
<tr>
<th>Input</th>
<th>Next stack action</th>
<th>New tos</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_1$: [e,0]</td>
<td>none</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>$i_2$: [e,()]</td>
<td>ignore</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$i_3$: [e,)]</td>
<td>won't occur</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$i_4$: [(,0]</td>
<td>push (</td>
<td>(</td>
<td>intermediate</td>
</tr>
<tr>
<td>$i_5$: [,0]</td>
<td>none</td>
<td>-</td>
<td>fail</td>
</tr>
<tr>
<td>$i_6$: [,()]</td>
<td>push (</td>
<td>(</td>
<td>intermediate</td>
</tr>
<tr>
<td>$i_7$: [,)]</td>
<td>pop (</td>
<td>(</td>
<td>intermediate</td>
</tr>
<tr>
<td>$i_8$: [,)]</td>
<td>pop (</td>
<td>0</td>
<td>intermediate (*)</td>
</tr>
<tr>
<td>$i_9$: [,)]</td>
<td>won't occur</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$i_{10}$: [,)]</td>
<td>won't occur</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We could interpret input $i_2$ in the intermediate state to mean that the string has terminated whilst the stack is non-empty and therefore the string fails. Better still we can just ignore this possibility and reserve the empty string $\lambda$ for autonomous moves. After (*) the machine makes an autonomous move, specified using $i_1$, from the intermediate to the pass state.

Inputs $i_3, i_8$ and $i_9$ turn out to be impossible because the symbol ‘)’ can never get onto the stack. Figure 3.2 defines the operation of this automaton.

![Figure 3-2 Operation of the FSM controller.](image)
We see that it has three states $q(0)$ (pass), $q(1)$ (intermediate - the string doesn’t pass at this stage but it still might if there is further input), and $q(2)$ (fail irrecoverably), and the ability to push down an ‘∈’ onto the stack, or to pop up the stack. It accepts a string if, and only if, when the last input symbol of a string has been read in, the stack is left empty.

**Definition of Stack Automata.**

Let us now try to construct a formal definition which will cover this class of automata.

Firstly, it should be noted that the last example was of a deterministic PDSA. With FSM’s the distinction between deterministic and non-deterministic was seen to be theoretically insignificant, since any NDFSM is equivalent to a (deterministic) FSM. However, with PDSA’s this turns out not to be the case, NDPDSA’s are genuinely more powerful than deterministic PDSA’s.

Secondly, in our example the stack was manipulated once independently of the next input symbol - the move corresponding to $\{\cdot, 0\}$. In general we shall allow symbols to be pushed down onto the stack or popped up, together with state transitions, without an input symbol necessarily being read in. We call this kind of move an autonomous one.

Thirdly, we shall allow not just one symbol to be pushed onto the stack at a time, but a string of symbols. In fact, this last change does not increase the power of PDA’s, any PDA of this type is equivalent to some PDA of the more primitive type, but it does make it much easier to describe their operation.

**Definition.** A NDPDSA is a 6-tuple $<Q, I, J, q(0), j(0), d>$ where

- $Q$ is a finite set of states
- $I$ is a finite set of input symbols
- $J$ is a finite set of stack symbols
- $q(0)$ in $Q$ is an initial state
- $j(0)$ in $J$ is an initial stack symbol
- $d$ is a function which maps $Q \times (I \cup \{\cdot\}) \times J \rightarrow \mathcal{P}(Q \times J^*)$, where $\mathcal{P}(X)$ denotes the set of all subsets of $X$.

The first five items of this definition are simple enough, however note that we assume the stack initially contains only the symbol $j(0)$. The definition of the function $d$ requires some explanation. The domain of the mapping consists of triples $(q(t), i(t), j(t))$, where $q(t)$ is the current state, $i(t)$ is an input symbol (which may be the null symbol $\cdot$), and $j(t)$ is the current symbol on top of the stack. The range of $d$ is the set of all subsets of $Q \times J^*$. Recall that $J^*$ is the set of all finite strings composed of symbols from $J$. Thus an element $(q(t+1), x)$ of $Q \times J^*$ can describe the resulting state $q(t+1)$, and the new string $x$ on ‘top’ of the stack. We can imagine that the original symbol $j(t)$ is popped from the stack and the whole string $x$ is then pushed onto it. The new top of stack symbol $j(t+1)$ then becomes the leftmost symbol of the string $x$.

The fact that the empty string $\cdot$ is allowed as an input symbol permits the automaton to move under the current state without reference to the current input $i(t)$ - an autonomous move of the form

$$d: (q, \cdot, j) \rightarrow (q', x).$$

Here, if $x$ is the empty string the symbol below rises to the top of the stack. When an autonomous move, or a sequence of autonomous moves, is completed the input $i(t)$ is then processed (it is not lost).

The ‘non-deterministic’ transitions are catered for in exactly the same way as for FSM’s, namely by replacing $Q \times J^*$ by the set of all subsets of $Q \times J^*$. 
We shall say a NDPDSA *accepts* an input string if, starting in its initial condition, there is some sequence of transitions which enable it to read in the entire string and finish with its stack empty.

**NDPDSA’s and (deterministic)PDA’s are not equivalent**

Unlike FSM’s, non-deterministic PDA’s and deterministic PDA’s are not equivalent as the following example illustrates.

Consider the language $L$ defined over the terminal symbols $\{0, 1\}$ such that
\[
L = \{ z : z = w[w] , w \in (0 + 1)^* \}
\]
where $[w]$ denotes the string $w$ written backwards, e.g. if $w = 0011$ then $[w] = 1100$. This is the language of (even length) *palindromes* over the terminal symbols $\{0, 1\}$.

A grammar which generates $L$ is
\[
S \rightarrow 0S0, S \rightarrow 1S1, S \rightarrow 00, S \rightarrow 11.
\]
the first two productions may be iterated any (finite) number of times to produce a centrally symmetric string, finally one of the last two productions must be used to eliminate $S$ whilst preserving central symmetry.

The grammar is a Context Free grammar since every production is of the form $A \rightarrow x$, where $A$ is a non-terminal symbol and $x$ is a string of terminals and non-terminals (this is the definition of a CFG). Consequently $L$ is a Context Free language and can therefore be accepted by some NDPDSA by a theorem of Chomsky which we shall shortly discuss.

To see why the extra power of a NDPDSA is actually necessary to parse such strings we begin by making two critical observations. Firstly, the automaton must read the input string in one pass. It cannot first count the length of the input string, from which the ‘centre’ point could be inferred, and then, on a second pass, push the first half of the string onto the stack and unstack this string symbol by symbol as the second half of the input string is read. Secondly, the stack is of arbitrary depth, whilst the FSM controller of the PDSA has a fixed number of states. Therefore the algorithm to parse a string must NOT require that the FSM absorb arbitrary amounts of the stack contents at any stage.

Plainly the essence of successful parsing is to locate the centre of the input string. However the second observation above makes this impossible for the DPDSA. The NDPDSA can accomplish this because we only require that there is SOME sequence of choices of autonomous moves which will lead to an empty stack when the end of the input string is reached. Consequently at each stage the NDPDSA can be granted the (non-deterministic) alternatives of EITHER assuming the string centre has been reached and then proceeding to unstack on this assumption, OR assuming that the centre has not been reached and then continuing to the next input symbol. Because of non-determinism we are not required to give an explicit mechanism for recovery in the event that the wrong choice is made, this is implicit in the definition of a NDPDSA which accepts by empty stack.

**Context free grammars and NDPDSA’s**

Let us further illustrate the idea of a NDPDSA by designing one which can act as an acceptor for the context free language generated by the grammar $G$ defined as
\[
G = <\{E, T, P\}, \{a, b, c, \ldots, (, )\}, Prod, E>
\]
where $Prod$ consists of
\[
E \rightarrow E + T, E \rightarrow T, T \rightarrow T.P, T \rightarrow P, P \rightarrow (E), P \rightarrow a, P \rightarrow b, P \rightarrow c.
\]
We can identify these productions as defining a very simplified 'formula language' (for arithmetic expressions involving three variables, multiplication and addition) in the following way.

Expression - Expression + Term, Expression - Term, Term - Term . Primary, Term - Primary, Primary - ( Expression ), Primary - a, or b, or c.

For example, a typical string, and its derivation, might be

\[ E = T - T.P - P.P - b.P - \ldots - b.(a + c) \]

Let us show how to design a NDPDSA which will accept \( L(G) \), and thus have the ability to parse a formula for syntactic correctness.

Firstly, we consider the set of states. It turns out that this non-deterministic PDSA will need only one state \( q(0) \), the initial state. The set of input symbols \( I \) corresponds to the terminal vocabulary of the grammar

\[ I = V(T) = \{ a, b, c, +, \ldots \} \].

The set of stack symbols corresponds to the total vocabulary \( J = V(T) \cup V(N) \).

From this set the symbol on top of the stack, \( j(0) \), is chosen to be the start symbol of the grammar, thus \( j(0) = E \).

Finally, we must construct the function \( d \). We shall explain the construction of a few typical transformations and leave it to the reader to verify the remainder and, if desired, to provide a formal description of the process. The complete result is given in the table below.

<table>
<thead>
<tr>
<th>Input</th>
<th>Mapping used</th>
<th>Stack</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, E) \rightarrow (q(0), T) )</td>
<td>( E )</td>
<td>Initial contents</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, T) \rightarrow (q(0), T.P) )</td>
<td>( T.P )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, T) \rightarrow (q(0), P) )</td>
<td>( P.P )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( b )</td>
<td>( (q(0), b, P) \rightarrow (q(0), \lambda) )</td>
<td>( .P )</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( . )</td>
<td>( (q(0), \ldots) \rightarrow (q(0), \lambda) )</td>
<td>( P )</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( ( )</td>
<td>( (q(0), \ldots) \rightarrow (q(0), (P)) )</td>
<td>( E )</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, E) \rightarrow (q(0), E+T) )</td>
<td>( E+T )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, E) \rightarrow (q(0), T) )</td>
<td>( T+T )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, T) \rightarrow (q(0), P) )</td>
<td>( P+T )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( a )</td>
<td>( (q(0), a, P) \rightarrow (q(0), \lambda) )</td>
<td>( +T )</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( + )</td>
<td>( (q(0), +, \ldots) \rightarrow (q(0), \lambda) )</td>
<td>( T )</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( (q(0), \lambda, T) \rightarrow (q(0), P) )</td>
<td>( P )</td>
<td>Autonomous move</td>
</tr>
<tr>
<td>( c )</td>
<td>( (q(0), c, P) \rightarrow (q(0), \lambda) )</td>
<td>)</td>
<td>Symbol accepted</td>
</tr>
<tr>
<td>( ) )</td>
<td>( (q(0), \lambda) \rightarrow (q(0), \lambda) )</td>
<td>)</td>
<td>Symbol accepted</td>
</tr>
</tbody>
</table>

**Design for a NDPDSA acceptor.**

Corresponding to the pair of productions \( E - E + T \) and \( E - T \) we have the mapping

\[ (q(0), \lambda, E) \rightarrow \{(q(0), E + T), (q(0), T)\} \].

The LHS indicates ’when the PDA is in state \( q(0) \), and has the symbol \( E \) on top of the stack, then, without taking an input symbol (i.e. taking the null string \( \lambda \)), the following transitions may be made’. Note the non-determinism, the transition does not have to be made. The RHS indicates ’either move to the state \( q(0) \) and replace the symbol on the top of the stack with the string \( E + T \) (with \( E \) uppermost) or with the string \( T \).

A slightly different transformation is obtained when the RHS of a production starts with a terminal symbol. Let us
consider for instance the production $P \rightarrow (E)$. The corresponding mapping for $d$ is

$$(q(0), (, P) \rightarrow \{(q(0), E)\}).$$

This denotes ‘when in state $q(0)$ with $P$ on the top of the stack, an input symbol ‘(‘ may be accepted and the PDSA moves to state $q(0)$ and replaces the string on the top of the stack by the string $E$’.

Both of these rules correspond with actual productions of the grammar. We must also, however, introduce a third type of mapping, this time not corresponding to a production. When a terminal symbol occurs which is not the first symbol on the RHS of a production (and hence may be moved onto the stack), we need to be able to compare it with the input symbol and, if they match, to remove it from the stack. This gives rise to mappings like

$$(q(0), +, +) \rightarrow \{(q(0), \lambda)\}.$$  

If we now follow the sequence of moves made by our NDPDSA in accepting the string $b.(a + c)$ the process of checking a valid input string will hopefully be clarified. (In the next section we shall introduce another way of looking at this process which should clarify matters even further.)

The final symbol is accepted and the stack is empty, so the input string is accepted as valid. Because of the non-determinism the autonomous moves must be selected from amongst those possible so as to produce a sequence of moves which results in the final acceptance of the input string. It is not merely a question of counting as in the well formed brackets example. The stack is used to anticipate possible developments in the incoming sequence and to remove such anticipations as they occur.

The above example only shows the stack automaton taking the correct actions. Since the actions are defined by the production rules, obviously there is the possibility of taking incorrect action. If this happens the procedure must not immediately fail, but simply back-track and try an alternative path from what is, after all, a finite set of possible routes. Only when all possibilities have been exhausted does the system move to a FAIL state.

The fundamental result which relates CFL’s and NDPDSA’s is

**Theorem** (Chomsky, 1962). The class of languages accepted by NDPDSA’s is precisely the class of CFL’s.

For a proof the reader is referred to Hopcroft and Ullman (section 5.3).

**Derivation trees and CFG’s.**

It is useful to display the derivation of a particular string from the production rules as a tree. These pictures, called derivation or parse trees, superimpose a structure on the words of a language that is useful in applications such as the compilation of programming languages. The vertices of a derivation tree are labelled with terminal or variable symbols (including $\lambda$) of the grammar. If an interior vertex is labelled $A$ and the sons of $A$ are labelled $X(1), X(2), \ldots, X(k)$ from the left then

$$A \rightarrow X(1)X(2)\ldots X(k)$$

must be a production in the grammar. The label of the root is the start symbol $S$. Finally, if a vertex has the label $\lambda$ then it is a leaf and the only son of its father.

**Example.** Consider the grammar $G = \langle \{S, A\}, \{a, b\}, P, S\rangle$, where $P$ consists of

$$S \rightarrow aAS, S \rightarrow a, A \rightarrow bA, A \rightarrow SS, A \rightarrow ba.$$  

The derivation tree for the string $aabbaa$ is shown in Figure 3.3.
Note that all the conditions outlined above are met in this case. The yield of the tree is \textit{aabbaa}. Note that in this case all leaves had terminals for vertices, but there is no reason why this should always be the case. If a leaf is labelled by a variable (i.e. there are no productions with that variable on the left) then it means that no string of terminals can be derived from the variable. In this case the variable is useless and can be eliminated from the grammar. The other possibility is that the leaf is labelled by \( \lambda \); in this case the leaf is the only son of its father.

If \( M \) is an acceptor for a context free grammar \( G \) the acceptance of a string \( x \) by \( M \) amounts to the assertion that there is at least one derivation tree in grammar \( G \) with yield \( x \). (Theorem 4.1, Hopcroft and Ullman). Moreover this tree can be determined by a constructive process. Often there will be more than one derivation corresponding to a particular parse tree. In the example given the \textit{leftmost derivation} corresponds to

\[
S \rightarrow aAS \rightarrow aShAS \rightarrow aabAS \rightarrow aabbaS \rightarrow aabbaa
\]

and the \textit{rightmost derivation} to

\[
S \rightarrow aAS \rightarrow aAa \rightarrow aShAA \rightarrow aShbaa \rightarrow aabbaa.
\]

**The Pumping Lemma for CFL’s.**

The pumping lemma for regular sets states that every sufficiently long string in a regular set contains a short substring that can be pumped. That is, inserting as many copies as we like always yields a string in the regular set. The pumping lemma for CFL’s states that there are always two short substrings close together that can be repeated, both the same number of times, as often as we like. The formal statement is as follows.

**Lemma** (Pumping lemma for CFL’s). Let \( L \) be any CFL. Then there is a constant \( n \), depending only on \( L \), such that if \( z \in L \) and \( l(z) \geq n \), then we may write \( z = uvwxy \) such that

(i) \( l(vx) \geq 1 \),

(ii) \( l(vwx) \leq n \),

and (iii) For all \( i \geq 0 \) we have \( uv^iwx^iy \in L \).

**Proof.** See Hopcroft and Ullman section 6.1.

**Example.** Consider the language \( L \) of words \( a^ib^jc^i \), where \( i > 0 \). Suppose \( L \) were context free and let \( n \) be the constant of the above lemma. Consider \( z = a^n b^n c^n \). Write \( z = uvwxy \) so as to satisfy the conditions of the lemma. We ask ourselves where \( v \) and \( x \), the strings that get pumped, could lie in \( a^n b^n c^n \). Since \( l(vwx) \leq n \), it is not possible for \( v \) to contain instances of \( a \)'s and \( c \)'s, because the rightmost \( a \) is \( n + 1 \) positions away from the leftmost \( c \). If \( v \) and \( x \) consist of \( a \)'s only, then \( uv \) (the string \( uv^iwx^iy \) with \( i = 0 \)) has \( n \) \( b \)'s and \( n \) \( c \)'s but fewer than \( n \) \( a \)'s, since \( l(vx) \geq 1 \). Thus, \( uvw \) is not of the form \( a^n b^n c^n \). But by the Pumping lemma \( uvw \in L \), a contradiction. Hence \( L \) is not context free.

**Exercise 2.1.** Show that the language of strings of the form \( a^{i,j}b^ic^j \), where \( i, j \geq 1 \), is not a CFL.
Solution. Suppose $L$ is a CFL and let $n$ be the constant of the Pumping lemma. Consider the string $z = a^n b^n c^n d^n$. Let $z = uvwx$ satisfy the conditions of the Pumping lemma. Then as $l(vwx) \leq n$, $vx$ can contain at most two different symbols. Furthermore, if $vx$ contains two different symbols, they must be consecutive, for example, $a$ and $b$. Now if $vx$ contains only $a$’s, then $uvy$ has fewer $a$’s than $c$’s and so is not in $L$, a contradiction. We proceed similarly if $vx$ consists of only $b$’s, only $c$’s, or only $d$’s. Now suppose $vx$ has $a$’s and $b$’s. Then $vwy$ still has fewer $a$’s than $c$’s. A similar contradiction occurs if $vx$ consists of $b$’s and $c$’s or $c$’s and $d$’s. Since these are the only possibilities, we conclude $L$ is not context free.

Chapter references


Exercises for Chapter 3

1. Define a Non-deterministic Push Down Automaton (NDPDA) and describe the relationship between NDPDA’s and Deterministic PDA’s.

For which class of languages in the Chomsky hierarchy do NDPDA’s act as acceptors. Define precisely the corresponding grammars.

Construct a push down automaton that accepts by empty stack all those strings on 0,1 that contain an equal number of 0’s and 1’s. (Note: this is not the same as accepting $n$ 0’s followed by $n$ 1’s.)

2. Define the terms Deterministic and Non-deterministic Push Down Stack Automata (DPDSA and NDPDSA). For what class of languages in the Chomsky hierarchy do NDPDSA’s act as acceptors?

Consider the language $L$ defined over the terminal symbols $\{0, 1\}$ such that

$L = \{ z : z = w[w] , w \in (0 + 1)^* \}$

where $[w]$ denotes the string $w$ written backwards, e.g. if $w = 0011$ then $[w] = 1100$. This is the language of (even length) palindromes over the terminal symbols $\{0, 1\}$.

Develop a grammar that generates $L$ and state to which class of languages $L$ belongs and why. Give a plausible explanation of why $L$ can be accepted by a NDPDSA but NOT accepted by a DPDSA.
Chapter 4 Turing machines: Context-sensitive and Phrase structured languages

Introduction.

In this chapter we introduce the Turing machine a simple mathematical model of a computer. Despite its simplicity, the Turing machine models the computing capability of a general purpose computer. The Turing machine is studied both for the class of languages it defines, the phrase structured or recursively enumerable languages, and the class of functions it computes, the partial recursive functions.

The theoretical importance of the Turing machine to computing, transformational grammar and logic is difficult to over-estimate. To understand the reasons why this is so we need to look back to 1900. At the turn of the century the mathematician Hilbert posed a list of outstanding problems for the mathematicians of the twentieth century. Perhaps the most important of these was the decision problem for the first-order predicate calculus.

Logical systems specify a set of legal strings of symbols called well-formed formulas (wffs), which, when appropriately interpreted become statements which are either true or false. Some wffs are designated as axioms and serve with rules of inference as the basis for the production of theorems. Each theorem has a proof, which is a finite sequence of string transformations leading from the axioms to the wff which is the statement of the theorem. Thus in any logical system there is a 'decision problem': is the set of theorems a decidable subset of the set of wffs?

The generally accepted definition of a 'decidable' subset has emerged as roughly: if and only if there is a Turing machine which given a member of the set can effectively determine whether it is a member of the subset. Thus if the decision problem for a particular logical system is soluble it means that it is possible to discover theorems by submitting wffs to a Turing machine and letting it pronounce 'yes' or 'no'.

In each logical system theorems are valid wffs; the completeness problem for the system asks whether every wff is a theorem. That is, it essentially asks whether every identically true statement that can be expressed within the system can also be proved within the system. This is clearly a desirable property for a logical system to have. We shall now briefly detail the propositional and predicate calculi, and consider their decision and completeness problems.

The propositional calculus, the 'logic' referred to in computer hardware, only allows wffs that (apart from fixed symbols) have simple variable names, say p, q, and r. For example in the wff

\[(p \text{ AND } q) \text{ OR } r \text{ implies } (p \text{ OR } r)\]

one might interpret p as 'the voltage on line one is 9 volts'. The decision problem is soluble by truth tables or Karnaugh maps. The propositional calculus is complete.

The predicate calculus, as its name suggests, allows one to name not only individuals but properties or predicates of variables. Given a predicate \(P(x)\) one can ask is it true for all \(x\) or (at least) if there exists some \(x\) for which it is true. Consequently the predicate calculus allows 'quantification': the use of fixed symbols to stand for 'for every' and 'there exists'.

The first-order predicate calculus restricts the application of quantifiers to simple variables so that

\[(\text{for every } x)(M(x) \text{ implies } H(x))\]

(which might be interpreted as every Man is Human) is a legal wff whereas the principle of mathematical induction

\[(\text{for every } P)(P(0) \text{ and } (\text{for every } n)(P(n) \text{ implies } P(n+1)))\]
implies

(for every n)(P(n))

is not. This last expression is a wff in the second-order predicate calculus which differs from the first only in that quantification over predicate names is allowed.

It turns out that the first-order predicate calculus (and hence the second order also) is undecidable, a result first proved by Church (1936) in his original paper on computability.

The first-order predicate calculus is nevertheless complete, a result first proved by Kurt Gödel in 1930. However, the second-order predicate calculus is incomplete, a result proved by Gödel in 1931 and one which had about the same effect on mathematics and logic as Einstein's theory of Relativity had on physics. Gödel's theorem says that any sufficiently rich formal system, for example the second-order predicate calculus and hence most of mathematics, is either inconsistent (in the sense that both a theorem and its negation may be proved) or is incomplete, i.e. contains valid wffs which cannot be proved within the system.

This result came as a bombshell to nineteenth century mathematicians and finally put paid to Hilbert's program for axiomatically constructing the whole of mathematics.

<table>
<thead>
<tr>
<th>Decidable</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional</td>
<td>Yes</td>
</tr>
<tr>
<td>First-order</td>
<td>No</td>
</tr>
<tr>
<td>Second-order</td>
<td>No</td>
</tr>
</tbody>
</table>

**Context-sensitive languages**

After context free next on the scale of generality (i.e. with even fewer restrictions) are the context sensitive languages. Despite the restrictive sounding name, these languages are indeed less restrictive than the context free languages.

**Definition.** Let \( G = \langle V(N), V(T), P, S \rangle \) be a grammar. Suppose that for every production \( x \rightarrow y \) in \( P \) we have \( l(x) \leq l(y) \), where \( l(x) \) is the number of symbols in the string \( x \). Then \( G \) is context sensitive.

Notice that context free languages are indeed a subset of context sensitive languages because the productions \( A \rightarrow x \) of a context free language have \( l(A) = 1 \leq l(x) \).

**Example.** Let \( G = \langle V(N), V(T), P, S \rangle \) where \( V(N) = \{ S, A, B \} \), \( V(T) = \{ a, b, c \} \) and \( P \) consists of the productions

\[
S \rightarrow aSBC, \quad S \rightarrow aBC, \quad CB \rightarrow BC, \quad aB \rightarrow ab, \quad bB \rightarrow bb, \quad bC \rightarrow bc, \quad cC \rightarrow cc.
\]

The language \( L(G) \) contains the word \( a^n b^n c^n \) for each \( n \geq 0 \), since the production \( S \rightarrow aSBC \) used \( n - 1 \) times gives

\[
S \rightarrow a^{n-1} S(BC)^{n-1}.
\]

Then we use the production \( S \rightarrow aBC \) to get

\[
S \rightarrow a^n (BC)^n.
\]

The production \( CB \rightarrow BC \) enables us to arrange the \( B \)'s and \( C \)'s so that all \( B \)'s precede all \( C \)'s and derive

\[
S \rightarrow a^n B^n C^n.
\]

Next we use \( aB \rightarrow ab \) once to get

45
The production $bB \rightarrow bb$ used $n - 1$ times then gives

$$S \rightarrow a^n b^n C^n.$$ 

Finally we use the production $bC \rightarrow bc$ followed by the production $cC \rightarrow cc$ together $n - 1$ times to get

$$S \rightarrow a^n b^n c^n$$

as required.

Each of the seven productions of the grammar has at least as many symbols on the right as on the left. So, this grammar is context sensitive.

**Exercise 4.1** Show that the words $a^n b^n c^n$ ($n > 0$) are the only terminal strings in $L(G)$.

What is the purpose of this restriction $l(x) \leq l(y)$? It is not hard to see that by placing a restriction of this kind on the productions, an upper bound is placed on the number of substitutions that could have been made in forming any given length string. This is very helpful when designing an acceptor automaton; without the restriction $l(x) \leq l(y)$ the intermediate derivations of a valid string may be much longer than the string itself. Because of this, for phrase structured languages we cannot guarantee that only a finite amount of computation is involved in searching for a derivation of any given string.

**Exercise 4.2** Let $L$ be context sensitive. Assume that no intermediate string is repeated in the derivation of a string $t$ (for if this were the case we could find a shorter derivation). If there are $m$ symbols in $V(N) \cup V(T)$ then show that an upper bound on the number of intermediate strings occurring in the derivation of $t$ is not more than $(m + 1)^{l(t)}$.

Some authors require that the productions of a context sensitive grammar be of the form

$$xAy \rightarrow xzy$$

with $x, y, z$ in $V^*$, $z \neq \lambda$, and $A$ in $V(N)$. It can be shown that this definition is equivalent to that given above. However, it does motivate the name ‘context sensitive’ since the production $xAy \rightarrow xzy$ allows $A$ to be replaced by $z$ whenever $A$ appears in the context of $x$ and $y$.

It turns out that the model for an automaton which can act as an acceptor for context sensitive languages is a restricted type of Turing machine. Before discussing Turing machines let us briefly consider a problem associated with the most general class of language we shall consider, the phrase structured languages.

Recall that the production rules of a phrase structured grammar are of the type $x \rightarrow y$, with no restriction on the length of $x$ with respect to that of $y$. For a genuinely phrase structured language it follows that any grammar will have production rules of the type $x \rightarrow y$, where $l(x) \leq l(y)$. Without the safeguard that $l(x) \leq l(y)$ we do not know how many backward steps are needed in parsing a given length string before abandoning the search. This simply means that phrase structured languages cannot be parsed with any certainty that the process will terminate.

**Turing machines**

The basic model of a Turing machine has a finite control, an input tape which is divided into cells, and a tape head which scans one cell of the tape at a time. The tape can be regarded as infinite in both directions, but we will make the restriction that when the tape is started the tape must be blank, except for some finite number of cells. The tape head has three functions, all of which are used in each operation cycle of the FSM controller. These functions are: reading the cell of the tape being scanned, writing on the scanned square, and moving left or right to an adjacent cell (which becomes the scanned cell in the next operation cycle). Each cell of the tape may hold exactly one of a finite set of tape
symbols. When a symbol is printed on the tape the previous symbol in the cell is erased.

**Definition.** A *Turing machine* $T$ is a 6-tuple $\langle Q, I, J, q(0), F, d \rangle$ where

- $Q$ is a finite set of states.
- $I$, a subset of $J - \{B\}$, is a finite set of input symbol initially on the tape.
- $J$ is a finite set of tape symbols ( One of these, usually denoted by $B$, is the blank.).
- $q(0)$ in $Q$ is an initial state.
- $F$, a subset of $Q$, is a set of final states.
- and $d$ is a function which maps $Q \times J \rightarrow Q \times (J - \{B\}) \times \{L, R\}$.

In each operation cycle the machine starts in some state $q(t)$, reads the symbol $j(t)$ written on the cell under the head, prints the new symbol $j(t+1)$, moves left ($L$) or right ($R$), and then enters the new state $q(t+1)$. Because the machine can move either way along the tape, it is possible to use the tape for the storage of arbitrarily large amounts of useful information. Of course for practical purposes (not that anyone actually bothers to build Turing machines) we may regard the tape as finite, with the provision that whenever the machine comes to an end of the finite portion, someone will attach another cell.

The definition given is that of a *deterministic* Turing machine. As with other automata we can replace the mapping $d$ by a function that maps

$Q \times J \rightarrow \text{the set of all subsets of } Q \times (J - \{B\}) \times \{L, R\}$

and thereby define a *non-deterministic* Turing machine.

However, as with FSM’s it emerges that deterministic and non-deterministic Turing machines are equivalent.

Suppose now that we have a string $s$ composed of symbols from $J - \{B\}$. Imagine that $s$ is written onto the tape in consecutive cells and that the head of a Turing machine $T$ is placed at the first symbol, $j(0)$, of $s$. The machine is now set to state $q(0)$ and started. If the Turing machine eventually reaches one of the states $q$ in $F$, in which case it halts, we say the string is *accepted* by $T$. However, for strings not accepted, it is possible that $T$ will never halt.

**Example.** Consider the following Turing machine that recognises the context free language $a^n b^n$ for all $n > 0$. Let $T = \langle Q, I, J, q(0), F, d \rangle$ where

- $Q = \{q(0), q(1), q(2), ..., q(4)\}$,
- $I = \{0, 1\}$,
- $J = \{0, 1, B, X, Y\}$,
- $F = \{q(4)\}$,

and $d$ is described by the 5-tuples below, in which

$q(i) \textit{ a b q(j) R}$

is interpreted as 'If in state $q(i)$ and reading $a$ then write $b$, change state to $q(j)$, and move Right.' ($R = \text{right, } L = \text{Left}$).

- $q(0) \textit{ 0 X q(1) R}$
- $q(0) \textit{ Y Y q(3) R}$
- $q(1) \textit{ 0 0 q(1) R}$
- $q(1) \textit{ Y Y q(1) R}$
- $q(1) \textit{ 1 Y q(2) L}$
- $q(2) \textit{ 0 0 q(2) L}$
- $q(2) \textit{ Y Y q(2) L}$
- $q(2) \textit{ X X q(0) R}$
- $q(3) \textit{ Y Y q(3) R}$
- $q(3) \textit{ B B q(4) R}$
- $\text{HALT}$
The machine starts in state $q(0)$ reads the leftmost 0, replaces it by an $X$ and enters state $q(1)$ and moves right. In state $q(1)$ it moves right, skipping over 0’s and Y’s until it finds the leftmost 1. Note that if a $B$ or an $X$ is encountered in state $q(1)$ then the input is rejected, either there are too many 0’s or the input is not in $0^*1^*$.

Moving right in state $q(1)$ on finding the leftmost 1 the machine changes it to a Y, enters state $q(2)$ and moves left. In state $q(2)$ it moves left, skipping over 0’s and Y’s, until it finds the leftmost 1. Note that if a $B$ or an $X$ is encountered in state $q(2)$ then the input is rejected, either there are too many 0’s or the input is not in $0^*1^*$.

If, having found the rightmost $X$, after entering state $q(0)$ the machine reads a Y (immediately to the right of the rightmost $X$) then the 0’s are exhausted. Now the machine skips the $Y$, enters state $q(3)$ and moves right. In state $q(3)$ the machine skips right over $Y$’s checking that no 1’s remain. If the $Y$’s are followed by a $B$ then state $q(4)$ is entered and the string is accepted: otherwise the string is rejected.

**Exercise.** Check the action of the machine on some rejected inputs such as 001101, 001, and 100.

**Example.** A Turing machine which can read and write blanks B, has states $Q = \{q(0), q(1), q(2), q(3), q(4)\}$, with single final state $q(4)$, and tape symbols $J = \{B, 1\}$. The initial state is $q(0)$ and the machine starts with the reading head above the leftmost symbol of the input tape. It is described by the 5-tuples below.

- $q(0) \ 1 \ 1 \ q(0) \ R$
- $q(0) \ B \ 1 \ q(1) \ L$
- $q(1) \ 1 \ 1 \ q(1) \ L$
- $q(1) \ B \ B \ q(2) \ R$
- $q(2) \ 1 \ 1 \ q(2) \ R$
- $q(2) \ B \ B \ q(3) \ L$
- $q(3) \ 1 \ B \ q(4) \ R$
- $q(4) \ R$

Describe the complete operation of this machine when given the input string consisting of $m$ 1’s followed by $B$ followed by $n$ 1’s ($m, n > 0$). Specify a simpler Turing machine which accomplishes the same effect.

**Solution.** The machine starts in state $q(0)$ and moves right reading and writing 1’s, remaining in state $q(0)$, until it encounters the separating blank $B$. It then replaces $B$ with a 1, enters state $q(1)$ and moves left.

In state $q(1)$ the machine moves left reading and writing 1’s, remaining in state $q(1)$, until it encounters the $B$ preceding the leftmost 1. It then writes a $B$, enters state $q(2)$ and moves right.

In state $q(2)$ the machine moves right reading and writing 1’s, remaining in state $q(2)$, until it encounters the $B$ following the rightmost 1. It then writes a $B$, enters state $q(3)$ and moves left.

In state $q(3)$ the machine replaces the rightmost 1 with a $B$, enters state $q(4)$ and halts.

The overall effect is to replace the string ($m$ 1’s)$B$(n 1’s) with the string ($m+n$ 1’s). In effect the Turing machine has performed an addition of two positive integers.

**Simpler machine:**

A simpler machine which has the same effect on the initial string is

- $q(0) \ 1 \ 1 \ q(0) \ R$
- $q(0) \ B \ 1 \ q(1) \ R$
- $q(1) \ B \ B \ q(2) \ L$
- $q(2) \ 1 \ 1 \ q(1) \ R$
- $q(2) \ B \ B \ q(3) \ R$

(There is at least one other solution.)

**Exercise 4.3 Construct a Turing machine that can multiply any two binary numbers.**

**Definition.** A language that is accepted by a Turing machine is said to be recursively enumerable (r.e.).
The term enumerable derives from the fact that it is precisely these languages whose strings can be enumerated (listed) by a Turing machine. It was proved by Chomsky (1959) that the recursively enumerable languages are precisely the phrase structured languages.

As we indicated earlier the class of r.e. languages includes some languages for which we cannot mechanically determine membership. If \( L \) is such a language, then any Turing machine \( M \) accepting \( L \) must fail to halt on some input not in \( L \). If \( w \) is in \( L \) then \( M \) eventually halts given input \( w \). However, as long as \( M \) is still running on some input, we can never tell whether \( M \) will eventually accept if we let it run on long enough, or whether \( M \) will run on forever.

**Turing’s Hypothesis**

Having introduced and defined Turing machines and indicated how one can be designed to accept a context free language, we can now move on to discuss one of the central results of automata theory. This result, generally known as *Turing’s hypothesis*, asserts the belief that ANY process which can be precisely or effectively described can be realised by a suitable designed Turing machine. One reason for the acceptance of the Turing machine as a general model of a computation is that the model we defined is invariant to many modifications which would seem, off-hand, to increase the computing power of the device. For example if a language is accepted by a ‘multitape’ Turing machine it is accepted by a single tape Turing machine. Since 1936 many great logicians, mathematicians, and computer scientists have framed alternative definitions of ‘computable’. All of them have come up with the same solution in the sense that the set of computable functions or effective procedures is the same in all cases. At any rate the belief that Turing’s hypothesis is valid is now so widely held that it is often turned ‘backwards’ and used as a definition of an ‘effective description’.

**Universal Turing machines**

It is plain that the operation of any Turing machine (with any tape) could be simulated by a general purpose digital computer. It is not so much more difficult to see that the converse also holds, namely the operation of any digital computer can be simulated by a suitably contrived Turing machine. After all, the operation of any computer proceeds according to simple rules, e.g. register-to-register transfers, elementary arithmetic and logic operations and so forth. We merely have to convince ourselves that each of these operations can be designed into a Turing machine.

Continuing in this vein it can be proved that we can devise a particular Turing machine on which we can simulate the operation of ANY Turing machine, a so called universal Turing machine. This is not really so surprising: for example it is perfectly feasible to write a FORTRAN interpreter in BASIC or an interpreter for any other computer programming language and, after all, being able to simulate any Turing machine is just like the action of an interpreter. Thus any BASIC home computer is also universal in the same sense.

To see this more clearly think what we do when we follow the action of a given Turing machine (the reader should have done this at least once by now). We probably get a paper and pencil (Turing tape), write down the initial state of the tape and, using the table which is the description of the Turing machine, proceed to emulate the head moves, reads and writes by recording the new intermediate states of the tape of the paper. Plainly, we can (in principle) do this for any Turing machine; it is a totally mechanical process. It is inconceivable that there is a description table and initial tape state for which we could not do it. In so doing we are acting as a universal Turing machine, and it should be intuitively apparent that one can also give a description of a Turing machine which does exactly the same thing.

The interested reader can find a detailed construction of a universal Turing machine in Minsky (Chapter 7, see page 142).

**Mini-project.** Using the description of a universal Turing machine in Minsky’s book write a computer program in any convenient language to simulate the action of the machine.

Does this mean that a universal Turing machine can solve any conceivable problem? We shall see that this is far from being the case.
Linear bounded automata and context sensitive languages

Definition. A linear bounded automaton (LBA) is a non-deterministic Turing machine which never leaves those cells on which the input was placed. The set of input symbols $I$ contains two special markers, which are the left and right end markers. These symbols are initially at the ends of the input string and their function is to prevent the tape head from leaving the region of the tape upon which the input appears.

It is not known whether the class of languages accepted by NDLBA’s properly contains the class accepted by DLBA’s. It is of course true that any language accepted by a NDLBA is accepted by a deterministic Turing machine. However the amount of tape required by that Turing machine may be an exponential function of the length of the input.

The interest in NDLBA’s stems from the fact (Kuroda, 1964) that the class of languages accepted is precisely the class of context sensitive languages.

The halting problem

When a Turing machine, with its input tape is started it may be a very, very long time before it completes its computation and comes to a halt. For many machine-tape pairs this will never happen, the ‘computation’ may go on for ever. It would be useful to have a decision procedure which would enable us, given any Turing machine $T$ and tape $t$, to determine whether the process will ever halt. If there is any decision procedure then it, by definition, can be executed by a Turing machine $A$ which takes as input a pair $(d(T), t)$, where $d(T)$ is some description of $T$ and $t$ is the tape, and which is guaranteed to halt printing ‘YES’ or ‘NO’, meaning $T$ does eventually halt given $t$ or $T$ does not halt given $t$, respectively.

If $A$ can solve the problem for all pairs $(d(T), t)$, then it can certainly do so for the special pairs $(d(T), d(T))$, where the tape $t$ is a description of $T$, rather than any old tape. We can therefore construct a new Turing machine $B$, which requires only one copy of the description $d(T)$ but otherwise behaves like $A$ applied to $(d(T), d(T))$. Thus $B$ scans $d(T)$ and halts printing

\[
\text{YES, if } T \text{ halts given } d(T), \\
\text{NO, if } T \text{ never halts given } d(T).
\]

Now modify $B$ by adding a loop to the YES exit. This new Turing machine $C$ obeys the rules

\[
\text{If } T \text{ halts given } d(T) \text{ print YES and go into a loop.} \\
\text{If } T \text{ never halts given } d(T) \text{ print NO and halt.}
\]

Now for the killer: what happens if $C$ is applied to the tape $d(C)$? Plainly it halts if $C$ applied $d(C)$ does not halt, and never halts if $C$ applied to $d(C)$ halts.

This is a contradiction and so we must conclude that a Turing machine such as $C$, and hence $B$, and hence $A$, cannot exist.

The unsolvability of the halting problem generalises to any computing machine which can manipulate data and interpret it as instructions. Incidentally, this does not sound the death knoll to the aspirations of artificial intelligence for there is no reason to suppose that human beings are not subject to the same limitations.

Other unsolvable problems.

Many genuinely unsolvable problems are known. In the context of the present course we list below some relevant questions together with what is known about their solvability or unsolvability.

\[
\text{RL = Regular languages} \\
\text{CF = Context free languages}
\]

50
CS = Context sensitive languages
PS = Phrase structured languages

<table>
<thead>
<tr>
<th>Question</th>
<th>RL</th>
<th>CF</th>
<th>CS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is L(G) empty/finite/infinite</td>
<td>S</td>
<td>S</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Does L(G) = V*</td>
<td>S</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Is L(G(1)) = L(G(2))</td>
<td>S</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Is L(G(1)) a subset of L(G(2))</td>
<td>S</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Is L((G(1)) INTERSECT L(G(2))) e/f/i</td>
<td>S</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Does L(G) = specific regular set</td>
<td>S</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Is L(G) a regular set</td>
<td>T</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Closed under INTERSECT</td>
<td>T</td>
<td>U</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Closed under COMPLEMENT</td>
<td>T</td>
<td>U</td>
<td>?</td>
<td>U</td>
</tr>
<tr>
<td>Closed under concatenation</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Closed under UNION</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

S = Solvable                                     U = Unsolvable
e=Empty                                          f=Finite
i= Infinite

Chapter references


Hopcroft, J. E. and Ullman, J. D. Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.


Exercises for Chapter 4.

1. Define the terms Context Free Grammar and Context Sensitive Grammar. Extend these descriptions to define the corresponding classes of languages. For each of the two classes of languages describe the corresponding class of ‘accepting’ automata.

Give a plausible explanation why the language \( L = \{a^n b^n c^n : n \geq 1 \} \) is NOT Context Free. Construct a grammar for this language, giving sufficient explanation of your production rules to make it clear that they do indeed generate \( L \).

2. Explain the operation of a Turing machine.

Define the notion of a grammar and describe the production rules permitted in a Phrase Structured grammar and a Context Free grammar. Why is it that a syntax checker for a Phrase Structured language cannot be guaranteed to parse every string successfully.

Give an informal construction of a Turing machine which, given an input string of \( a \)'s, \( b \)'s and \( c \)'s, changes \( a \)'s to \( b \)'s and \( b \)'s to \( a \)'s and erases (replaces by the blank \( B \)) every \( c \) but the last.
SOLUTIONS TO EXERCISES

Solutions for Chapter 2

1(i). **Definition:** A *Finite State Machine* is an ordered 5-tuple $< I, Z, Q, d, w >$ where

- $I$ is a finite set of discrete input messages.
- $Z$ is a finite set of discrete output messages.
- $Q$ is a finite set of internal states.
- $d$ is a rule which given the current state $q(t) \in Q$ and an input $i(t) \in I$, defines the next state $q(t+1)$ in $Q$, i.e. $q(t+1) = d(q(t), i(t))$.
- $w$ is a rule which given the current state $q(t) \in Q$ and an input $i(t) \in I$, defines the corresponding output $z(t)$ in $Z$, i.e. $z(t) = w(q(t), i(t))$.

A subtle point may need stressing here, namely that the output $z(t)$ is measured immediately, but by the time one measures the state it has changed to $q(t+1)$.

For the given FSM:

<table>
<thead>
<tr>
<th>Function $d$</th>
<th>Function $w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i(t)$</td>
<td>$i(t)$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Input messages $I = \{0, 1\}$
Output messages $Z = \{0, 1\}$
Internal states $Q = \{q(0), q(1), q(2)\}$

<table>
<thead>
<tr>
<th>$i(t)$</th>
<th>$i(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q(0)</td>
</tr>
<tr>
<td>1</td>
<td>q(0)</td>
</tr>
<tr>
<td>q(0)</td>
<td>q(0)</td>
</tr>
<tr>
<td>q(1)</td>
<td>q(2)</td>
</tr>
<tr>
<td>resulting state</td>
<td>q(t+1)</td>
</tr>
<tr>
<td>q(1)</td>
<td>q(1)</td>
</tr>
<tr>
<td>q(2)</td>
<td>q(2)</td>
</tr>
<tr>
<td>resulting output</td>
<td>q(2)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>z(t)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Informal description:** All 1’s are replaced by 0’s and single 0’s are replaced by 1’s until the second of two consecutive 0’s is read; this symbol is also replaced by a 1 and thereafter any symbol is replaced by 0.

(ii) **Regular Expressions.**

(i) Any terminal symbol $t$ is a regular expression. It describes the set consisting of precisely the (one letter) string $t$.

(ii) If $E$ and $F$ are regular expressions then so is $EF$ and this describes the set $\{x : x = ef\}$, where $e$ is a string in the set described by $E$ and $f$ is a string in the set described by $F$.

(iii) If $E$ and $F$ are regular expressions then so is $E + F$ and this describes the set $\{x : x = e \text{ or } f\}$, where $e$ is a string in the set described by $E$ and $f$ is a string in the set described by $F$.

(iv) If $E$ is a regular expression then so is $E^*$.  

**Definition.** The *regular expressions* are all those that can be constructed from the above four rules and no others.
Relationship FSM/Regular expressions: The FSM's are precisely the acceptors for the sets of strings described by regular expressions.

(iii) State diagram in Figure 5.1

[Diagram description. States q(0) (initial), q(1), q(2), q(3), Fail. Accept states q(1), q(2), q(3).
Arcs
From To Labelled/Input
q(0) q(1) a+b
q(0) q(2) c
q(2) q(2) b+c
q(2) q(3) a
q(3) q(3) b+c
q(3) FAIL a
]

Regular expression: (a + b)(a + b + c)* + c((b + c)* + (b + c)*a(b + c)*)

2(i) Definition: A (deterministic) Finite State Machine is an ordered 5-tuple <I, Z, Q, d, w> where

I is a finite set of discrete input messages.
Z is a finite set of discrete output messages.
Q is a finite set of internal states.
d is a rule which given the current state q(t) ∈ Q and an input i(t) ∈ I, defines the next state q(t+1) in Q. i.e. q(t+1) = d(q(t), i(t)).
w is a rule which given the current state q(t) in Q and an input i(t) in I, defines the corresponding output z(t) in Z. i.e. z(t) = w(q(t), i(t)).

A subtle point may need stressing here, namely that the output z(t) is measured immediately, but by the time one measures the state it has changed to q(t+1).

Definition. The NDFSM is a generalisation of the idea of the (deterministic) FSM. We note that in the definition of an FSM the next state function d : Q × I → Q. In the definition of an NDFSM we replace this function by d : Q × I → Η(Q), where Η(Q) is the set of all subsets of Q.

In terms of an acceptor this is interpreted as a finite tree search in the sense that if there exist a path through the state diagram which, for the given input string, leads to an accept state then the string is accepted.

Characteristic feature: there is at least one state in the diagram which has more than one arrow exiting for the same input.

(ii) Equivalent DFSM: Most simply described by a diagram constructed by means of the algorithm given in the course - see Figure 5.2

States p(0) (initial state), p(1), p(2).
Accept states p(1), p(2).
Arcs
From To Labelled/Input
p(0) p(0) 0
p(0) p(1) 1
p(1) p(0) 1
p(1) p(2) 0
p(2) p(2) 0+1

Figure 5-2 Solution to Q2.2(ii).
Regular expression/analysis:

First solution.

\[ 0^* \quad \text{Resulting state } q(0) \]
\[ 0^*1 \quad \text{Resulting state } q(1) \]

Now any string \(0^*(0 + 1)\) may cause a transition from \(q(1)\) to \(q(0)\).

\[ 0^*10^*(0 + 1) \quad \text{Resulting state } q(0) \text{ or } q(1) \]

If 1 occurs in \((0 + 1)\) the FSM must now be in state \(q(0)\). There must be at least one 1 in the input string if it is to be accepted. Suppose now that we are about to process the LAST 1. Then we must be in state \(q(0)\) and we have gone around the loop \((0^*10^*(0 + 1))^*\) for the last time. In state \(q(0)\) we can have an arbitrary number of 0's followed by the final 1, which takes us to state \(q(1)\). The string so far is \((0^*10^*(0+1))^*0^*1\), we are in state \(q(1)\) and the last 1 has been processed. Finally the string may terminate in an arbitrary number of 0's. This yields

\[(0^*10^*(0+1))^*0^*10^*\]

as the resulting regular expression.

Second solution. All acceptable strings must have a 'head' which moves to \(q(0)\) followed by \(10^*\). Now \(q(0)\) is also the start state. So the problem reduces to characterising 'heads', i.e. strings which map \(q(0)\) to \(q(0)\). The two possible paths are \(0^* + 0^*10^*(0+1)\) and we can go around any number of times. So another answer is

\[(0^* + 0^*10^*(0 + 1))^*10^*\]

Remark 1. Other convincing solutions were

\[ 0^*1(10^*1)^* + 0^*1(10^*1)^*0(0 + 1)^* \]
\[ (0 + 11)^*(1 + (10(0 + 1))^*) \]
\[ 0^*1 + (0^*1)(10^*1)^* + ((0^*1) + (0^*1(10^*1)^*))0(0 + 1)^* \]

Remark 2. One fact, which is clear from the DFSM but not immediately obvious from either regular expression, is that if at any time the machine is in state \(q(0)\) and receives the input string 10 then any tail of 0's and 1's is acceptable.

3. First part (bookwork). Regular Expressions.

(i) Any terminal symbol \(t\) is a regular expression. It describes the set consisting of precisely the (one letter) string \(t\).

(ii) If \(E\) and \(F\) are regular expressions then so is \(EF\) and this describes the set \(\{x : x = ef\}\), where \(e\) is a string in the set described by \(E\) and \(f\) is a string in the set described by \(F\).

(iii) If \(E\) and \(F\) are regular expressions then so is \(E + F\) and this describes the set \(\{x : x = e \text{ or } f\}\), where \(e\) is a string in the set described by \(E\) and \(f\) is a string in the set described by \(F\).

(iv) If \(E\) is a regular expression then so is \(E^*\).

Definition. The regular expressions are all those that can be constructed from the above four rules and no others.

Relationship FSMs:

Theorem (Kleene, 1956). The FSM’s are precisely the acceptors for regular expressions.
Second part.

a) Diagram description.

States q(0) (Initial), q(1), q(2). Accept state q(0). Arcs
From  To  Labelled/Input
q(0)  q(0)  0
q(0)  q(1)  1
q(1)  q(1)  0
q(1)  q(2)  1
q(2)  q(2)  0
q(2)  q(0)  1

Regular expression \((0^*10^*10^*)^*0^*\)

b) Diagram description.

States q(0) (Initial), q(1), q(2), q(3), FAIL. Accept states q(0), q(3). Arcs
From  To  Labelled/Input
q(0)  q(0)  0
q(0)  q(1)  1
q(1)  FAIL  0
q(1)  q(2)  1
q(2)  FAIL  0
q(2)  q(3)  1
q(3)  q(3)  1
q(3)  q(0)  0

Regular expression \((0^*1111^*0)^*0^* + 0^*1111^*\)

c) Diagram description.

States q(0) (Initial), q(1), q(2). FAIL. Accept states q(0), q(1), q(2). Arcs
From  To  Labelled/Input
q(0)  q(1)  0 + 1
q(1)  FAIL  1
q(1)  q(2)  0
q(2)  FAIL  1
q(2)  q(0)  0

Regular expression \(((0+1)00)^* + ((0+1)00)^*\)

4. First part (bookwork). Definition: A (deterministic) Finite State Machine is an ordered 5-tuple \(<I, Z, Q, d, w>\) where

- \(I\) is a finite set of discrete input messages.
- \(Z\) is a finite set of discrete output messages.
- \(Q\) is a finite set of internal states.
- \(d\) is a rule which given the current state \(q(t) \in Q\) and an input \(i(t) \in I\), defines the next state \(q(t+1)\) in \(Q\). i.e. \(q(t+1) = d(q(t), i(t))\).
w is a rule which given the current state \( q(t) \) in \( Q \) and an input \( i(t) \) in \( I \), defines the corresponding output \( z(t) \) in \( Z \). i.e. \( z(t) = w(q(t), i(t)) \).

A subtle point may need stressing here, namely that the output \( z(t) \) is measured immediately, but by the time one measures the state it has changed to \( q(t+1) \).

**Brief description of regular expressions:** A regular expression is a symbolic representation of a set of strings over some finite set of (terminal) symbols. The regular expression is composed from the terminal symbols and three connectives

+ i.e. exclusive OR.

Terminal symbol concatenation i.e. one terminal symbol followed by another.

* which is interpreted as repetitively concatenating the symbol immediately preceding * 'any finite number of times, including zero'.

**Example.** The regular expression \( 0^*(1 + 0) \) denotes the set of strings in which any member begins with any number of 0’s followed by a 0 or a 1. The terminal symbols here are 0 and 1.

**Relationship FSMs:**

**Theorem** (Kleene, 1956). The FSM’s are precisely the acceptors for regular expressions.

Second part.

(a) Yes. \((0 + 1)(0 + 1)^*\)

(b) Yes. \(10^*\)

(c) No. For if it were the case: by the Pumping lemma there is a sufficiently long string containing a non-empty sub-string of 1’s which can be pumped so that the resulting string has length which is not a power of 2 - contradiction.

(d) No. Any FSM must eventually lose count. Note: a Push Down Stack automaton can do it (but this is an infinite machine).

(e) No. The 1’s before and after the 0 must be paired off, any FSM must eventually lose count - again a Stack automaton can do it.

(f) Yes. \((0 + 1)^*E\)

(g) Yes. \((0 + 1)^*E(0 + 1)^*\)

(h) No ! This is not solved by EE. An FSM can recognise \( E = 0^*1 \), but no FSM can recognise \( \{0^n10^n : n \geq 1 \} \).

5. **First part (bookwork).**

**Definition.** A grammar consists of the following:

(a) A vocabulary \( V \) (as before), but this time split into two disjoint sets of symbols:

A finite set of **terminal symbols** \( V(T) \). The elements of \( V(T) \) will normally be denoted by lower case letters \( a, b, \) etc. Terminal symbols are the symbols that actually appear in the strings of the language.

A finite set of **non-terminal symbols** \( V(N) \). The elements of \( V(N) \) will normally be denoted by upper
case letters A, B, etc. Non-terminal symbols do not actually appear in the strings of the language but are used by the production rules (see (b) below) to generate strings of terminals that are syntactically correct. Non-terminal symbols are often called **variables**.

(b) A finite set $P$ of **production rules**. Each production is of the form $x \rightarrow y$, where $x$ and $y$ are strings of symbols from $V^*$, with $x$ not equal to the empty string. Thus $x$ and $y$ are finite strings of variables or terminals. Elements of $V^*$ will normally be denoted by lower case letters $x, y$, etc. The arrow symbol `$\rightarrow$' is read as *may be replaced by*.

(c) A start symbol $S$, which is usually one of the $V(N)$.

A **grammar** is left-linear (or right-linear) if and only if all production rules are of the form $A \rightarrow Bt$ or $A \rightarrow t$, (or $A \rightarrow tB$, $A \rightarrow t$), where $A$ and $B$ are non-terminals and $t$ is a (possibly empty) terminal symbol.

**Relationship:** A language can be accepted by an FSM if and only if it has a left-linear (or right-linear) grammar - the existence of one implies the existence of the other.

**Remark:** both grammars in (a) and (b) are left-linear and so generate languages which can be described by regular expressions.

a) By iterating the first production rule an acceptable string can begin with any number of 0's. This gives 0*. If a 1 occurs, using the second production rule, it may be followed by any number of 1's by iterating rule 3 but must then be followed by a 0, using rule 4, and can then be followed by any number of 0's. This gives $0^*(11^*00^*)$ and the part in brackets may be iterated any number of times by using rules 1-4 again. This gives $0^*(11^*00^*)^*$. Finally, using rules 1 and 5 the string may be followed by any number of 0's before using rule 6. This gives the final regular expression as $0^*(11^*00^*)^*0^*$ or $0^*(11^*00^*)^*$

Another solution is $(0 + 11^*0)^*$

**Diagram description.**

**States** q(0) (Initial), q(1). Accept state q(0).

<table>
<thead>
<tr>
<th>Arcs</th>
<th>From</th>
<th>To</th>
<th>Labelled/Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>q(0)</td>
<td>q(0)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>q(0)</td>
<td>q(1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>q(1)</td>
<td>q(1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>q(1)</td>
<td>q(0)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

b) We must first use rule 1, which then forces us to use rule 2. The string so generated is $bbB$. We may next use rule 3 any number of times, which yields $bb(b^*)B$. We may then use rule 5 to terminate, yielding $bb(b^*)a$, or we may use rule 4 and iterate rules 1-3, as before, any number of times. This gives

$$bb(b^*)a (bb(b^*)a)^*$$

Note that the null string is not accepted. (Unless explicitly excluded the null string is normally accepted.)
Similarly, we could go around the loop more than once, in fact as many times as we like. Thus we can either not go around the loop at all or we can go round it once, either way the string is accepted.

We could avoid this rather bizarre solution by excluding $q(0)$ as an accept state and replacing the FAIL state by an ACCEPT state in the FSM.

Diagram description.

States q(0) (Initial), q(1), q(2), ACCEPT.

Arcs

From | To | Labelled/Input
--- | --- | ---
q(0) | q(1) | b
q(1) | q(2) | b
q(2) | q(0) | a
q(2) | q(0) | a

We have proved is that given any sufficiently long string accepted by a FSM, we can find a substring (within the first $n+1$ symbols) that may be ‘pumped’, i.e. repeated as many times as we like, and the resulting string will still
be accepted by the FSM. We obtain the statement of the lemma by taking:

\[
\begin{align*}
    z &= a(1)a(2)...a(m) \\
    u &= a(1)a(2)...a(j) \\
    v &= a(j+1) ... a(k) \\
    w &= a(k+1) ... a(m).
\end{align*}
\]

\[\square\]

Second part. Suppose the set \(L = \{ 0^{P} : p \text{ prime } \}\) is regular and choose \(p > n\), where \(n\) is the number asserted to exist by the Pumping Lemma. Then we can write \(z = uvw\), where \(l(uv) < n\), \(l(v) \geq 1\). The lemma asserts that \(uvw\) is also in \(L\) for every \(i \geq 0\).

Now

\[
l(uvw) = l(u) + i.l(v) + l(w) = l(u) + l(v) + l(w) + (i-1).l(v) = p + (i-1).l(v)
\]

If we now choose \(i = p + 1\) then

\[
l(uvw) = p + p.l(v) = p.(1 + l(v)) = N \text{ say},
\]

where \(l(v) \geq 1\). The right-hand side, i.e. \(N\), is therefore not prime but according to the lemma \(0^{N} \in L\), and this is a contradiction. Hence \(L\) is not regular. \(\blacksquare\) (Note. This result was first proved by Minsky.)

7. Lemma (Pumping lemma for regular sets). Let \(L\) be a regular set. Then there is a constant \(n\) such that if \(z\) is any word in \(L\) and \(l(z) \geq n\), we may write \(z = uvw\) in such a way that \(l(uv) < n\), \(l(v) \geq 1\), and for all \(i \geq 0\), \(uvw^{i}\) is in \(L\). Moreover, \(n\) is no greater than the number of states in the smallest FSM accepting \(L\).

Hence or otherwise: Theorem. The set of strings accepted by a FSM \(M\) with \(n\) states is

1. Non-empty if and only if the FSM accepts a string of length less than \(n\).
2. Infinite if and only if the FSM accepts some sentence of length \(l\) where \(n \leq l < 2n\).

Proof. (1) The ‘if’ part is obvious. Suppose \(M\) accepts a non-empty set. Let \(w\) be a word as short as any other word accepted. By the pumping lemma, \(l(w) < n\), for if \(w\) were a shortest word and \(l(w) \geq n\), then \(w = uvy\), and \(uv\) is a shorter word in the language.

(2) If \(w \in L(M)\) and \(n \leq l < 2n\), then by the pumping lemma, \(L(M)\) is infinite. That is, \(w = w(1)w(2)w(3)\), and for all \(i\), \(w(1)w(2)^{i}w(3) \in L\). Conversely if \(L(M)\) is infinite, then there exists \(w \in L(M)\), where \(l(w) \geq n\). If \(l(w) < 2n\) we are done. If no string is of length between \(n\) and \(2n - 1\), let \(w\) be of length at least \(2n\), but as short as any string in \(L(M)\) whose length is \(\geq 2n\). Again by the pumping lemma, we can write \(w = w(1)w(2)w(3)\) with \(1 \leq l(w(2)) \leq n\) and \(w(1)w(3) \in L(M)\). Either \(w\) was not a shortest word of length \(2n\) or more, or \(l(w(1)w(3))\) is between \(n\) and \(2n - 1\), a contradiction in either case. \(\blacksquare\)

[It should be noted that the algorithms suggested by the theorem are highly inefficient, but this form of the result is easy to prove.]

(a) Thus from part (1) an algorithm to determine whether \(L(M)\) is empty is: see if any string of length up to \(N\) is in \(L(M)\). Clearly there is such a procedure that is guaranteed to halt.

An easier way to check whether an FSM accepts only the empty set is to take its state diagram and delete all accept states that are not reachable on any input from the start state. If one or more accept states remain, the language is non-empty.
(b) In part (2), the algorithm to decide whether or not \( L(M) \) is infinite is: see if any string of length between \( N \) and \( 2N - 1 \) is in \( L(M) \). Again, clearly there is such a procedure that is guaranteed to halt. As before, if we delete all states that are not reachable on any input from the start state, then without changing the language accepted we may delete all states that are not final and from which one cannot reach a final state. The FSM accepts an infinite language if and only if the resulting state diagram has a cycle.

8. Regular Expressions.

(i) Any terminal symbol \( t \) is a regular expression. It describes the set consisting of precisely the (one letter) string \( t \).

(ii) If \( E \) and \( F \) are regular expressions then so is \( EF \) and this describes the set \( \{ x : x = ef \} \), where \( e \) is a string in the set described by \( E \) and \( f \) is a string in the set described by \( F \).

(iii) If \( E \) and \( F \) are regular expressions then so is \( E + F \) and this describes the set \( \{ x : x = e \text{ or } f \} \), where \( e \) is a string in the set described by \( E \) and \( f \) is a string in the set described by \( F \).

(iv) If \( E \) is a regular expression then so is \( E^* \).

Definition. The regular expressions are all those that can be constructed from the above four rules and no others.

Relationship FSM’s: Theorem (Kleene, 1956). The FSM’s are precisely the acceptors for regular expressions.

Deduce closure: From Kleene’s theorem if a language is accepted by an FSM then the language can be described by a regular expression \( E \) (say). By Axiom (iv) for regular expressions \( E^* \) is also regular and hence, again by Kleene’s theorem, also describes a language accepted by a FSM. This proves the result.

Last part: One possible solution is given in Aleksander and Hanna p 124. The machine described there is a NDFSM. However, it is quite simple to write down directly a (deterministic) FSM which accepts strings of the form required and that is the approach adopted here.

States q(0) (initial), q(1), q(2), FAIL. Accept state q(2). Arcs

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Labelled/Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>q(0)</td>
<td>q(1)</td>
<td>0</td>
</tr>
<tr>
<td>q(0)</td>
<td>FAIL</td>
<td>1</td>
</tr>
<tr>
<td>q(1)</td>
<td>q(0)</td>
<td>0</td>
</tr>
<tr>
<td>q(1)</td>
<td>q(2)</td>
<td>1</td>
</tr>
<tr>
<td>q(2)</td>
<td>FAIL</td>
<td>0+1</td>
</tr>
</tbody>
</table>

\( L^* \) consists of any concatenation of strings from \( L \) and by definition of * this includes the empty string \( \lambda \). Examples are

\( \lambda, 01, 0001, 000001, 000101, 010001, 00000101, 0101010001 \) etc.

A DFSM which accepts \( L^* \) is

States p(0) (initial), q(1), q(0), FAIL. Accept state p(0). Arcs

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Labelled/Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(0)</td>
<td>q(1)</td>
<td>0</td>
</tr>
<tr>
<td>p(0)</td>
<td>FAIL</td>
<td>1</td>
</tr>
</tbody>
</table>

\[\text{Figure 5-9 Solution to Q2.8(i).}\]

\[\text{Figure 5-10 Solution to Q2.8(ii).}\]
### Solutions for Chapter 3

1. **Definition of NDPDA:** A NDPDsa is a 6-tuple \( <Q, I, J, q(0), j(0), d> \) where

- \( Q \) is a finite set of states
- \( I \) is a finite set of input symbols
- \( J \) is a finite set of stack symbols
- \( q(0) \) in \( Q \) is an initial state
- \( j(0) \) in \( J \) is an initial stack symbol

\( d \) is a function which maps \( Q \times (I \cup \{\lambda\}) \times J \rightarrow \mathcal{P}(Q \times J^*) \), where \( \mathcal{P}(X) \) denotes the set of all subsets of \( X \).

The first five items of this definition are simple enough, however note that we assume the stack initially contains only the symbol \( j(0) \). The definition of the function \( d \) requires some explanation. The domain of the mapping consists of triples \( (q(t), i(t), j(t)) \), where \( q(t) \) is the current state, \( i(t) \) is an input symbol (which may be the null symbol \( \lambda \)), and \( j(t) \) is the current symbol on top of the stack. The range of \( d \) is the set of all subsets of \( Q \times J^* \). Recall that \( J^* \) is the set of all finite strings composed of symbols from \( J \). Thus an element \( (q(t+1), x) \) of \( Q \times J^* \) can describe the resulting state \( q(t+1) \), and the new string \( x \) on 'top' of the stack. We can imagine that the original symbol \( j(t) \) is popped from the stack and the whole string \( x \) is then pushed onto it. The new top of stack symbol \( j(t+1) \) then becomes the leftmost symbol of the string \( x \).

**Relationship:** Unlike FSM's, where each NDFSM can be replaced by an equivalent DFSM, for PDA's the class of NDPA's is genuinely larger than the class of DPDA's, i.e. NDPA's are more powerful than DPDA's.

**Chomsky class:** The NDPDA's are precisely the acceptors for the Context Free Languages (CFL's). This was proved by Chomsky in 1962.

**Definition CFG:** We first need to define a grammar.

**Definition.** A grammar consists of the following:

(a) A vocabulary \( V \) (as before), but this time split into two disjoint sets of symbols:

- A finite set of **terminal symbols** \( V(T) \). The elements of \( V(T) \) will normally be denoted by lower case letters \( a, b, \) etc. Terminal symbols are the symbols that actually appear in the strings of the language.

- A finite set of **non-terminal symbols** \( V(N) \). The elements of \( V(N) \) will normally be denoted by upper case letters \( A, B, \) etc. Non-terminal symbols do not actually appear in the strings of the language but are used by the production rules (see (b) below) to generate strings of terminals that are syntactically correct. Non-terminal symbols are often called **variables**.

(b) A finite set \( P \) of **production rules**. Each production is of the form \( x \rightarrow y \), where \( x \) and \( y \) are strings of symbols from \( V^* \), with \( x \) not equal to the empty string. Thus \( x \) and \( y \) are finite strings of variables or terminals. Elements of \( V^* \) will normally be denoted by lower case letters \( x, y, \) etc. The arrow symbol \( \rightarrow \) is read as 'may be replaced by'.
In fact this definition covers the most general type of grammar we shall consider, the phrase structured grammars, but we shall begin by considering some restricted special cases.

**Definition.** A grammar \( G = \langle V(N), V(T), P, S \rangle \) is context free if all productions in \( P \) are of the form \( A \rightarrow x \), where \( A \) is a variable and \( x \) a string of symbols from \( (V(N) \cup V(T))^* \).

**Construction:** Moves will be described in the following notation.

\[
\langle \text{state}, \text{input}, \text{stacktop} \rangle \rightarrow \langle \text{next_state}, \text{stack_action} \rangle
\]

Stack alphabet is \( \lambda \) (empty), 0 or 1 and stacktop will be one of these symbols.

stack_action is one of: POP, PUSHDOWN(alphabet symbol), or None.

The PDA is described by the moves

\[
\begin{align*}
\langle q(0), 0, \lambda \rangle & \rightarrow \langle q(0), \text{PUSH}(0) \rangle \\
\langle q(0), 1, \lambda \rangle & \rightarrow \langle q(0), \text{PUSH}(1) \rangle \\
\langle q(0), 0, 0 \rangle & \rightarrow \langle q(0), \text{PUSH}(0) \rangle \\
\langle q(0), 1, 1 \rangle & \rightarrow \langle q(0), \text{PUSH}(1) \rangle \\
\langle q(0), 0, 1 \rangle & \rightarrow \langle q(0), \text{POP} \rangle \\
\langle q(0), 1, 0 \rangle & \rightarrow \langle q(0), \text{POP} \rangle
\end{align*}
\]

Informally the idea is that the stack is empty if equal numbers of 0's and 1's have been received. If one symbol has predominated then that symbol will be on top of the stack, if the same symbol is now received then it will be pushed onto the stack (thereby keeping account of the amount by which it predominates) otherwise the other symbol is read in which case the stack is popped.

2. **First part (bookwork).** **Definition.** A NDPDSA is a 6-tuple \(< Q, I, J, q(0), j(0), d >\) where

\[
\begin{align*}
Q & \text{ is a finite set of states} \\
I & \text{ is a finite set of input symbols} \\
J & \text{ is a finite set of stack symbols} \\
q(0) & \text{ in } Q \text{ is an initial state} \\
j(0) & \text{ in } J \text{ is an initial stack symbol}
\end{align*}
\]

and \( d \) is a function which maps \( Q \times (I \cup \{ \lambda \}) \times J \rightarrow \mathcal{P}(Q \times J^*) \), where \( \mathcal{P}(X) \) denotes the set of all subsets of \( X \).

The first five items of this definition are simple enough, however note that we assume the stack initially contains only the symbol \( j(0) \). The definition of the function \( d \) requires some explanation. The domain of the mapping consists of triples \( (q(t), i(t), j(t)) \), where \( q(t) \) is the current state, \( i(t) \) is an input symbol (which may be the null symbol \( \lambda \)), and \( j(t) \) is the current symbol on top of the stack. The range of \( d \) is the set of all subsets of \( Q \times J^* \). Recall that \( J^* \) is the set of all finite strings composed of symbols from \( J \). Thus an element \( (q(t+1), x) \) of \( Q \times J^* \) can describe the resulting state \( q(t+1) \), and the new string \( x \) on 'top' of the stack. We can imagine that the original symbol \( j(t) \) is popped from the stack and the whole string \( x \) is then pushed onto it. The new top of stack symbol \( j(t+1) \) then becomes the leftmost symbol of the string \( x \).

The fact that the empty string \( \lambda \) is allowed as an input symbol permits the automaton to move under the current state without reference to the current input \( i(t) \) - an autonomous move of the form

\[
d: (q, \lambda, j) \rightarrow (q', x).
\]
Here, if \( x \) is the empty string the symbol below rises to the top of the stack. When an autonomous move, or a sequence of autonomous moves, is completed the input \( i(t) \) is then processed (it is not lost).

The 'non-deterministic' transitions are catered for in exactly the same way as for FSM's, namely by replacing \( Q \times J^* \) by the set of all subsets of \( Q \times J^* \).

**Second part.** A grammar which generates \( \mathcal{L} \) is

\[
S \rightarrow 0S0, \ S \rightarrow 1S1, \ S \rightarrow 00, \ S \rightarrow 11.
\]

The first two productions may be iterated any (finite) number of times to produce a centrally symmetric string, finally one of the last two productions must be used to eliminate \( S \) whilst preserving central symmetry. The grammar is a *Context Free* grammar since every production is of the form \( A \rightarrow x \), where \( A \) is a non-terminal symbol and \( x \) is a string of terminals and non-terminals (this is the definition of a CFG). Consequently \( L \) is a Context Free language and can therefore be accepted by some NDPDSA by a theorem of Chomsky (1962). To see why the extra power of a NDPDSA is actually necessary to parse such strings we begin by making two critical observations. Firstly, the automaton must read the input string in one pass. It cannot first count the length of the input string, from which the 'centre' point could be inferred, and then, on a second pass, push the first half of the string onto the stack and unstack this string symbol by symbol as the second half of the input string is read. Secondly, the stack is of arbitrary depth, whilst the FSM controller of the PDA has a fixed number of states. Therefore the algorithm must NOT require that the FSM absorb the entire stack contents at any stage. Plainly the essence of successful parsing is to locate the centre of the input string. However the second observation above makes this impossible for the DPDSA. The NDPDSA can accomplish this because we only require that there is SOME sequence of choices of autonomous moves which will lead to an empty stack when the end of the input string is reached. Consequently at each stage the NDPDSA can be granted the (non-deterministic) alternatives of EITHER assuming the string centre has been reached and then proceeding to unstack on this assumption, OR assuming that the centre has not been reached and then continuing to the next input symbol. Because of non-determinism we are not required to give an explicit mechanism for recovery in the event that the wrong choice is made, this is implicit in the definition of a NDPDSA which accepts by empty stack.

**Solutions for Chapter 4**

1. **Context Free Grammar:** Definition. A grammar \( G = \langle V(N), V(T), P, S \rangle \) is context free if all productions in \( P \) are of the form \( A \rightarrow x \), where \( A \) is a variable and \( x \) a string of symbols from \( (V(N) \cup V(T))^* \).

**Context Sensitive Grammar:** Definition. Let \( G = \langle V(N), V(T), P, S \rangle \) be a grammar. Suppose that for every production \( x \rightarrow y \) in \( P \) we have \( l(x) \leq l(y) \), where \( l(x) \) is the number of symbols in the string \( x \). Then \( G \) is context sensitive.

The language defined by a grammar \( G \) can be defined as the set of all strings \( L(G) \) such that: (i) the string consists solely of terminals (elements of \( V(T) \)), and (ii) the string can be derived from \( S \) (the start symbol) using the production rules of the grammar.

**Theorem** (Chomsky, 1962). The class of languages accepted by Non-deterministic Push Down Automata (NDPDA) is precisely the class of CFL's.

**Definition.** A Non-deterministic Linear Bounded Automaton (NDLBA) is a non-deterministic Turing machine which never leaves those cells on which the input was placed.

The interest in NDLBA's stems from the following fact

**Theorem** (Kuroda, 1964). The class of languages accepted by NDLBA's is precisely the class of Context Sensitive languages.

**Plausible argument:** We shall argue that a a push down stack automaton is inadequate and hence the language cannot
be Context Free. First, we recapitulate the argument that the language \( \{0^n1^n : n \geq 1 \} \) is not regular, but requires the extra power of a push down acceptor or regular language.

To recognise strings of the form \( 0^n1^n \) on a FSM acceptor, we must represent in the state of the machine the number of 0’s read in so far. Subsequently incoming 1’s can then be checked off against the 0’s. A FSM has only a finite capacity for counting 0’s and thus there is a limit to the maximum number \( n \) for which \( 0^n1^n \) can be recognised. A pushdown automaton can store the 0’s in its memory and unstack them one at a time as needed. It then ‘accepts’ a string by empty stack.

For \( 0^n1^n2^n \), the use of a push down stack is not adequate. Suppose a string has the form \( 0^r1^s2^t \). If the 0’s are stacked, the only way of discovering whether \( r = s \) is by unstacking a 0 every time a 1 is read. Thus the 0’s and 1’s are lost and we cannot check that \( r = t \). (A formal argument can be provided using the Pumping lemma for CFL’s - see Chapter 3.)

A Context Sensitive grammar which generates the language \( L \) is given by

1. \( S \rightarrow 0SBC \)
2. \( S \rightarrow 0BC \)
3. \( CBC \rightarrow CDC \)
4. \( CD \rightarrow BD \)
5. \( BD \rightarrow BC \)
6. \( 0B \rightarrow 01 \)
7. \( 1B \rightarrow 11 \)
8. \( 1C \rightarrow 12 \)
9. \( 2C \rightarrow 22 \)

Rules 1 and 2 generate strings \( 0^n(BC)^n \); then using rules 3 to 5 the string is rearranged as \( 0^nB^nC^n \), which is then converted to \( 0^i1^j2^k \) using rules 6-7 and 8-9. If the second stage is omitted or curtailed, the string will contain \( B \) in some context \( 2B \), and no move is defined for \( B \) in this situation.

2. Explain operation of Turing machine:

The basic model of a Turing machine has a FSM controller, an input tape (regarded as infinite in both directions) which is divided into cells, and a tape head which scans one cell at a time. In each operation cycle the machine starts in some state \( q(t) \), reads the symbol \( j(t) \) written on the cell under the head, prints the new symbol \( j(t+1) \), moves left (\( L \)) or right (\( R \)), and then enters the new state \( q(t+1) \). The alphabet of ‘tape symbols’ \( J \) is finite and includes a blank symbol, usually denoted by \( B \): when the Turing machine reads a blank symbol it halts.

**Define the term grammar:**

**Definition.** A grammar consists of the following:

(a) A vocabulary \( V \) (as before), but this time split into two disjoint sets of symbols:

A finite set of terminal symbols \( V(T) \). The elements of \( V(T) \) will normally be denoted by lower case letters \( a, b, \) etc. Terminal symbols are the symbols that actually appear in the strings of the language.

A finite set of non-terminal symbols \( V(N) \). The elements of \( V(N) \) will normally be denoted by upper case letters \( A, B, \) etc. Non-terminal symbols do not actually appear in the strings of the language but are used by the production rules (see (b) below) to generate strings of terminals that are syntactically correct. Non-terminal symbols are often called variables.

(b) A finite set \( P \) of production rules. Each production is of the form \( x \rightarrow y \), where \( x \) and \( y \) are strings of symbols from \( V^* \), with \( x \) not equal to the empty string. Thus \( x \) and \( y \) are finite strings of variables or terminals. Elements of \( V^* \) will normally be denoted by lower case letters \( x, y, \) etc. The arrow symbol ‘\( \rightarrow \)’ is read as ‘may be replaced by’.

(c) A start symbol \( S \), which is usually one of the \( V(N) \).
Phrase structured: the above definition of a grammar is that of a *phrase structured grammar* (being the most general we consider).

**Context Free**: A grammar is **Context Free** if all productions rules are of the form \( A \rightarrow y \), where \( A \) is a variable and \( y \) a string of symbols from \( V \).

**Syntax checker**: In a genuinely phrase structured language any grammar will have production rules of the type \( x \rightarrow y \), where \( l(x) > l(y) \). Without the safeguard (inherent in all the grammars lower in the Chomsky heirarchy) that \( l(x) \leq l(y) \) we do not know how many backward steps are required in parsing a given length string before abandoning the search. This means that syntactic strings will always (eventually) be accepted but there is no certainty that non-syntactic strings will be eventually rejected (the parsing may not terminate).

Informal description of solution machine:

The machine, starting as usual in state \( q(0) \), accomplishes the substitution of \( b \) for \( a \) and \( a \) for \( b \) by the quintuples

\[
q(0) \quad a \ b \ q(0) \ R
\]

(In state \( q(0) \), read \( a \), write \( b \), goto \( q(0) \), move Right)

\[
q(0) \quad b \ a \ q(0) \ R
\]

When the first \( c \) is reached, the machine enters state \( q(1) \) and continues to move right, changing \( a \) to \( b \) and \( b \) to \( a \), in accordance with the quintuples

\[
q(0) \quad c \ c \ q(1) \ R \\
q(1) \quad a \ b \ q(1) \ R \\
q(1) \quad a \ b \ q(1) \ R
\]

When the second \( c \), if any is reached, the machine has to move back along the tape to find and erase the first \( c \). This is done by the use of a third state \( q(2) \) and the quintuples

\[
q(1) \quad c \ c \ q(2) \ L \\
q(2) \quad a \ a \ q(2) \ L \\
q(2) \quad b \ b \ q(2) \ L \\
q(2) \quad c \ B \ q(3) \ R
\]

The machine now enters a fourth state \( q(3) \), after erasing the first \( c \). In state \( q(3) \) the machine moves to the right until it reaches the second \( c \). It then reverts to state \( q(1) \) and resumes the process of changing \( a \) to \( b \) and \( b \) to \( a \). This requires the quintuples

\[
q(3) \quad a \ a \ q(3) \ R \\
q(3) \quad b \ b \ q(3) \ R \\
q(3) \quad c \ c \ q(1) \ R
\]

Since none of its quintuples has the blank as its second member, the machine halts as soon as it comes to a blank on the tape.

**Solution to Exercise 4.1.**

Number the production rules in the order given 1 to 7.

In any derivation beginning with \( S \), until we use production 2 we cannot use 4 - 7, since each of productions 4 - 7 require a terminal immediately to the left of a \( B \) or \( C \). Until production 2 is used all strings derived consist of \( a \)'s
followed by an $S$, followed by $B$’s and $C$’s.

After 2 is used, the string consists of $n$ a’s, for some $n > 0$, followed by $n$ $B$’s and $n$ $C$’s in some order. Now no $S$’s appear in the string, so productions 1 and 2 may no longer be used. Note that the form of the string is all terminals followed by all variables. After applying any of productions 3 - 7, we see that the string will still have that property. Note that 4 - 7 are only applicable at the boundary between terminals and variables. Each has the effect of converting one $B$ to $b$ or one $C$ to $c$. Production 3 causes $B$’s to migrate to the left, and $C$’s to the right.

Suppose that a $C$ is converted to $c$ before all $B$’s are converted to $b$’s. Then the string can be written as $a^n b^i c x$, where $i < n$ and $x$ is a string of $B$’s and $C$’s, but not all $C$’s. Now, only productions 3 and 7 may be applied; 7 at the interface between terminals and variables, and 3 among the variables. We may use 3 to reorder the $B$’s and $C$’s of $x$, but not to remove any $B$’s. Production 7 can convert $C$’s to $c$’s at the interface, but eventually, a $B$ will be the leftmost variable. There is no production that can change the $B$, so this string can never result in a string with no variables.

We conclude that all $B$’s must be converted to $b$’s at the interface between terminals and variables before any $C$’s are converted to $c$’s. Thus, from $a^n$ followed by $n$ $B$’s and $n$ $C$’s in any order $a^n b^n c^n$ is the only derivable string.
## INDEX

- acceptor 6, 20
- acceptors 10
- Chomsky hierarchy 19
- closed 28
- closure 28
- compilation 41
- compiler 36
- compiler-compiler 32
- continuous state automaton 9
- decision problem 44
- derivation trees 41
- deterministic
  - FSM 8
  - PDSA 38
- Finite-State-Machine 9
- grammar 21, 64
- Hilbert 44
- Kurt Gödel 45, 63
- Markov chain 10
- NDPDSA 38
- next state vector 15
- Next state vector (NSV) 15
- non-deterministic
  - FSM 10
  - Turing machine 47
- non-deterministic FSM 10
- oracular model 11
- palindromes 39
- parse trees 41
- partial recursive 44
- phrase structured 44, 46
- phrase structured grammars
  - definition 21
- Push Down Stack automata 35
- Push Down Stack Automaton 11
- recursively enumerable 44
- regular expression 22
- regular grammar 24
- regular sets 28
- reset input 12
- right linear 24
- semantics 20
- SP-partition 17
- stack 35
- strongly connected 14
- substitution property 17
- syntax 20
- Turing machine 44, 46