

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

LESSON NOTES



Course with code: Theory of Computation-BCS503

Semester: 5th

Academic Year 2026-2027

Course Coordinators: Dr. ANIL KUMAR C J
Associate Professor & Head

MODULE-1

INTRODUCTION TO AUTOMATA THEORY

Automata theory is the study of abstract computing devices or “machines”. Before there were computers in the 1930’s, Allan Turing studied an abstract machine that had all the capabilities of today’s computers at least as far as in what they could compute Turing’s goal was to describe precisely the boundary between what a computing machine could do and what it could not do; his conclusions apply not only to his abstract Turing machines but to today’s real machines.

Why Study Automata Theory?

There are several reasons why the study of automata and complexity is an important part of the core of Computer Science.

Finite automata are a useful model for many important kinds of hardware and Software.

1. Software for designing and checking the behavior of digital circuits.
2. The “lexical analyzer” of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords and punctuation.
3. Software for scanning large bodies of text, such as collections of Web Pages, to find occurrences of words, phrases or other patterns.
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

Example: Perhaps the simplest nontrivial finite automaton is an on/off Switch. The device remembers whether it is in the “on” state or the “off” state, and it allows the user to press a button whose effect is different, depending on the state of the switch_ That is, if the switch is in the on state, then pressing the button changes it to the on state, and if the switch is in the on state, then pressing the same button turns it to the on state.

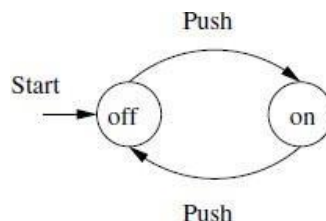


Figure 1.1 A finite automaton modelling an on/off switch

The Central Concepts of Automata Theory

Let us know the most important definitions of terms that pervade the theory of automata. These concepts include the “alphabet” (a set of symbols), “strings” (a list of symbols from an alphabet) and “language” (a set of strings from the same alphabet).

i) Alphabets:

An alphabet is a finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet.

Common alphabets include:

1. $\Sigma = \{0, 1\}$ the binary alphabet.
2. $\Sigma = \{a, b, \dots, z\}$, the set of all lower-case letters.
3. The set of all ASCII characters, or the set of all printable ASCII characters.

ii) Strings:

A string (or sometimes word) is a finite sequence of symbols chosen from some alphabet. For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$. The string 111 is another string chosen from this alphabet.

iii) The Empty String

The empty string is the string with zero occurrences of symbols. This string, denoted ϵ , is a string that may be chosen from any alphabet whatsoever.

iv) Length of a String

Length of a string is the number of positions for symbols in the string. Ex: 01101 has length 5. The standard notation for the length of a string w is $|w|$. For example, $|011|=3$ and $|\epsilon|=0$.

v) Powers of an Alphabet

If Σ is an Alphabet we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k each of whose symbols is in Σ .

Example: Note $\Sigma^0 = \{\epsilon\}$, regardless of what alphabet Σ is.

That is, ϵ is the only string whose length is 0.

If $\Sigma = \{0,1\}$, then

$$\Sigma^1 = \{0,1\},$$

$$\Sigma^2 = \{00,01,10,11\}$$

$$\Sigma^3 = \{000,001,010,011,100,101,110,111\} \text{ and so on.}$$

The set of all strings over an alphabet Σ is conventionally denoted Σ^* .

For instance, $\{0, 1\}^* = \{ \epsilon, 0, 1, 00, 01, 10, 11, 000, \dots \}$

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty strings from alphabet Σ is denoted Σ^+ . Thus, two appropriate equivalences are:

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

vi) Concatenation of Strings

Let x and y be strings. Then xy denotes the concatenation of x and y , that is, the string formed by making a copy of x and following it by a copy of y . More precisely, if x is the string composed of i symbols $x = a_1a_2\dots a_i$ and y is the string composed of j symbols $y = b_1b_2\dots b_j$, then xy is the string of length $i+j$: $xy = a_1a_2\dots a_i b_1b_2\dots b_j$.

vii) Languages

A set of strings all of which are chosen from some Σ^* , where Σ is a particular Alphabet, is called a language. If Σ is an alphabet, and L then L is a language over Σ .

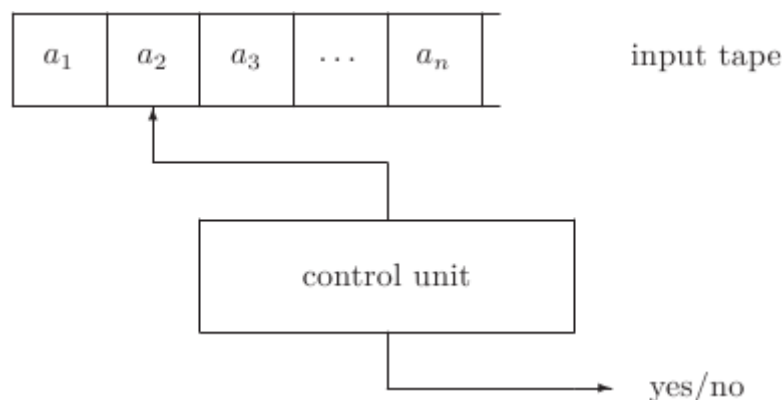
Examples:

1. The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0 : \{\epsilon, 01, 0011, 000111, \dots\}$
2. The set of strings of 0's and 1's with an equal number of each:
 $\{\epsilon, 01, 10, 0011, 0101, 1001, \dots\}$
3. The set of binary numbers whose value is a prime:
 $\{10, 11, 101, 111, 1011, \dots\}$
4. Σ^* is a language for any alphabet Σ .

5. Φ , the empty language, is a language over any alphabet.
6. $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Note that $\Phi \neq \{\epsilon\}$; the former has no strings and the latter has one string.

FINITE AUTOMATA

A finite automaton is a mathematical model which is used to study the abstract machines or abstract computing devices with inputs chosen from Σ , where Σ stands for set of alphabets using which any string can be obtained. On reading the string, the machine may accept or reject the string.



There mainly 3 different types of FA,

1. Deterministic Finite Automata
2. Non-deterministic Finite Automata
3. Non-deterministic finite automata with ϵ -moves (ϵ -NFA)

1. Deterministic Finite Automata (DFA):

The term 'deterministic' refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state.

Definition of a Deterministic Finite Automaton(2M):

A deterministic finite is defined by a quintuple as, $A=(Q, \Sigma, \delta, q_0, F)$ where,

- Q is a finite set of states.
- Σ is a finite non-empty set of symbols called input alphabet
- δ is a transition function that takes arguments a state and an input symbol and returns a state.
If q is a state, and a is an input symbol, then $\delta(q,a)$ is that state p such that there is an arc labelled a from q to p .

$$\delta : Q \times \Sigma \rightarrow Q$$

- q_0 is the start state, one of the states in Q i.e. $q_0 \in Q$
- F is a set of final or accepting states. The set F is a subset of Q .

How a DFA Processes Strings: Initially the DFA is in state q_0 . The DFA reads the input string from left to right one symbol at a time. The changes to state of DFA is dictated by δ . When the last symbol of input string is read, if this results in the DFA reaching any of its final state, we say that the string is accepted by DFA, otherwise the string is rejected.

Simpler Notations for DFA's: There are two preferred notations for describing automata.

- a) **A transition diagram**, which is a graph.
- b) **A transition table**, which is a tabular listing of the function δ which by implication tells us the set of states and the input alphabet.

a) Transition Diagrams: A transition diagram for a DFA $A=(Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

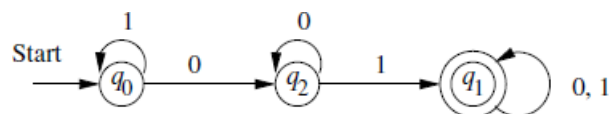
- For each state in Q there is a node
- For each state q in Q and each input symbol a in Σ , let $\delta(q, a)=p$. Then the transition diagram has an arc from node q to node p , labelled a . If there are several input symbols that cause transitions from q to p , then the transition diagram can have one arc, labeled by the list of these symbols.
- There is an arrow into the start state q_0 labeled *Start*. This arrow does not originate at any node.
- Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Ex: If $A=(Q, \Sigma, \delta, q_0, F)$ defined as $A=(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$

where $\delta = \{ \delta(q_0,0)=q_2,$

$$\begin{aligned}
\delta(q_0, 1) &= q_0, \\
\delta(q_1, 0) &= q_1, \\
\delta(q_1, 1) &= q_1, \\
\delta(q_2, 0) &= q_2, \\
\delta(q_2, 1) &= q_1 \}
\end{aligned}$$

The transition diagram is,



Transition Tables:

A transition table is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$. The transition table for the above example is:

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Extended Transition Function:

We define an extended transition function that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$.

“The extended transition function is a function that takes a state q and a string w and returns a state p - the state that the automaton reaches when starting in state q and processing the sequence of inputs w ”.

We define $\hat{\delta}$ by induction on the length of the input string, as follows:

BASIS: $\hat{\delta}(q, \epsilon) = q$. That is, if we are in state q and read no inputs, then we are still in state q .

INDUCTION: Suppose w is a string of the form xa ; that is, a is the last symbol of w , and x is the string consisting of all but the last symbol.³ For example, $w = 1101$ is broken into $x = 110$ and $a = 1$. Then

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \quad (2.1)$$

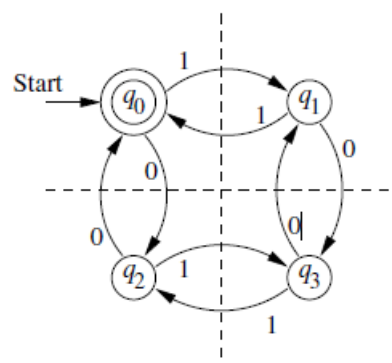
Example 2.4: Let us design a DFA to accept the language

$$L = \{w \mid w \text{ has both an even number of 0's and an even number of 1's}\}$$

It should not be surprising that the job of the states of this DFA is to count both the number of 0's and the number of 1's, but count them modulo 2. That is, the state is used to remember whether the number of 0's seen so far is even or odd, and also to remember whether the number of 1's seen so far is even or odd. There are thus four states, which can be given the following interpretations:

- q_0 : Both the number of 0's seen so far and the number of 1's seen so far are even.
- q_1 : The number of 0's seen so far is even, but the number of 1's seen so far is odd.
- q_2 : The number of 1's seen so far is even, but the number of 0's seen so far is odd.
- q_3 : Both the number of 0's seen so far and the number of 1's seen so far are odd.

State q_0 is both the start state and the lone accepting state. It is the start state, because before reading any inputs, the numbers of 0's and 1's seen so far are both zero, and zero is even. It is the only accepting state, because it describes exactly the condition for a sequence of 0's and 1's to be in language L .



Therefore DFA $A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$ where δ is described by the transition diagram as shown above.

Suppose the input is 110101. Since this string has even numbers of 0's and 1's both, we expect it is in the language. Thus, we expect that $\hat{\delta}(q_0, 110101) = q_0$ since q_0 is the only accepting state.

The check involves computing $\hat{\delta}(q_0, w)$ for each prefix w of 110101, starting at ϵ and going in increasing size. The summary of this calculation is:

- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$.

The Language of a DFA:

Now, we can define the *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$. This language is denoted $L(A)$, and is defined by

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$$

That is, the language of A is the set of strings w that take the start state q_0 to one of the accepting states. If L is $L(A)$ for some DFA A , then we say L is a *regular language*.

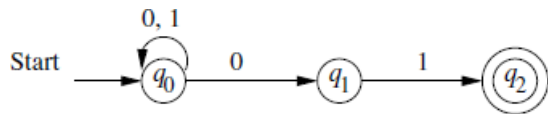
2. Nondeterministic Finite Automata (NFA):

A “nondeterministic finite automaton” (NFA) has the power to be in several states at once. This ability is often expressed as an ability to “guess” something about its input.

Like the DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. It also has a transition function, which we shall commonly call δ . The difference between the DFA and the NFA is in the type of δ . For the NFA δ is a function that takes a state and input symbol as arguments but returns a set of zero, one, or more states, rather than returning exactly one state as the DFA must.

Ex: Let us design a NFA to accept all and only the strings of 0's and 1's that end in **01**.

Here the possible strings are, $L = \{01, 101, 001, 11101, 101001, \dots\}$



Here the NFA has the option of going either to q_0 or to q_1 and in fact it does both, as we shall see when we make the definitions precise. In state q_1 the NFA checks that the next symbol is 1 and if so, it goes to state q_2 and accepts. Notice that there is no arc out of q_1 labeled 0, and there are no arcs at all out of q_2 .

Definition of Nondeterministic Finite Automata:

An NFA is represented as a quintuple,

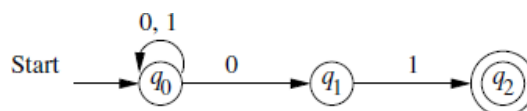
$$A = (Q, \Sigma, \delta, q_0, F)$$

where:

1. Q is a finite set of *states*.
2. Σ is a finite set of *input symbols*.
3. q_0 , a member of Q , is the *start state*.
4. F , a subset of Q , is the set of *final* (or *accepting*) states.
5. δ , the *transition function* is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q . Notice that the only difference between an NFA and a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

The transition table for given NFA can be represented as,



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

The Extended Transition Function for NFA:

As for DFA's, we need to extend the transition function δ of an NFA to a function $\hat{\delta}$ that takes a state q and a string of input symbols w , and returns the set of states that the NFA is in if it starts in state q and processes the string w .

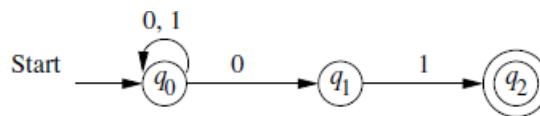
BASIS: $\hat{\delta}(q, \epsilon) = \{q\}$. That is, without reading any input symbols, we are only in the state we began in.

INDUCTION: Suppose w is of the form $w = xa$, where a is the final symbol of w and x is the rest of w . Also suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \dots, p_k\}$. Let

$$\bigcup_{i=1}^k \delta(p_i, a) = \{r_1, r_2, \dots, r_m\}$$

Then $\hat{\delta}(q, w) = \{r_1, r_2, \dots, r_m\}$. Less formally, we compute $\hat{\delta}(q, w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labeled a .

Example: Let us trace the input 00101 by the NFA to accept strings ending with 01.



1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$.
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Since we reached the end of the string and automata is in states $\{q_0, q_2\}$ where q_2 is the final state, the given string is accepted.

The Language of an NFA:

An NFA accepts a string w if it is possible to make any sequence of choices of next state, while reading the characters of w , and go from the start state to any accepting state.

Formally, if $A=(Q, \Sigma, \delta, q_0, F)$, then,

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

That is, $L(A)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

Equivalence of Deterministic and Nondeterministic Finite Automata

- Although there are many languages for which an NFA is easier to construct than a DFA, it is a surprising fact that every language that can be described by some NFA can also be described by some DFA.
- Moreover, the DFA in practice has about as many states as the NFA, although it often has more transitions. In the worst case, however, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.
- The proof that DFA's can do whatever NFA's can do involves an important "construction" called the **subset construction** because it involves constructing all subsets of the set of states of the NFA. In general, many proofs about automata involve constructing one automaton from another.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. Notice that the input alphabets of the two automata are the same, and the start state of D is the set containing only the start state of N . The other components of D are constructed as follows.

- Q_D is the set of subsets of Q_N ; i.e., Q_D is the *power set* of Q_N . Note that if Q_N has n states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state of Q_D . Inaccessible states can

be “thrown away,” so effectively, the number of states of D may be much smaller than 2^n .

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N 's states that include at least one accepting state of N .
- For each set $S \subseteq Q_N$ and for each input symbol a in Σ ,

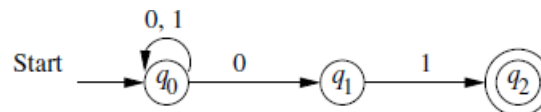
$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S , see what states N goes to from p on input a , and take the union of all those states.

Example:

Let us convert the following NFA to DFA using subset construction method.

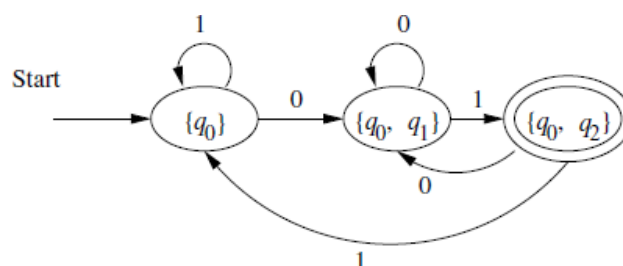
Given NFA,



Since N 's set of states is $\{q_0, q_1, q_2\}$ the subset construction produces a DFA with $2^3=8$ states, corresponding to all the subsets of these three states. Following figure shows the transition table for these eight states.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

The DFA constructed from the NFA is,



3. Finite Automata with Epsilon-Transitions:

An NFA that is allowed to make a transition spontaneously, without receiving an input symbol is called ϵ -NFA.

Ex: Let us design a ϵ -NFA that accepts **decimal numbers** consisting of:

1. An optional + or – sign
2. A string of digits
3. A decimal point and
4. Another string of digits. Either this string of digits or the string (2) can be empty, but both cannot be empty at the same time.

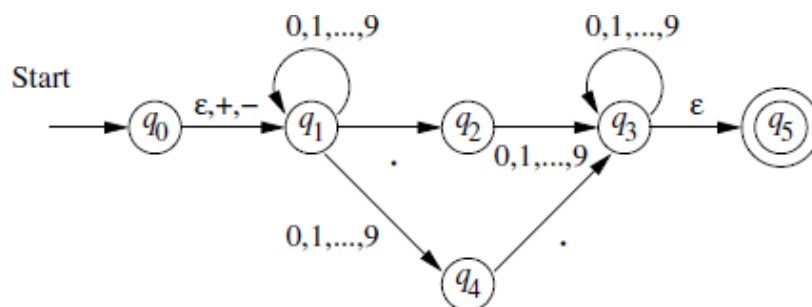


Figure: An ϵ -NFA accepting decimal numbers

The Formal Notation for an ϵ -NFA:

An ϵ -NFA may be represented exactly as we do an NFA with one exception: the transition function must include information about transition on ϵ .

Definition: Formally ϵ -NFA is defined by,

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q is a finite set of states.
- Σ is a finite non-empty set of symbols called input alphabet
- δ is a transition function that takes arguments a state and an input symbol or the symbol ϵ and returns zero or one or more states.

$$\delta : Q \times \Sigma \cup \{ \epsilon \} \rightarrow 2^Q$$

- q_0 is the start state, one of the states in Q i.e. $q_0 \in Q$
- F is a set of final or accepting states. The set F is a subset of Q .

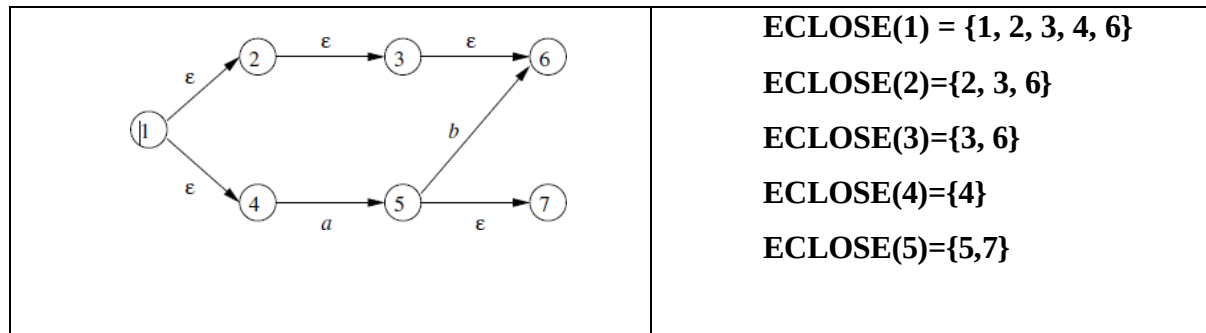
Epsilon-Closures:

The ϵ -closure of any state q denoted by $ECLOSE(q)$ is the set of all states which are reachable from q on ϵ transitions only. Formally ϵ -closure $ECLOSE(q)$ recursively, as follows:

BASIS: State q is in $ECLOSE(q)$.

INDUCTION: If state p is in $ECLOSE(q)$, and there is a transition from state p to state r labeled ϵ , then r is in $ECLOSE(q)$. More precisely, if δ is the transition function of the ϵ -NFA involved, and p is in $ECLOSE(q)$, then $ECLOSE(q)$ also contains all the states in $\delta(p, \epsilon)$.

Example:



Extended Transitions and Languages for ϵ -NFA's:

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ -NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w . As always, ϵ 's along this path do not contribute to w . The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) = ECLOSE(q)$. That is, if the label of the path is ϵ , then we can follow only ϵ -labeled arcs extending from state q ; that is exactly what $ECLOSE$ does.

INDUCTION: Suppose w is of the form xa , where a is the last symbol of w . Note that a is a member of Σ ; it cannot be ϵ , which is not in Σ . We compute $\hat{\delta}(q, w)$ as follows:

1. Let $\{p_1, p_2, \dots, p_k\}$ be $\hat{\delta}(q, x)$. That is, the p_i 's are all and only the states that we can reach from q following a path labeled x . This path may end

with one or more transitions labeled ϵ , and may have other ϵ -transitions, as well.

2. Let $\bigcup_{i=1}^k \delta(p_i, a)$ be the set $\{r_1, r_2, \dots, r_m\}$. That is, follow all transitions labeled a from states we can reach from q along paths labeled x . The r_j 's are *some* of the states we can reach from q along paths labeled w . The additional states we can reach are found from the r_j 's by following ϵ -labeled arcs in step (3), below.
3. Then $\hat{\delta}(q, w) = \text{ECLOSE}(\{r_1, r_2, \dots, r_m\})$. This additional closure step includes all the paths from q labeled w , by considering the possibility that there are additional ϵ -labeled arcs that we can follow after making a transition on the final "real" symbol, a .

Language of an ϵ -NFA $E=(Q, \Sigma, \delta, q_0, F)$ is defined as follows:

$$L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Eliminating ϵ -transitions: Converting ϵ -NFA to DFA

Given any ϵ -NFA E , we can find a DFA D that accepts the same language as E . The construction we use is very close to the subset construction, as the states of D are subsets of the states of E . The only difference is that we must incorporate ϵ -transitions of E , which we do through the mechanism of the ϵ -closure.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E . More precisely, we shall find that all accessible states of D are ϵ -closed subsets of Q_E , that is, sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$. Put another way, the ϵ -closed sets of states S are those such that any ϵ -transition out of one of the states in S leads to a state that is also in S . Note that $\emptyset$ is an ϵ -closed set.
2. $q_D = \text{ECLOSE}(q_0)$; that is, we get the start state of D by closing the set consisting of only the start state of E . Note that this rule differs from the original subset construction, where the start state of the constructed automaton was just the set containing the start state of the given NFA.
3. F_D is those sets of states that contain at least one accepting state of E . That is, $F_D = \{S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.
4. $\delta_D(S, a)$ is computed, for all a in Σ and sets S in Q_D by:
 - (a) Let $S = \{p_1, p_2, \dots, p_k\}$.
 - (b) Compute $\bigcup_{i=1}^k \delta_E(p_i, a)$; let this set be $\{r_1, r_2, \dots, r_m\}$.
 - (c) Then $\delta_D(S, a) = \text{ECLOSE}(\{r_1, r_2, \dots, r_m\})$.

Example: Convert the given NFA to DFA

	ϵ	a	b	c
$\rightarrow p$	$\{q, r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Ans:

Let us find **ECLOSE** of all the states of given NFA,

$$\mathbf{ECLOSE}(p) = \{p, q, r\}$$

$$\mathbf{ECLOSE}(q) = \{q\}$$

$$\mathbf{ECLOSE}(r) = \{r\}$$

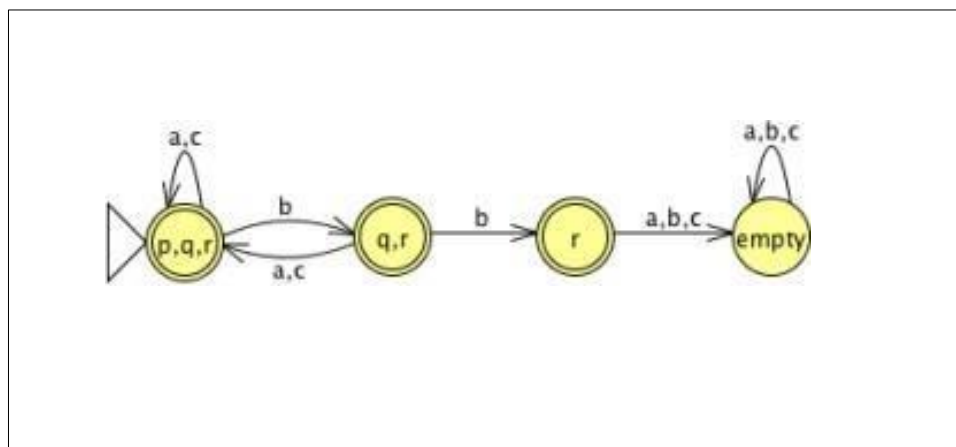
Step 1: Since p is the start state of the given NFA, start state of DFA is,

$$\mathbf{ECLOSE}(p) = \{p, q, r\}$$

Step 2: Find transition function of the DFA as follows (use subset-construction method) :

	a	b	c
$\rightarrow * \{p, q, r\}$	$\{p, q, r\}$	$\{q, r\}$	$\{p, q, r\}$
$* \{q, r\}$	$\{p, q, r\}$	$\{r\}$	$\{p, q, r\}$
$* \{r\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

The transition diagram of the DFA is,



An Application of Finite Automata: Text Search

1. Finding Strings in Text:

A common problem in the age of the Web and other online text repositories is the following: Given a set of words, and all documents that contain one (or all) of those words. A search engine is a popular example of this process.

The search engine uses a particular technology, called *inverted indexes*, where for each word appearing on the Web (there are 100,000,000 different words), a list of all the places where that word occurs is stored. Machines with very large amounts of main memory keep the most common of these lists available, allowing many people to search for documents at once.

Inverted-index techniques do not make use of finite automata, but they also take very large amounts of time for crawlers to copy the Web and set up the indexes. There are a number of related applications that are unsuited for inverted indexes, but are good applications for automaton-based techniques. The characteristics that make an application suitable for searches that use automata are:

1. The repository on which the search is conducted is rapidly changing. For example:
 - (a) Every day, news analysts want to search the day's online news articles for relevant topics. For example, a financial analyst might search for certain stock ticker symbols or names of companies.
 - (b) A "shopping robot" wants to search for the current prices charged for the items that its clients request. The robot will retrieve current catalog pages from the Web and then search those pages for words that suggest a price for a particular item.
2. The documents to be searched cannot be catalogued. For example, Amazon.com does not make it easy for crawlers to find all the pages for all the books that the company sells. Rather, these pages are generated "on the fly" in response to queries. However, we could send a query for books on a certain topic, say "finite automata" and then search the pages retrieved for certain words, e.g., "excellent" in a review portion.

2. Nondeterministic Finite Automata for Text Search:

Suppose we are given a set of words, which we shall call the keywords, and we want to find occurrences of any of these words. In applications such as these. A useful way to proceed is to design a nondeterministic finite automaton, which signals, by entering an accepting state, that it has seen one of the keywords.

The text of a document is fed, one character at a time to this NFA, which then recognizes occurrences of the keywords in this text. There is a simple form to an NFA that recognizes a set of keywords.

1. There is a start state with a transition to itself on every input symbol, e.g. every printable ASCII character if we are examining text. Intuitively, the start state represents a “guess” that we have not yet begun to see one of the keywords, even if we have seen some letters of one of these words.
2. For each keyword $a_1a_2 \cdots a_k$, there are k states, say $q_1, q_2, \dots, q_k$. There is a transition from the start state to q_1 on symbol a_1 , a transition from q_1 to q_2 on symbol a_2 , and so on. The state q_k is an accepting state and indicates that the keyword $a_1a_2 \cdots a_k$ has been found.

Example 2.14: Suppose we want to design an NFA to recognize occurrences of the words **web** and **ebay**. The transition diagram for the NFA designed using the rules above is in Fig. 2.16. State 1 is the start state, and we use Σ to stand for the set of all printable ASCII characters. States 2 through 4 have the job of recognizing **web**, while states 5 through 8 recognize **ebay**. $\square$

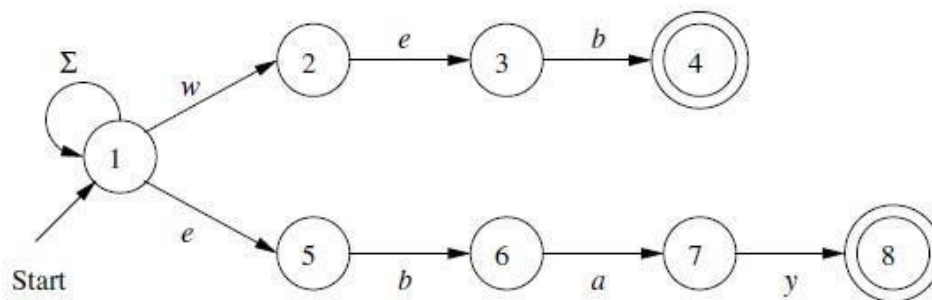


Figure 2.16: An NFA that searches for the words **web** and **ebay**

3. A DFA to Recognize a Set of Keywords:

The rules for constructing the set of DFA states is as follows:

- a) If q_0 is the start state of the NFA, then $\{q_0\}$ is one of the states of the DFA.
- b) Suppose p is one of the NFA states, and it is reached from the start state along a path whose symbols are $a_1a_2 \cdots a_m$. Then one of the DFA states is the set of NFA states consisting of:
 1. q_0 .
 2. p .
 3. Every other state of the NFA that is reachable from q_0 by following a path whose labels are a suffix of $a_1a_2 \cdots a_m$, that is, any sequence of symbols of the form $a_ja_{j+1} \cdots a_m$.

The DFA to search keywords **web** and **ebay** is:

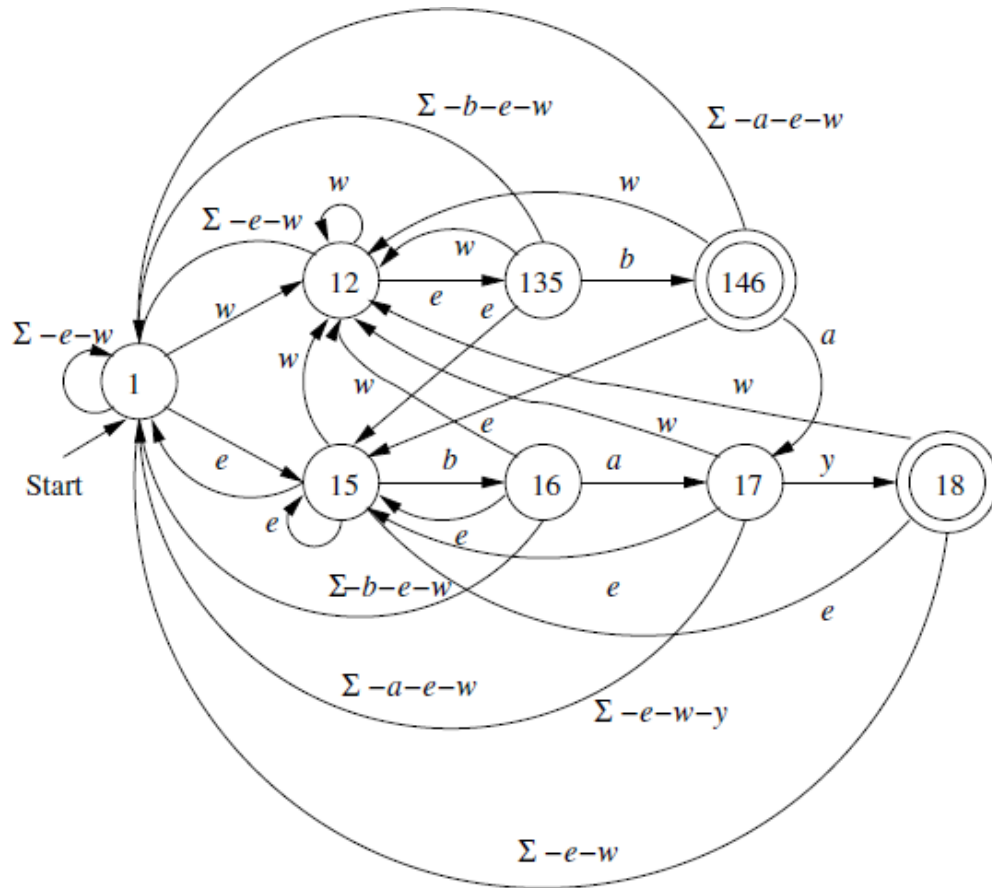


Figure 2.17: Conversion of the NFA from Fig. 2.16 to a DFA

Note: For exercises/problems under this module- refer class work.

MODULE-2

REGULAR EXPRESSIONS AND LANGUAGES

Regular Expressions:

Regular expressions are another type of language-defining notation. We shall find that regular expressions can define exactly the same languages that the various forms of automata (DFA and NFA) describe: the regular languages. Regular expressions are closely related to NFA and can be thought of as a “user-friendly” alternative to the NFA notation for describing regular languages.

We describe regular language through the notation of R. E. This notation involves a combination of strings or symbols from some alphabet Σ , (,), operators like +, . , *.

In R. E notation we use 3 operations on languages that the operators of R. E represent.

- 1. Union:** The union of two regular languages, L_1 and L_2 , which are represented using $L_1 \cup L_2$, is also regular and which represents the set of strings that are either in L_1 or L_2 or both.

Example:

$L_1 = \{00, 10, 11, 01\}$ and

$L_2 = \{\epsilon, 100\}$

then $L_1 \cup L_2 = \{\epsilon, 00, 10, 11, 01, 100\}$.

2. Concatenation:

The concatenation of two regular languages, L_1 and L_2 , which are represented using $L_1 \cdot L_2$ is also regular and which represents the set of strings that are formed by taking any string in L_1 concatenating it with any string in L_2 .

Example:

$L_1 = \{0, 1\}$ and $L_2 = \{00, 11\}$ then $L_1 \cdot L_2 = \{000, 011, 100, 111\}$.

3. The closure or star or Kleene closure:

If L is a regular language, then the Kleene closure i.e. L^* of L_1 is also regular and represents the set of those strings which are formed by taking a number of strings from L_1 and the same string can be repeated any number of times and concatenating those strings.

Example:

$L_1 = \{0,1\}$ then L^* is all strings possible with symbols 0 and 1, including a null string.

$L^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 00101, 110011, \dots\}$

Definition of Regular Expressions and the Languages associated with R. E:

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.
2. If a is any symbol, then **a** is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that **a** refers to a , should be obvious.
3. A variable, usually capitalized and italic such as L , is a variable, representing any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E + F) = L(E) \cup L(F)$.
 2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.
 3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = (L(E))^*$.
 4. If E is a regular expression, then (E) , a parenthesized E , is also a regular expression, denoting the same language as E . Formally; $L((E)) = L(E)$.
-

Precedence of Regular Expression Operators:

Like other algebras, the regular-expression operators have an assumed order of “precedence”, which means that operators are associated with their operands in a particular order.

1. The star operator is of highest precedence
2. Next in precedence comes the concatenation or dot operator
3. The union or ‘+’ has least precedence.

Ex: $01^* + 1$ can be grouped as, $(0(1^*)) + 1$

Examples:

Finding RE for the given Language:

1. Let $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$.

$L = \{aa, ab, abba, aabb, ba, baabaa, \dots\}$

Then, RE = $((a + b)(a + b))^*$

or

RE = $(aa + ab + ba + bb)^*$

2. Let $L = \{w \in \{a, b\}^* : w \text{ starting with string } abb\}$.

$L = \{abb, abba, abbb, abbab, \dots\}$

Then, RE = $abb(a + b)^*$

3. Let $L = \{w \in \{a, b\}^* : w \text{ ending with string } abb\}$.

$L = \{abb, aabb, babb, ababb, \dots\}$

RE = $(a + b)^*abb$

4. $L = \{w \in \{0, 1\}^* : w \text{ have } 001 \text{ as a substring}\}$.

$L = \{001, 1001, 000101, \dots\}$

RE = $(0 + 1)^*001(0 + 1)^*$

5. $L = \{w \in \{0, 1\}^* : w \text{ does not have } 001 \text{ as a substring}\}$.

$L = \{0, 1, 010, 110, 101, \dots\}$

RE = $(1 + 01)^*0^*$

6. $L = \{w \in \{a, b\}^* : w \text{ contains an odd number of a's}\}.$

$L = \{a, aaa, ababa, bbaaaaba, \dots\}$

RE = $b^*(ab^*ab^*)^* a b^*$ or $b^*ab^*(ab^*ab^*)^*$

7. $L = \{w \in \{a, b\}^* : \#a(w) \bmod 3 = 0\}.$

$L = \{aaa, abbaba, baaaaaa, \dots\}$

RE = $(b^*ab^*ab^*a)^*b^*$

8. Let $L = \{w \in \{a, b\}^* : \#a(w) \leq 3\}.$

$L = \{a, aa, ba, aaab, bbbabb, \dots\}$

RE = $b^*(a + \epsilon)b^*(a + \epsilon)b^*(a + \epsilon)b^*$

9. $L = \{w \in \{0, 1\}^* : w \text{ contains no consecutive 0's}\}$

$L = \{0, \epsilon, 1, 01, 10, 1010, 110, 101, \dots\}$

RE = $(0 + \epsilon)(1 + 10)^*$

10. $L = \{w \in \{0, 1\}^* : w \text{ contains at least two 0's}\}$

$L = \{00, 1010, 1100, 0001, 1010, 100, 000, \dots\}$

RE = $(0 + 1)^*0(0 + 1)^*0(0 + 1)^*$

11. $L = \{a^n b^m / n \geq 4 \text{ and } m \leq 3\}$

RE = $(aaaa)a^*(\epsilon + b + bb + bbb)$

12. $L = \{a^n b^m / n \leq 4 \text{ and } m \geq 2\}$

RE = $(\epsilon + a + aa + aaa + aaaa)bb(b)^*$

13. $L = \{a^{2n} b^{2m} / n \geq 0 \text{ and } m \geq 0\}$

RE = $(aa)^*(bb)^*$

14. $L = \{a^n b^m : (m+n) \text{ is even}\}$

Since $(m+n)$ is even when both a's and b's are even or both odd.

RE = $(aa)^*(bb)^* + a(aa)^*b(bb)^*$

15. $L = \{w : w \in \{a, b\}^* \text{ and } n_a(w) \& n_b(w) \text{ is even}\}$

RE = $(aa+bb)^* + ((ab+ba)(aa+bb)^*(ab+ba))^*$

- For odd number of a's and even number of b's we need one extra a at any position in the above expression.

RE = $(aa+bb)^* b (aa+bb)^* +$

$((ab+ba)(aa+bb)^*(ab+ba))^* b ((ab+ba)(aa+bb)^*(ab+ba))^*$

Finite Automata and Regular Expressions:

We know that DFA, NFA and ϵ -NFA accepts the same class of languages. In order to show that R. E. defines the same class, we must show that:

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.
2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an ϵ -NFA with transitions accepting the same language.

From DFA's to Regular Expressions:

Converting DFA's to Regular Expressions by Eliminating States:

Given a DFA with its set of states and transitions. When we eliminate a state s , all the paths that went through s no longer exist in the automaton. If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p , the labels of paths that went from some state q to state p through s . Since the label of this arc may now involve strings, rather than single symbols and there may even be an infinite number of such strings, we cannot simply list the strings as a label. Fortunately, there is a simple finite way to represent all such strings, use a regular expression.

Method for constructing R. E from FA.

1. For each accepting state q_f apply the above reduction process to produce an equivalent automaton with RE labels on the arcs. Eliminate all states except q_f and start state q_0 .
 2. If q_f is not equal to q_0 , then we are left with two state automaton.
-

3. If $q_f = q_0$ start state is also final state, then we must perform state elimination so that we are left with one state automata.
4. The desired RE is the sum or union of all the expressions derived from the reduced automaton for each accepting state by rules (2) and (3).

Example:

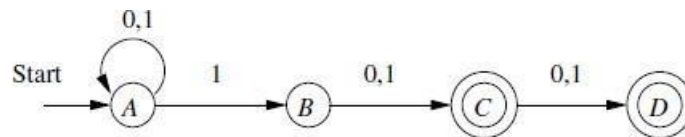


Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

Ans:

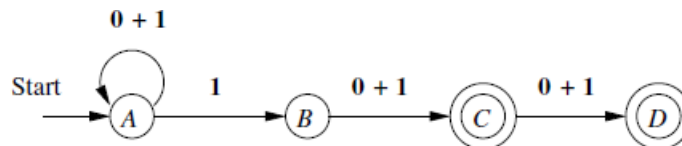


Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

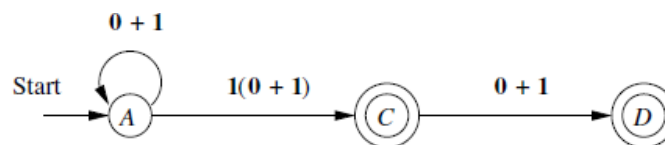


Figure 3.13: Eliminating state B

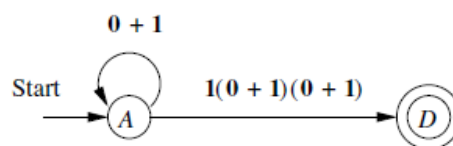


Figure 3.14: A two-state automaton with states A and D

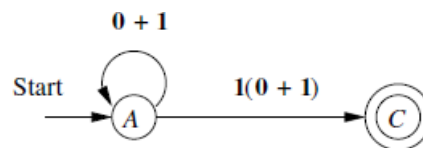


Figure 3.15: Two-state automaton resulting from the elimination of D

Therefore the final RE is: $(0+1)^*1(0+1)+(0+1)^*1(0+1)(0+1)$

Converting Regular Expressions to Automata:

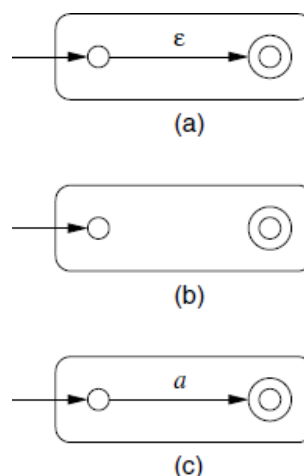
Prove that every language defined by a regular expression is also defined by a finite automaton(6M)

Theorem 3.7: Every language defined by a regular expression is also defined by a finite automaton.

PROOF: Suppose $L = L(R)$ for a regular expression R . We show that $L = L(E)$ for some ϵ -NFA E with:

1. Exactly one accepting state.
2. No arcs into the initial state.
3. No arcs out of the accepting state.

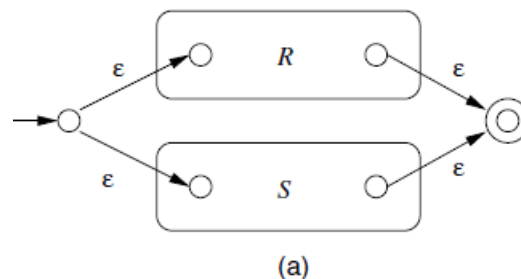
BASIS: There are three parts to the basis, shown in Fig. 3.16. In part (a) we see how to handle the expression ϵ . The language of the automaton is easily seen to be $\{\epsilon\}$, since the only path from the start state to an accepting state is labeled ϵ . Part (b) shows the construction for $\emptyset$. Clearly there are no paths from start state to accepting state, so $\emptyset$ is the language of this automaton. Finally, part (c) gives the automaton for a regular expression a . The language of this automaton evidently consists of the one string a , which is also $L(a)$. It



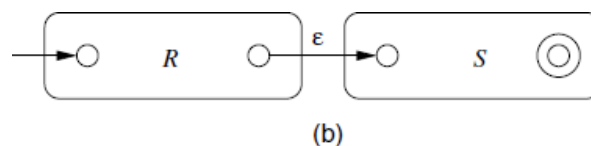
Induction:

The three parts of the induction are shown in Fig. 3.17. We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expression that is, the languages of these subexpressions are also the languages of ϵ -NFA's with a single accepting state. The four cases are:

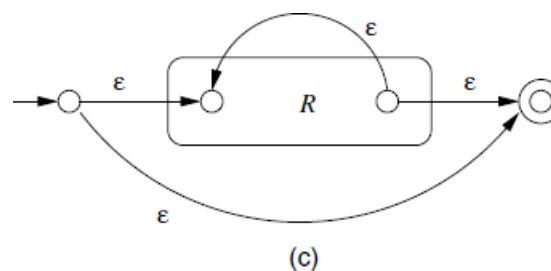
1. The expression is $\mathbf{R+S}$ for some smaller expressions R and S . Then the automaton of Fig. (a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for R or the automaton for S . We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R)$ or $L(S)$ respectively. Thus, the language of the automaton is $\mathbf{L(R) \cup L(S)}$.



2. The expression is $\mathbf{RS}$ for some smaller expressions R and S . The automaton for the concatenation is shown in Fig. (b). The idea is that the only paths from start to accepting state go first through the automaton for R , where it must follow a path labeled by a string in $L(R)$, and then through the automaton for S , where it follows a path labeled by a string in $L(S)$. Thus, the paths in the automaton of Fig. (b) are all and only those labeled by strings in $L(R)L(S)$.



3. The expression is R^* for some smaller expression R . Then we use the automaton of Fig. (c). That automaton allows us to go either:
 - a) Directly from the start state to the accepting state along a path labeled ϵ . That path lets us accept ϵ , which is in $L(R^*)$ no matter what expression R is.
 - b) To the start state of the automaton for R , through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in $L(R^*)$.

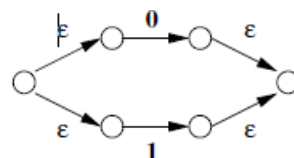


4. The expression is (R) for some smaller expression R . The automaton for R also serves as the automaton for (R) , since the parentheses do not change the language defined by the expression.

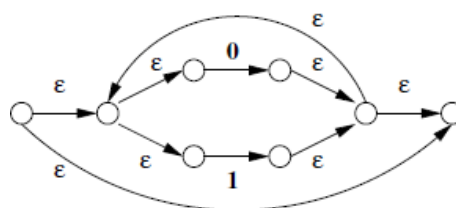
Example:

Let us convert the regular expression $(0+1)^* 1(0+1)$ to an ϵ -NFA.

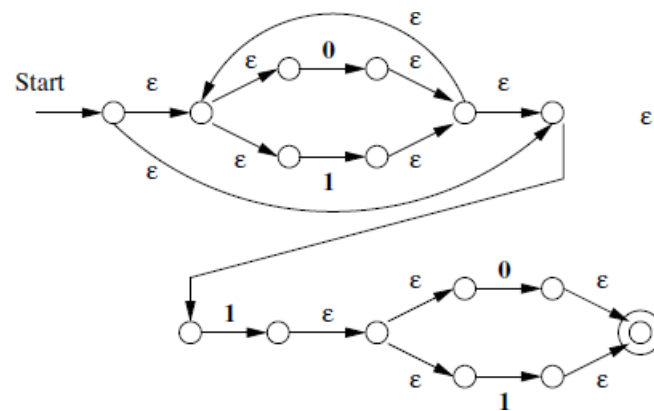
Step 1: Construct an automaton for $0+1$.



Step 2: Automaton for $(0+1)^*$



Step 3: Automata for $(0+1)^* 1(0+1)$



Applications of Regular Expressions

A regular expression is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. Two important classes of regular-expression-based applications are: **Lexical analyzers and text search.**

Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities.

The UNIX notations for RE involves short-hands that allow complex RE to be written briefly. Most of real applications deal with ASCII character set. The UNIX RE allows us to write character classes to represent large set of characters as briefly as possible.

The rules for character classes are:

- 1) The symbol . (dot) stands for “any character”.
 - 2) $[a_1 a_2 a_3 \dots a_k] \Rightarrow a_1 + a_2 + a_3 + \dots + a_k$
 - 3) $[A-Z] \Rightarrow$ all the characters from A to Z.
 $[0-9] \Rightarrow$ digits ranging from 0 to 9.
-

- There are special notations for several of the most common classes of characters. For instance:
 - a) `[:digit:]` is the set of ten digits, the same as `[0-9]`.³
 - b) `[:alpha:]` stands for any alphabetic character, as does `[A-Za-z]`.
 - c) `[:alnum:]` stands for the digits and letters (alphabetic and numeric characters), as does `[A-Za-z0-9]`.

Operators used in UNIX systems:

1. The operator `|` is used in place of `+` to denote union.
2. The operator `?` means “zero or one of.” Thus, `R?` in UNIX is the same as `ε + R` in this book’s regular-expression notation.
3. The operator `+` means “one or more of.” Thus, `R+` in UNIX is shorthand for `RR*` in our notation.
4. The operator `{n}` means “*n* copies of.” Thus, `R{5}` in UNIX is shorthand for `RRRRR`.

1) Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called a “lexical analyzer” or “LEX”. This component scans the source program and recognizes all tokens, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

Example: Lex command to recognize keyword **else**, **identifier** and **some relational operators** are:

```
else                                {return(ELSE);}  
  
|[A-Za-z][A-Za-z0-9]*             {code to enter the found identifier  
                                   in the symbol table;  
                                   return(ID);  
                                   }  
  
>=                                {return(GE);}  
  
=                                  {return(ASGN);}  
  
...
```

Figure 3.19: A sample of lex input

2) Finding Patterns in Text

By using regular expression notation, it becomes easy to describe the patterns at a high level, with little effort. Suppose that we want to scan a very large number of Web pages and detect addresses such as street address.

To begin, a street address will probably end in “Street” or its abbreviation “St”. However, some people live on “Avenues” or “Roads” and these might be abbreviated in the address as well. Thus, we might use as the ending for our regular expression something like:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

To include the house number as part of the address, most house numbers are a string of digits. However, some will have a letter following as in “123 Main St.”. The regular expression is:

```
'[0-9]+[A-Z]? [A-Z] [a-z]* ( [A-Z] [a-z] ) *  
(Street|St\. |Avenue|Ave\. |Road|Rd\. ) '
```

Proving Languages Not to Be Regular

Regular languages are the languages accepted by DFA's, by NFA's, by ϵ -NFA's and also defined by regular expressions.

Not every language is a regular language. Let us consider the language, $L = \{0^n 1^n \mid n \geq 1\}$. This language contains all strings that consist of one or more 0's followed by an equal number of 1's. We claim that L is not a regular language, as it is not possible to track how many 0's we read to match with 1's.

The Pumping Lemma for Regular Languages: (State and Prove Pumping Lemma for regular Languages-6M)

Theorem 4.1: (The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. For all $k \geq 0$, the string xy^kz is also in L .

That is, we can always find a nonempty string y not too far from the beginning of w that can be “pumped”; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L .

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A . Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1 a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \dots, n$ define state p_i to be $\delta(q_0, a_1 a_2 \cdots a_i)$, where δ is the transition function of A , and q_0 is the start state of A . That is, p_i is the state A is in after reading the first i symbols of w . Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n + 1$ different p_i 's for $i = 0, 1, \dots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j , with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \cdots a_i$.
2. $y = a_{i+1} a_{i+2} \cdots a_j$.
3. $z = a_{j+1} a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w . The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j .

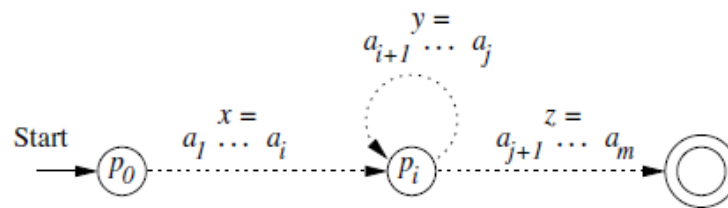


Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input $xy^k z$ for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x . Since p_i is also p_j , it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z . Thus, A accepts xz .

If $k > 0$, then A goes from q_0 to p_i on input x , circles from p_i to p_i k times on input y^k , and then goes to the accepting state on input z . Thus, for any $k \geq 0$, $xy^k z$ is also accepted by A ; that is, $xy^k z$ is in L . $\square$

Applications of the Pumping Lemma:

Basic application of Pumping lemma is to prove that language is not regular.

Steps to prove Language is not regular by contradiction method.

1. Assume L is regular and n be the number of states of FA.
2. Apply Pumping Lemma theorem for the given language.
3. Choose a string w , where $w \in L$ and $|w| \geq n$
4. Split w into xyz such that $|xy| \leq n$ and $y \neq \epsilon$
5. Choose a value for k such that xy^kz is not in L .
6. Since the result is contradictory to our assumption, the given language is not regular.

(Refer class work for problems and its solutions)

Closure Properties of Regular Languages:

In this section, we shall prove several theorems of the form “if certain languages are regular and a language L is formed from them by certain operations, e.g., L is the union of two regular languages, then L is also regular”. These theorems are often called **closure properties of the regular languages**.

The principal closure properties for regular languages are:

1. The union of two regular languages is regular.
 2. The closure (star) of a regular language is regular.
 3. The concatenation of regular languages is regular.
 4. The intersection of two regular languages is regular.
 5. The complement of a regular language is regular.
 6. The difference of two regular languages is regular.
 7. The reversal of a regular language is regular.
 8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
 9. The inverse homomorphism of a regular language is regular.
-

1. Regular languages are closed under Union, Concatenation and Star

Theorem: If L and M are regular, then LUM, L.M, and L* are also regular.

Proof: For proof we make use of RE. If L and M are regular languages then there exists regular expressions R1 & R2 such that:

$$L=L(R1)$$

$$M=L(R2)$$

By the definition of Res, we have

- R1+R2 is RE denoting the language LUM.
- R1.R2 is RE denoting the language L.M.
- R1* is RE denoting the language L*.

Closure Under Complementation

Theorem 4.5: If L is a regular language over alphabet Σ , then $\overline{L} = \Sigma^* - L$ is also a regular language.

PROOF: Let $L = L(A)$ for some DFA $A = (Q, \Sigma, \delta, q_0, F)$. Then $\overline{L} = L(B)$, where B is the DFA $(Q, \Sigma, \delta, q_0, Q - F)$. That is, B is exactly like A , but the accepting states of A have become nonaccepting states of B , and vice versa. Then w is in $L(B)$ if and only if $\hat{\delta}(q_0, w)$ is in $Q - F$, which occurs if and only if w is not in $L(A)$. $\square$

Closure Under Intersection

Theorem: If L and M are regular, then the regular language is closed under intersection.

Proof: Since we already proved the RLs are closed under union and Complementation, we can obtain the intersection of languages L and M by the identity (as per one of DeMorgan's law, we can obtain intersection of two sets in terms of complement of union of complement of languages):

$$L \cap M = \overline{\overline{L} \cup \overline{M}}$$

.Thus Regular languages are closed under intersection.

Closure Under Difference

In terms of languages, $L - M$, the difference of L and M , is the set of strings that are in language L but not in language M . The regular languages are also closed under this operation, and the proof follows easily from the theorems just proven.

Theorem 4.10: If L and M are regular languages, then so is $L - M$.

PROOF: Observe that $L - M = L \cap \overline{M}$. By Theorem 4.5, $\overline{M}$ is regular, and by Theorem 4.8 $L \cap \overline{M}$ is regular. Therefore $L - M$ is regular. $\square$

Reversal:

The *reversal* of a string $a_1 a_2 \cdots a_n$ is the string written backwards, that is, $a_n a_{n-1} \cdots a_1$. We use w^R for the reversal of string w . Thus, 0010^R is 0100 , and $\epsilon^R = \epsilon$.

The reversal of a language L , written L^R , is the language consisting of the reversals of all its strings. For instance, if $L = \{001, 10, 111\}$, then $L^R = \{100, 01, 111\}$.

Given a language L that is $L(A)$ for some finite automaton, perhaps with nondeterminism and ϵ -transitions, we may construct an automaton for L^R by:

1. Reverse all the arcs in the transition diagram for A .
2. Make the start state of A be the only accepting state for the new automaton.
3. Create a new start state p_0 with transitions on ϵ to all the accepting states of A .

The result is an automaton that simulates A “in reverse,” and therefore accepts a string w if and only if A accepts w^R . Now, we prove the reversal theorem formally.

Theorem 4.11: If L is a regular language, so is L^R .

PROOF: Assume L is defined by regular expression E . The proof is a structural induction on the size of E . We show that there is another regular expression E^R such that $L(E^R) = (L(E))^R$; that is, the language of E^R is the reversal of the language of E .

BASIS: If E is ϵ , $\emptyset$, or a , for some symbol a , then E^R is the same as E . That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, and $\{a\}^R = \{a\}$.

INDUCTION: There are three cases, depending on the form of E .

1. $E = E_1 + E_2$. Then $E^R = E_1^R + E_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversals of the two languages and taking the union of those languages.

2. $E = E_1 E_2$. Then $E^R = E_2^R E_1^R$. Note that we reverse the order of the two languages, as well as reversing the languages themselves. For instance, if $L(E_1) = \{01, 111\}$ and $L(E_2) = \{00, 10\}$, then $L(E_1 E_2) = \{0100, 0110, 11100, 11110\}$. The reversal of the latter language is

$$\{0010, 0110, 00111, 01111\}$$

If we concatenate the reversals of $L(E_2)$ and $L(E_1)$ in that order, we get

$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

which is the same language as $(L(E_1 E_2))^R$. In general, if a word w in $L(E)$ is the concatenation of w_1 from $L(E_1)$ and w_2 from $L(E_2)$, then $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Then $E^R = (E_1^R)^*$. The justification is that any string w in $L(E)$ can be written as $w_1 w_2 \cdots w_n$, where each w_i is in $L(E)$. But

$$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

Each w_i^R is in $L(E^R)$, so w^R is in $L((E_1^R)^*)$. Conversely, any string in $L((E_1^R)^*)$ is of the form $w_1 w_2 \cdots w_n$, where each w_i is the reversal of a string in $L(E_1)$. The reversal of this string, $w_n^R w_{n-1}^R \cdots w_1^R$, is therefore a string in $L(E_1^*)$, which is $L(E)$. We have thus shown that a string is in $L(E)$ if and only if its reversal is in $L((E_1^R)^*)$.

Homomorphisms

A string homomorphism is a function on strings that works by substituting a particular string for each symbol.

Formally, if h is a homomorphism on alphabet Σ , and $w = a_1a_2 \cdots a_n$ is a string of symbols in Σ , then $h(w) = h(a_1)h(a_2) \cdots h(a_n)$. That is, we apply h to each symbol of w and concatenate the results, in order. For instance, if h is the homomorphism in Example 4.13, and $w = 0011$, then

$$h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab, \text{ as we claimed in that example.}$$

Theorem 4.14: If L is a regular language over alphabet Σ , and h is a homomorphism on Σ , then $h(L)$ is also regular.

PROOF: Let $L = L(R)$ for some regular expression R . In general, if E is a regular expression with symbols in Σ , let $h(E)$ be the expression we obtain by replacing each symbol a of Σ in E by $h(a)$. We claim that $h(R)$ defines the language $h(L)$.

The proof is an easy structural induction that says whenever we take a subexpression E of R and apply h to it to get $h(E)$, the language of $h(E)$ is the same language we get if we apply h to the language $L(E)$. Formally, $L(h(E)) = h(L(E))$.

BASIS: If E is ϵ or $\emptyset$, then $h(E)$ is the same as E , since h does not affect the string ϵ or the language $\emptyset$. Thus, $L(h(E)) = L(E)$. However, if E is $\emptyset$ or ϵ , then $L(E)$ contains either no strings or a string with no symbols, respectively. Thus $h(L(E)) = L(E)$ in either case. We conclude $L(h(E)) = L(E) = h(L(E))$.

The only other basis case is if $E = \mathbf{a}$ for some symbol a in Σ . In this case, $L(E) = \{a\}$, so $h(L(E)) = \{h(a)\}$. Also, $h(E)$ is the regular expression that is the string of symbols $h(a)$. Thus, $L(h(E))$ is also $\{h(a)\}$, and we conclude $L(h(E)) = h(L(E))$.

INDUCTION: There are three cases, each of them simple. We shall prove only the union case, where $E = F + G$. The way we apply homomorphisms to regular expressions assures us that $h(E) = h(F + G) = h(F) + h(G)$. We also know that $L(E) = L(F) \cup L(G)$ and

$$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \quad (4.2)$$

by the definition of what “+” means in regular expressions. Finally,

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \quad (4.3)$$

because h is applied to a language by application to each of its strings individually. Now we may invoke the inductive hypothesis to assert that $L(h(F)) = h(L(F))$ and $L(h(G)) = h(L(G))$. Thus, the final expressions in (4.2) and

(4.3) are equivalent, and therefore so are their respective first terms; that is, $L(h(E)) = h(L(E))$.

Inverse Homomorphisms:

Homomorphisms may also be applied “backwards,” and in this mode they also preserve regular languages. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possibly the same) alphabet T .² Let L be a language over alphabet T . Then $h^{-1}(L)$, read “ h inverse of L ,” is the set of strings w in Σ^* such that $h(w)$ is in L . Figure 4.5 suggests the effect of a homomorphism on a language L in part (a), and the effect of an inverse homomorphism in part (b).

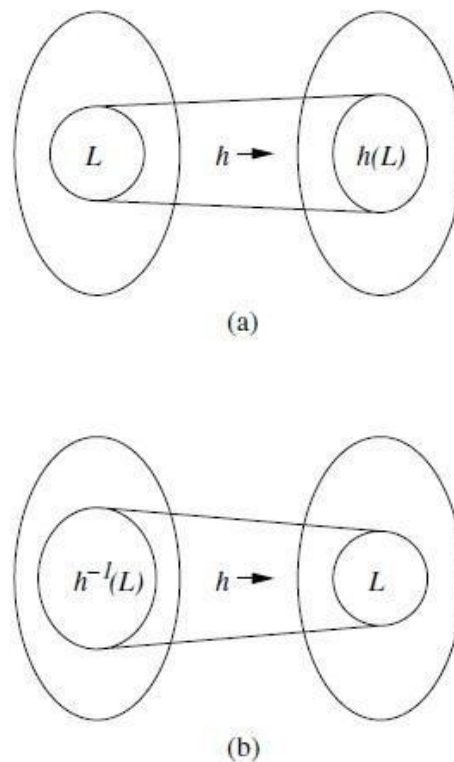


Figure 4.5: A homomorphism applied in the forward and inverse direction

Theorem 4.16: If h is a homomorphism from alphabet Σ to alphabet T , and L is a regular language over T , then $h^{-1}(L)$ is also a regular language.

PROOF: The proof starts with a DFA A for L . We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 4.6. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

Formally, let L be $L(A)$, where DFA $A = (Q, T, \delta, q_0, F)$. Define a DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

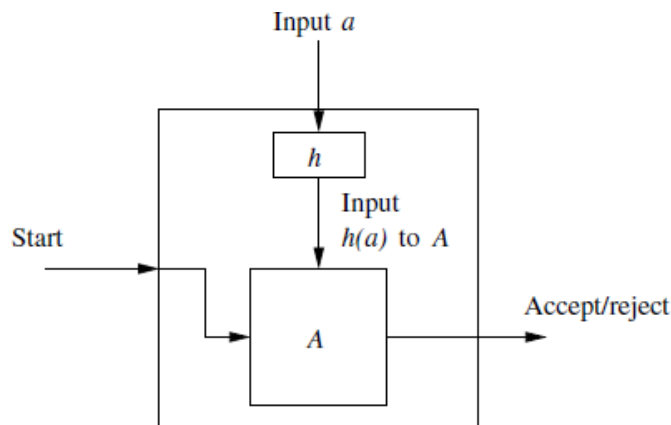


Figure 4.6: The DFA for $h^{-1}(L)$ applies h to its input, and then simulates the DFA for L

where transition function γ is constructed by the rule $\gamma(q, a) = \hat{\delta}(q, h(a))$. That is, the transition B makes on input a is the result of the sequence of transitions that A makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ , it could be one symbol, or it could be many symbols, but $\hat{\delta}$ is properly defined to take care of all these cases.

It is an easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings w that are in $h^{-1}(L)$.

Equivalence and Minimization of Automata:

Testing Equivalence of States:

We say that states p and q are **equivalent** if:

For all input strings w , $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

If two states are not equivalent, then we say they are distinguishable. That

is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. The algorithm, which we refer to as the **table-filling**

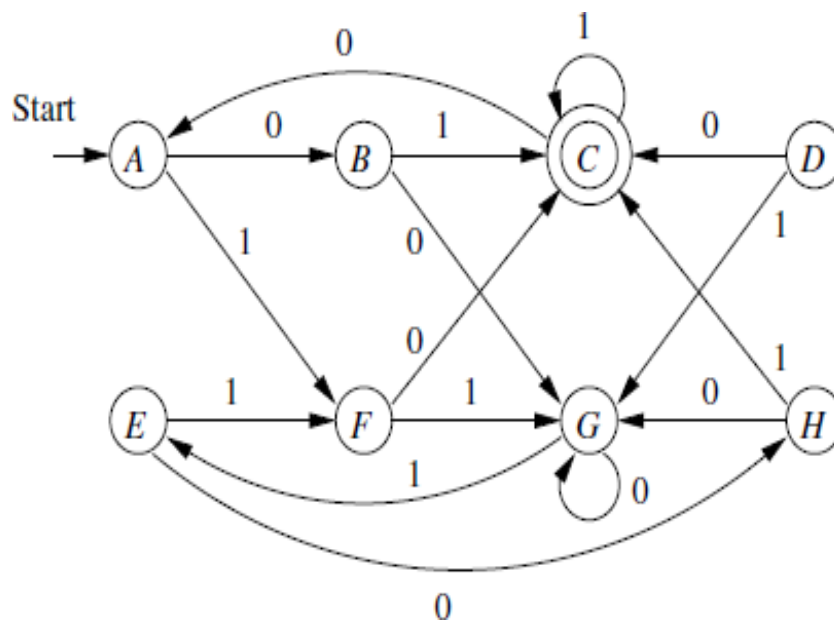
algorithm, is a recursive discovery of distinguishable pairs in a DFA $A=(Q, \Sigma, \delta, q_0, F)$.

Table-filling algorithm:

BASIS: If p is an accepting state and q is nonaccepting, then the pair $\{p, q\}$ is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a , $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then $\{p, q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s ; that is, exactly one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting. Then string aw must distinguish p from q , since $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is the same pair of states as $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$.

Example: Consider the following DFA,



Let us execute the table-filling algorithm on the DFA,

<i>B</i>	<i>x</i>						
<i>C</i>	<i>x</i>	<i>x</i>					
<i>D</i>	<i>x</i>	<i>x</i>	<i>x</i>				
<i>E</i>		<i>x</i>	<i>x</i>	<i>x</i>			
<i>F</i>	<i>x</i>	<i>x</i>	<i>x</i>		<i>x</i>		
<i>G</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	
<i>H</i>	<i>x</i>		<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>

Figure 4.9: Table of state inequivalences

Minimization of DFA's:

An important consequence of the test for equivalence of states is that we can “minimize” DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent.

Consider the table of Fig. 4.9, where we determined the state equivalences and distinguishabilities for the states of Fig. 4.8. The partition of the states into equivalent blocks is {A, E}, {B, H}, {C}, {D, F}, {G}.

Note: **The equivalence of states is transitive.** That is, if in some DFA $A=(Q, \Sigma, \delta, q_0, F)$ we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filling algorithm to find all the pairs of equivalent states.
2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B . Suppose S is a set of equivalent states of A , and a is an input symbol. Then there must exist one block T of states such that for all states q in S , $\delta(q, a)$ is a member of block T . For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by Theorem 4.24. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S . As a consequence, we can let $\gamma(S, a) = T$. In addition:
 - (a) The start state of B is the block containing the start state of A .
 - (b) The set of accepting states of B is the set of blocks containing accepting states of A . Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.

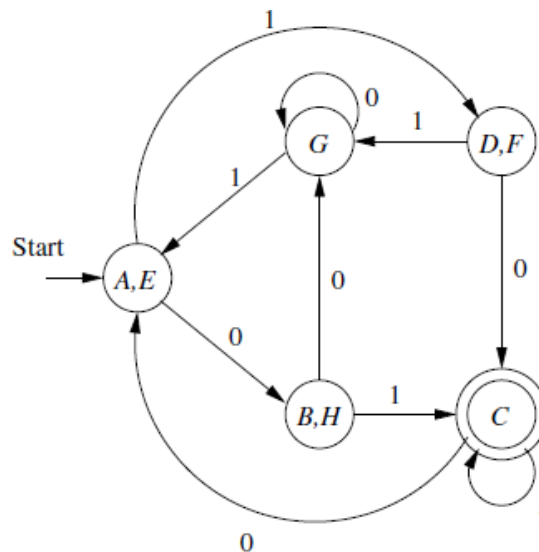


Figure 4.12: Minimum-state DFA equivalent to Fig. 4.8

Module-3

Context – Free Grammars

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG). A context-free grammar is a formal notation for expressing recursive definitions of languages.

Definition of Context-Free Grammars:

The context-free grammar is defined by quadruple,

$$G=(V, T, P, S)$$

where,

- V is a finite set of variables, also called sometimes non-terminals. Each variable represents a language i.e., a set of strings.
- T is a finite set of symbols that form the strings of the language being defined. We call this alphabet the terminals or terminal symbols.
- S is the start symbol, is one of the variables represents the language being defined.
- There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of:
 - a) A variable that is being defined by the production. This variable is often called the head of the production.
 - b) The production symbol $\rightarrow$
 - c) A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head.

Example: Consider the grammar for arithmetic expression as follows:

$$G= (V, T, S, P) \text{ with}$$

$V = \{E\}$ $T = \{+, *, \text{id}\}$ $P = \{ E \rightarrow E + E$ $E \rightarrow E - E$ $E \rightarrow E * E$ $E \rightarrow E / E$ $E \rightarrow \text{id} \quad \}$

and E is the start symbol.

Derivations using a Grammar:

“The process of obtaining string of terminals and/or non-terminals from the start symbol by applying some or all productions is called derivation”.

The process of deriving strings by applying productions from head to body requires the definition of a new relation symbol $\Rightarrow$. Suppose $G = (V, T, P, S)$ is a CFG. Let $\alpha A \beta$ be a string of terminals and variables, with A a variable. That is, α and β are strings in $(V \cup T)^*$, and A is in V . Let $A \rightarrow \gamma$ be a production of G . Then we say $\alpha A \beta \Rightarrow_G \alpha \gamma \beta$. If G is understood, we just say $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions.

We may extend the $\Rightarrow$ relationship to represent zero, one, or many derivation steps, much as the transition function δ of a finite automaton was extended to $\hat{\delta}$. For derivations, we use a $*$ to denote “zero or more steps,” as follows:

Let us derive the sentence **id+id*id** from the grammar for arithmetic expression:

 $E \Rightarrow E + E$ $\Rightarrow \text{id} + E$ $\Rightarrow \text{id} + E * E$ $\Rightarrow \text{id} + \text{id} * E$ $\Rightarrow \text{id} + \text{id} * \text{id}$

Leftmost and Rightmost Derivations

LMD:

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a **leftmost derivation**, and we indicate that a derivation is leftmost by using the relations $\Rightarrow_{lm}$ and $\xRightarrow{*}_{lm}$, for one or many steps, respectively.

RMD:

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation **rightmost** and use the symbols $\Rightarrow_{rm}$ and $\xRightarrow{*}_{rm}$ to indicate one or many rightmost derivations steps, respectively.

Ex: Consider the string $w = \mathbf{id+id*id}$. Let us derive w using LMD and RMD as follows:

LMD:

$$\begin{aligned} E &\Rightarrow_{lm} E+E \\ &\Rightarrow_{lm} id+E \\ &\Rightarrow_{lm} id+E*E \\ &\Rightarrow_{lm} id+id*E \\ &\Rightarrow_{lm} id+id*id \end{aligned}$$

RMD:

$$\begin{aligned} E &\Rightarrow_{rm} E+E \\ &\Rightarrow_{rm} E+E*E \\ &\Rightarrow_{rm} E+E*id \end{aligned}$$

$$\xRightarrow{rm} E+id*id$$

$$\xRightarrow{rm} id+id*id$$

The Language of a Grammar:

If $G = (V, T, P, S)$ is a CFG, the *language* of G , denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \xRightarrow{*}_G w\}$$

If a language L is the language of some context-free grammar, then L is said to be **context-free language or CFL**.

Sentential Forms:

Derivations from the start symbol produce strings that have a special role. We call these “sentential forms.” That is, if $G = (V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \xRightarrow{*}_{lm} \alpha$ is a *sentential form*. If $S \xRightarrow{*}_{lm} \alpha$, then α is a *left-sentential form*, and if $S \xRightarrow{*}_{rm} \alpha$, then α is a *right-sentential form*. Note that the language $L(G)$ is those sentential forms that are in T^* ; i.e., they consist solely of terminals.

Examples:

Design Context free grammar for the given language:

1. Construct the CFG for the language having any number of a's over the set

$$\Sigma = \{a\}.$$

Ans :

$$S \rightarrow aS \mid \epsilon$$

Therefore, $G = (V, T, S, P)$

where $V = \{S\}$, $T = \{a\}$, $P = \{S \rightarrow aS \mid \epsilon\}$, S is the start symbol.

2. Construct a CFG for the regular expression $(0+1)^*$

Ans: $S \rightarrow 0S \mid 1S \mid \epsilon$

Therefore, $G=(V,T,S,P)$

where $V=\{S\}$, $T=\{0,1\}$, $P=\{S \rightarrow 0S \mid 1S \mid \epsilon\}$, S is the start symbol.

3. Construct a CFG for a language $L = \{wcw^R \mid w \in (a, b)^*\}$. Also derive the string “abbcbbba”

Ans:

$S \rightarrow aSa \mid bSa \mid c$

Derivations:

$S \Rightarrow aSa$

$S \Rightarrow abSba \quad //S \rightarrow bSb \ S$

$\Rightarrow abbSbba \quad //S \rightarrow bSb \ S$

$\Rightarrow abbcbbba \quad //S \rightarrow c$

4. Construct a CFG for the language $L = a^n b^{2n}$ where $n \geq 1$.

Ans:

$S \rightarrow aSbb \mid ab$

5. $L = \{ a^n b^n \mid n \text{ is a positive integer} \}$

Ans:

$S \rightarrow aSb \mid ab$

therefore $G=(V, T, P, S)$ where,

$V = \{ S \}$, $T = \{ a, b \}$, $P = \{ S \rightarrow aSb, S \rightarrow ab \}$ and S is the start symbol.

6. Write CFG for balanced paranthesis.

$S \rightarrow (S) \mid [S] \mid \{S\} \mid SS \mid \epsilon$

7. $L = \{ww^R : w \in \{a,b\}^*\}$

Ans: $S \rightarrow aSa \mid bSb \mid \epsilon$

8. The CFG for Plaindrome of 0's and 1's is

Ans:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$$

9. $L = \{0^i 1^j \mid i \neq j, i \geq 0, j \geq 0\}$

Ans:

$$S \rightarrow 0S1 \mid A \mid B$$

$$A \rightarrow 0A \mid 0$$

$$B \rightarrow 1B \mid 1$$

10. $L = \{a^n b^m c^n \mid n \geq 0, m \geq 0\}$

Ans:

$$S \rightarrow aAc \mid \epsilon$$

$$A \rightarrow aAc \mid B$$

$$B \rightarrow bB \mid \epsilon$$

11. $L = \{w \in \{a,b\}^* : n_a(w) = n_b(w)\}$

Ans: $S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$

12. $L = \{w \in \{a,b\}^* : n_a(w) > n_b(w)\}$

Ans: $S \rightarrow aSb \mid bSa \mid SS \mid a \mid aS$

13. $L = \{w \in \{a,b\}^* : n_a(w) < n_b(w)\}$

Ans: $S \rightarrow aSb \mid bSa \mid SS \mid b \mid bS$

Note: Refer notes for More problems and its solutions.

Parse Trees

There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar.

Constructing Parse Trees

Let us consider a grammar $G=(V, T, P, S)$. The parse trees for G are trees with the following conditions:

1. Root of the tree is labelled by start symbol S of the grammar
2. Each interior node is labeled by a variable in V .
3. Each leaf is labeled by either a variable, a terminal, or ϵ . However, if the leaf is labelled ϵ , then it must be the only child of its parent.
4. If an interior node is labeled A , and its children are labeled

$$X_1, X_2, \dots, X_k$$

respectively, from the left, then $A \rightarrow X_1 X_2 \dots X_k$ is a production in P .

Note that the only time one of the X 's can be ϵ is if that is the label of the only child, and $A \rightarrow \epsilon$ is a production of G .

Example: Consider the following grammar,

$$S \rightarrow ASB \mid c$$

$$A \rightarrow \epsilon \mid aA$$

$$B \rightarrow \epsilon \mid bB$$

A derivation sequence of **acb**,

$$S \Rightarrow ASB$$

$$\Rightarrow aASB$$

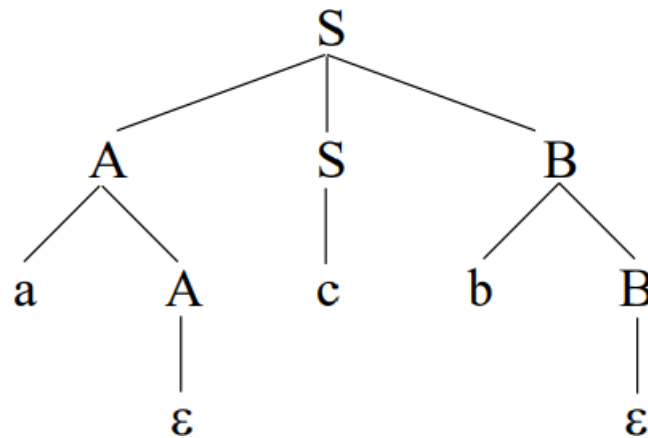
=>**aSB**

=>**acB**

=>**acbB**

=>**acb**

The Parse tree of **acb** is,



The Yield of a Parse Tree

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the **yield** of the tree, which is always a string that is derived from the root variable. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ϵ .
2. The root is labeled by the start symbol.

These are the parse trees whose yields are strings in the language of the underlying grammar.

Ex: in the above parse tree yield is, $a \in c b \in \Rightarrow acb$.

Exercise on parse tree:

1. Consider the grammar $S \rightarrow (L) \mid a$

$$L \rightarrow L, S \mid S$$

- i. What are the terminals, nonterminal and the start symbol?
- ii. Find the parse tree for the following sentence
 - a. (a,a)
 - b. $(a, (a, a))$
 - c. $(a, ((a,a),(a,a)))$
- iii. Construct LMD and RMD for each.

Ambiguity in Grammars and Languages:

Applications of CFG's often rely on the grammar to provide the structure of files. When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

Ambiguous Grammars:

Q. What do you mean by ambiguous grammar? Explain with example.

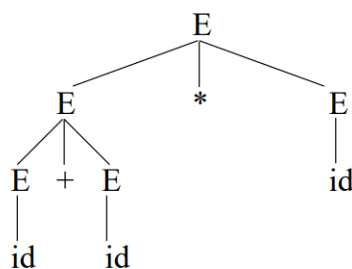
A CFG is ambiguous if it produces more than one parse tree for a string in the language. – i.e. there exists any string in the language that is the yield of two or more parse trees. In other way, if we are able to derive any string of a grammar either by applying LMD two or more time or applying RMD two or more time we call such grammar as Ambiguous grammar.

Example:

1. A good example for ambiguous grammar is the grammar for **arithmetic expressions**:

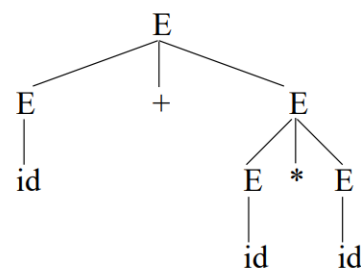
$$E \rightarrow E+E \mid E * E \mid E-E \mid E/E \mid id \mid (E)$$

For the expression $id+id*id$ we can generate 2 parse trees and 2 leftmost or 2 rightmost derivations as follows:



$$\begin{aligned} E &\Rightarrow_{lm} E * E \Rightarrow_{lm} E + E * E \Rightarrow_{lm} id + E * E \\ &\Rightarrow_{lm} id + id * E \Rightarrow_{lm} id + id * id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow_{rm} E * E \Rightarrow_{rm} E * id \Rightarrow_{rm} E + E * id \\ &\Rightarrow_{rm} E + id * id \Rightarrow_{rm} id + id * id \end{aligned}$$



$$\begin{aligned} E &\Rightarrow_{lm} E + E \Rightarrow_{lm} id + E \Rightarrow_{lm} id + E * E \\ &\Rightarrow_{lm} id + id * E \Rightarrow_{lm} id + id * id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow_{rm} E + E \Rightarrow_{rm} E + E * E \Rightarrow_{rm} E + E * id \\ &\Rightarrow_{rm} E + id * id \Rightarrow_{rm} id + id * id \end{aligned}$$

Since we got 2 different parse trees and LMD's we say that given grammar is ambiguous.

Removing Ambiguity From Grammars:

For the most parsers, the grammar must be unambiguous. Because,

- unambiguous grammar means unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An ambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

Example 1:

There are two causes of ambiguity in the grammar of arithmetic expression,

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id$$

1. The precedence of operators is not respected.
2. A sequence of identical operators can group either from the left or from the right.

Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity rules**. We can eliminate ambiguity of this grammar using the following precedence and associativity rules.

Precedence: from highest to lowest

$*, /$ (left to right)

$+, -$ (left to right)

Therefore the unambiguous grammar for arithmetic expression is,

$$E \rightarrow E + T \mid E - T$$
$$T \rightarrow T * F \mid T / F$$
$$F \rightarrow (E) \mid id$$

Exercise Questions: Prove that the following grammars are ambiguous.

a) $S \rightarrow AB \mid aaB$

$$A \rightarrow a \mid Aa$$
$$B \rightarrow b$$

b) $S \rightarrow ABA$

$$A \rightarrow aA \mid \epsilon$$
$$B \rightarrow bB \mid \epsilon$$

Inherent Ambiguity

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language.

Consider the Language,

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $a^+ b^+ c^+ d^+$ such that either:

1. There are as many a 's as b 's and as many c 's as d 's, or
2. There are as many a 's as d 's and as many b 's as c 's.

The grammar for this language is,

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

Since it is impossible to derived more than one parse tree for any strings of this grammar, it is an example for inherently ambiguous grammar.

PUSHDOWN AUTOMATA

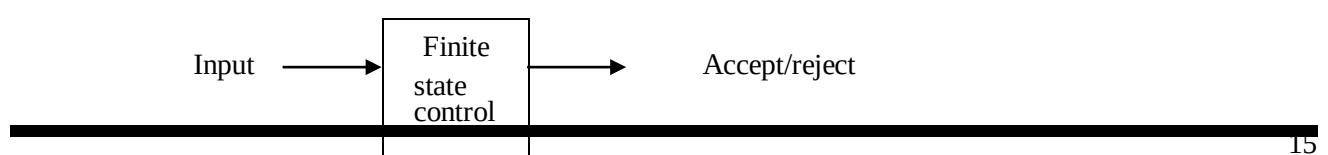
The context free languages have a type of automaton that defined them. This automaton, called a “pushdown automaton”, is an extension of the nondeterministic finite automaton with ϵ – transitions, which is one of the ways to define the regular languages.

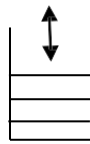
The pushdown automaton is essentially an ϵ – NFA with the addition of a stack. The stack canbe read, pushed and popped only at the top, just like the “stack” data structure.

We define two different versions of the pushdown automaton: one that accepts by entering an accepting state, like finite automata do and another version that accepts by emptying its stack, regardless of the state it is in. we show that these two variations accept exactly the context free languages i.e. grammars can be converted to equivalent pushdown automata and vice-versa.

Model of Pushdown Automata

- ❖ The pushdown automaton is in essence a nondeterministic finite automaton with ϵ – transitions permitted and one additional capability: a stack on which it can store a string of “stack symbols”.
- ❖ The presence of a stack means that unlike finite automaton, the pushdown automaton can remember an infinite amount of information.
- ❖ The model of PDA is as shown below:





Stack

Figure 6.1: A Pushdown automata with stack

- ❖ As shown in the Figure 6.1, PDA has finite state control that reads inputs, one symbol at a time. The pushdown automaton is allowed to observe the symbol at the top of the stack and to base its transition on its current state, input symbol, and the symbol at the top of stack.

FORMAL DEFINITION OF PUSHDOWN AUTOMATA

- ❖ Formal definition for *pushdown automata* (PDA) involves seven components. A PDA P is defined as follows:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

- ❖ Where,

- Q : a finite set of *states*, like the states of a finite automata
- Σ : a finite set of *input symbols*, also analogous to the corresponding component of a finite automaton.
- Γ : a *finite stack alphabet*. It is the set of symbols that we are allowed to push onto the stack.
- δ : the *transition function*. As for a finite automaton, δ governs the behavior of the automaton.

$$\delta: Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow Q \times \Gamma^*$$

Formally, δ takes as argument a triple $\delta(q, a, X)$, where:

1. q is the state in Q
2. a is either an input symbol in Σ or $a = \epsilon$, the empty string, which is assumed not to be an input symbol.
3. X is a stack symbol, which is a member of Γ .

The output of δ is a finite set of pairs (p, γ) , where p is the new state

and γ is the string of stack symbols that replaces X at the top of stack.

- q_0 : the *start state*. The PDA is in this state before making any transitions.
- Z_0 : the *start symbol*. Initially, the PDA's stack consists of one instance of this symbol, and nothing else.
- F : the set of *accepting states*, or *final states*.

A Graphical Notation for PDA's:

The **transition function** δ will explain the behavior of a PDA. It has following components:

- a) The nodes correspond to the states of the PDA.
- b) An arrow labeled *Start* indicates the start state, and doubly circled states are accepting, as for finite automata.
- c) The arcs correspond to transitions of the PDA in the following sense. An arc labeled $a, X/\alpha$ from state q to state p means that $\delta(q, a, X)$ contains the pair (p, α) , perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

Example: Let us design a PDA P to accept the language $L = \{ww^R : w \in \{0,1\}^*\}$

We shall use a stack symbol Z_0 to mark the bottom of the stack. We need to have this symbol present so that, after we pop w off the stack and realize that we have seen ww^R on the input, we still have something on the stack to permit us to make a transition to the accepting state, q_2 . Thus, our PDA for L_{ww^R} can be described as:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

where δ is defined by the following rules:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.
2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.
3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.
4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.
5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1 , then we have found an input of the form ww^R . We go to state q_2 and accept.

Transition diagram is,

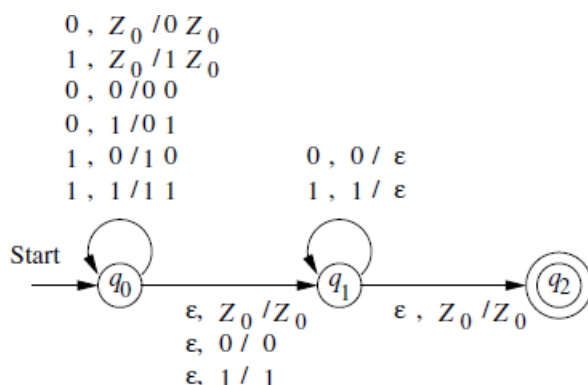


Figure 2: PDA for $L = \{ww^R : w \in \{0,1\}^*\}$

Instantaneous Descriptions of a PDA:

The PDA goes from configuration to configuration, in response to input symbols (or sometimes ϵ) but unlike the finite automaton, where the state is the only thing that we need to know about the automaton, the PDA's configuration involves both the state and the contents of the stack. We shall represent the configuration of a PDA by a triple (q, w, γ) , where:

1. **q** is the state,
2. **w** is the remaining input, and
3. **γ** is the stack contents

Such a triple is called an **instantaneous description**, or ID of the pushdown automaton.

Example:

Let us consider the action of the PDA of Example $L=\{ww^R\}$ on the input 1111. Since q_0 is the start state and Z_0 is the start symbol, the initial ID is $(q_0, 1111, Z_0)$. The IDs of the PDA are:

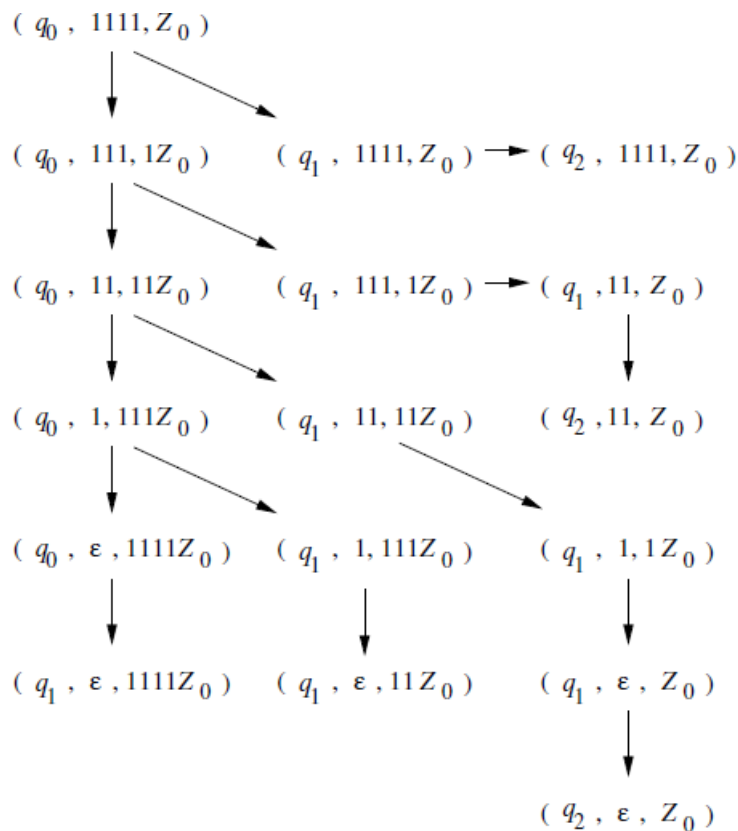


Figure 6.3: ID's of the PDA of Example 6.2 on input 1111

The Languages of a PDA:

- ❖ We have assumed that a PDA accepts its input by consuming it and entering an

accepting state. We call this approach “*acceptance by final state*”.

- ❖ There is a second approach to define the language of PDA that has important applications. We may also define for any PDA the language “*accepted by empty stack*”, i.e. the set of strings that cause the PDA to empty the stack, starting from the initial ID.
- ❖ These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack.

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α . That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

6.2.2 Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\}$$

for any state q . That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.²

EQUIVALENCE OF ACCEPTANCE BY FINAL STATE AND EMPTY STACK

From Empty Stack to Final State:

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0 , which must not be a symbol of Γ ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

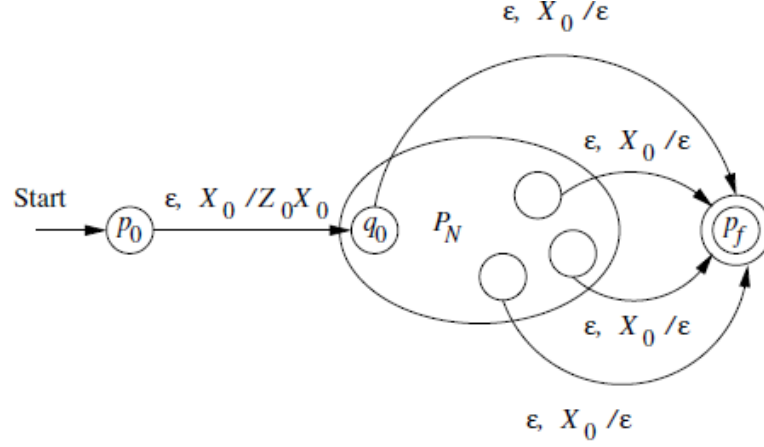


Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

We also need a new start state, p_0 , whose sole function is to push Z_0 , the start symbol of P_N , onto the top of the stack and enter state q_0 , the start state of P_N . Then, P_F simulates P_N , until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f , which is the accepting state of P_F ; this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N , pushing its start symbol Z_0 onto the stack.
2. For all states q in Q , inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ , $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.
3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q .

We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.

(If) We are given that $(q_0, w, Z_0) \vdash_{P_N}^* (q, \epsilon, \epsilon)$ for some state q . Theorem 6.5 lets us insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0 X_0) \vdash_{P_N}^* (q, \epsilon, X_0)$. Since by rule (2) above, P_F has all the moves of P_N , we may also conclude that $(q_0, w, Z_0 X_0) \vdash_{P_F}^* (q, \epsilon, X_0)$. If we put this sequence of moves together with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \vdash_{P_F} (q_0, w, Z_0 X_0) \vdash_{P_F}^* (q, \epsilon, X_0) \vdash_{P_F} (p_f, \epsilon, \epsilon) \quad (6.1)$$

Thus, P_F accepts w by final state.

From Final State to Empty Stack

Now, let us go in the opposite direction: take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack. The construction is simple and is suggested in Fig. 6.7. From each accepting state of P_F , add a transition on ϵ to a new state p . When in state p , P_N pops its stack and does not consume any input. Thus, whenever P_F enters an accepting state after consuming input w , P_N will empty its stack after consuming w .

To avoid simulating a situation where P_F accidentally empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack. The marker is P_N 's start symbol, and like the construction of Theorem 6.9, P_N must start in a new state p_0 , whose sole function is to push the start symbol of P_F on the stack and go to the start state of P_F . The construction is sketched in Fig. 6.7, and we give it formally in the next theorem.

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

PROOF: The construction is as suggested in Fig. 6.7. Let

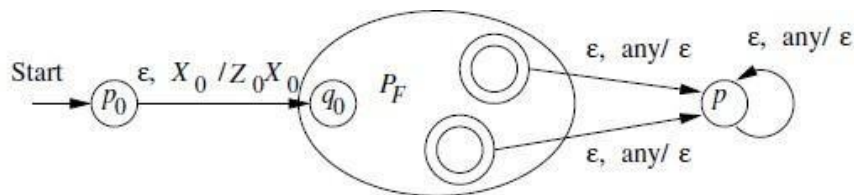


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

where δ_N is defined by:

1. $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$. We start by pushing the start symbol of P_F onto the stack and going to the start state of P_F .
2. For all states q in Q , input symbols a in Σ or $a = \epsilon$, and Y in Γ , $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F .
3. For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ) . By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.
4. For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Once in state p , which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.

Deterministic Pushdown Automata

Definition:

Intuitively, a PDA is deterministic if there is never a choice of move in any situation. These choices are of two kinds. If $\delta(q, a, X)$ contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move. However, even if $\delta(q, a, X)$ is always a singleton, we could still have a choice between using a real input symbol, or making a move on ϵ . Thus, we define a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be *deterministic* (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. $\delta(q, a, X)$ has at most one member for any q in Q , a in Σ or $a = \epsilon$, and X in Γ .
2. If $\delta(q, a, X)$ is nonempty, for some a in Σ , then $\delta(q, \epsilon, X)$ must be empty.

Example for Deterministic PDA is,

The PDA to accept $L = \{wCw^R : w \in \{0,1\}^*\}$

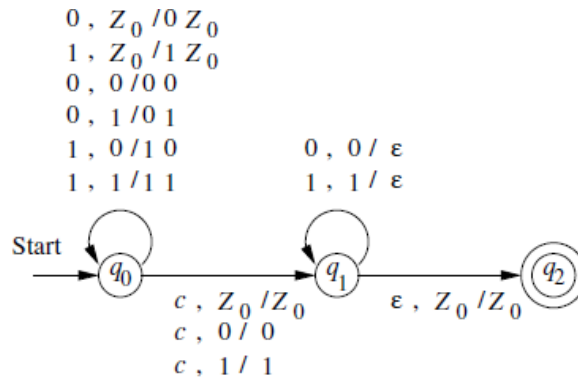
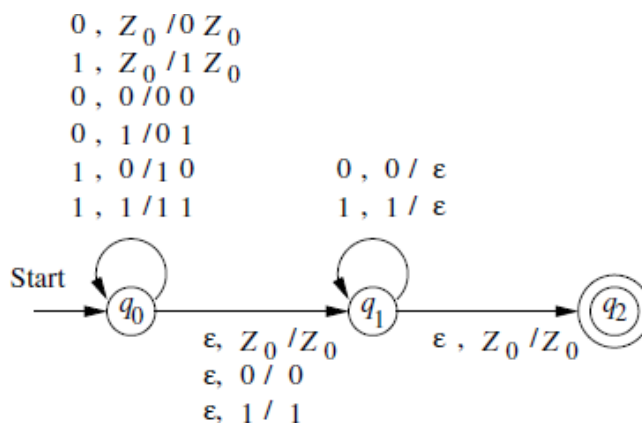


Figure 6.11: A deterministic PDA accepting L_{wcwr}

Example for Non-deterministic PDA is,

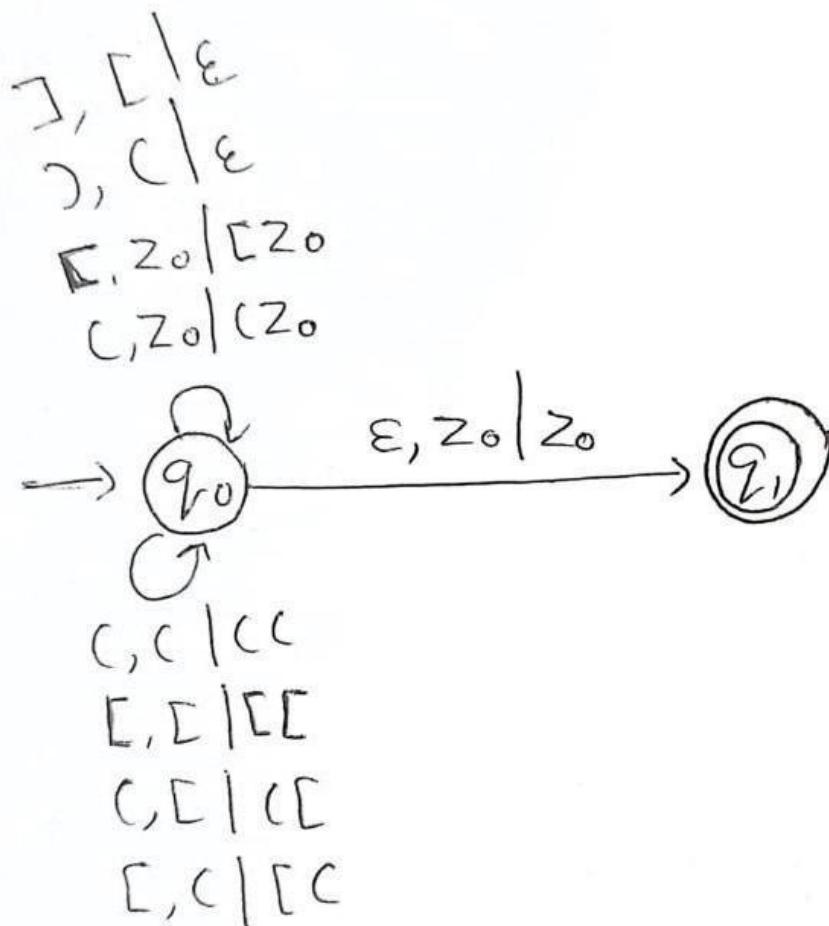
The PDA to accept $L = \{ww^R : w \in \{a,b\}^*\}$



Regular Languages and Deterministic PDA's:

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P .

PROOF: Essentially, a DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have



Therefore PDA $P=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Where,

$Q=\{q_0, q_1\}$

$\Sigma=\{a,b\}$

$\Gamma=\{[, Z_0\}$

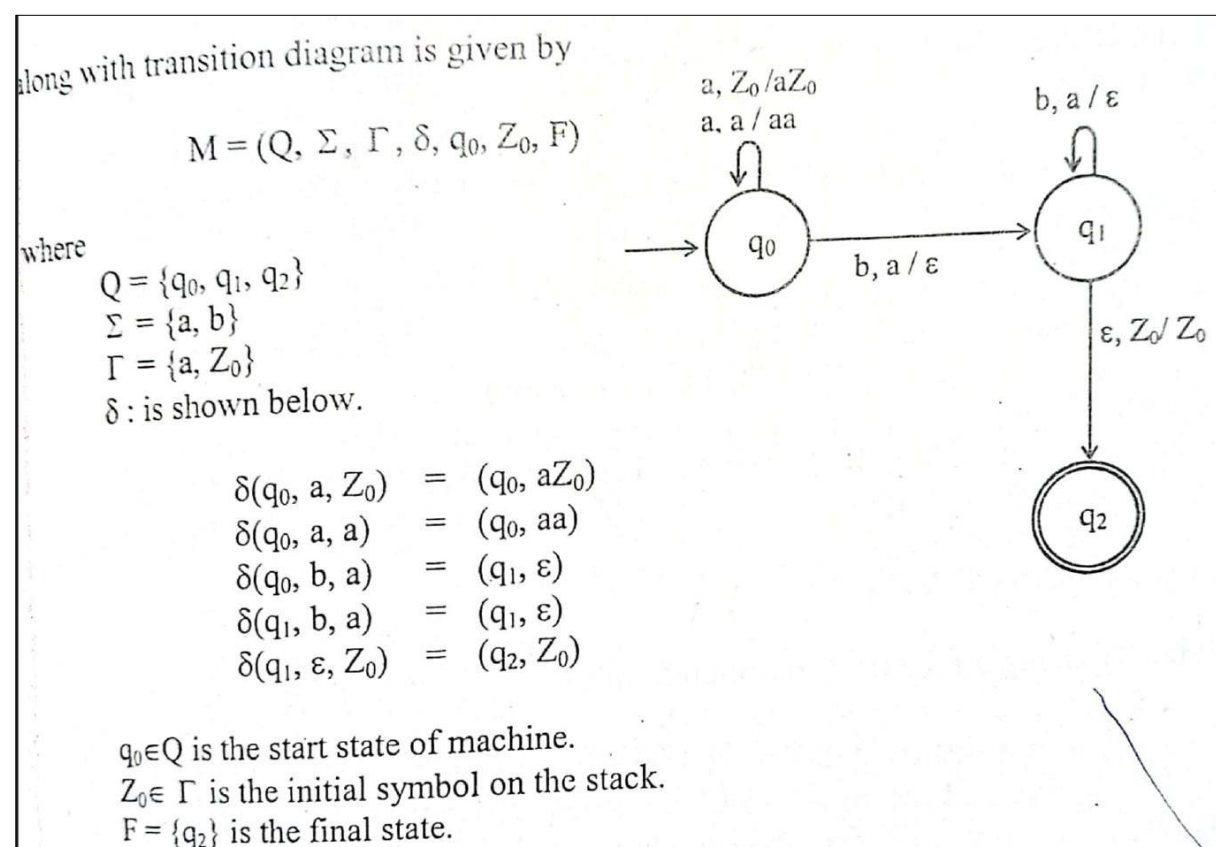
δ is as shown in the transition diagram.

q_0 is the initial state

$F=\{q_1\}$

3. Construct a Pushdown automata to accept the language $L=\{a^n b^n : n \geq 1\}$.

Also show the IDs for the string $w=aaabbb$.



Given $w=aaabbb$. The IDs are:

Initial ID	
$(q_0, aaabbb, Z_0)$	$\vdash (q_0, aabbbb, aZ_0)$
	$\vdash (q_0, abbbb, aaZ_0)$
	$\vdash (q_0, bbbb, aaaZ_0)$
	$\vdash (q_1, bb, aaZ_0)$
	$\vdash (q_1, b, aZ_0)$
	$\vdash (q_1, \epsilon, Z_0)$
	$\vdash (q_2, \epsilon, Z_0)$
	(Final Configuration)

Since q_2 is the final state and input string is ϵ in the final configuration, the string $aaabbb$

is accepted by the PDA.

4. Construct a PDA to accept the language $L = \{ w : w \in \{a,b\}^* \text{ and } n_a(w) = n_b(w) \}$.

$$L = \{ w \mid n_a(w) = n_b(w) \}$$

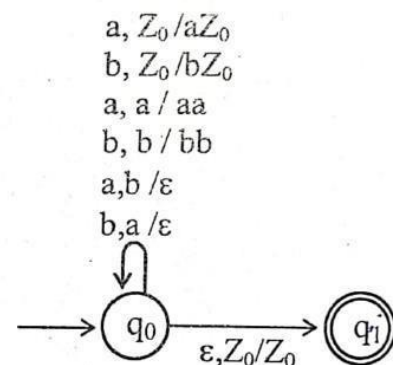
is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$$\begin{aligned} Q &= \{q_0, q_1\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{a, b, Z_0\} \\ \delta &: \text{is shown below.} \end{aligned}$$

$$\begin{aligned} \delta(q_0, a, Z_0) &= (q_0, aZ_0) \\ \delta(q_0, b, Z_0) &= (q_0, bZ_0) \\ \delta(q_0, a, a) &= (q_0, aa) \\ \delta(q_0, b, b) &= (q_0, bb) \\ \delta(q_0, a, b) &= (q_0, \epsilon) \\ \delta(q_0, b, a) &= (q_0, \epsilon) \\ \delta(q_0, \epsilon, Z_0) &= (q_1, Z_0) \end{aligned}$$



$q_0 \in Q$ is the start state of machine.

$Z_0 \in \Gamma$ is the initial symbol on the stack.

$F = \{q_1\}$ is the final state.

Let us trace the string $w=abbbaa$ as follows:

To accept the string: The sequence of moves made by the PDA for the string $abbbaa$ is shown below.

Initial ID	
$(q_0, abbbaa, Z_0)$	$\vdash (q_0, bbbaa, aZ_0)$
	$\vdash (q_0, bbbaa, Z_0)$
	$\vdash (q_0, baa, bZ_0)$
	$\vdash (q_0, aa, bbZ_0)$
	$\vdash (q_0, a, bZ_0)$
	$\vdash (q_0, \epsilon, Z_0)$
	$\vdash (q_1, \epsilon, Z_0)$
	(Final Configuration)

Since q_1 is the final state and input string is ϵ in the final configuration, the string

$abbbaa$

Note: For more problems refer class work

6.3 Equivalence of PDA's and CFG's

Now, we shall demonstrate that the languages defined by PDA's are exactly the context-free languages. The plan of attack is suggested by Fig. 6.8. The goal is to prove that the following three classes of languages:

1. The context-free languages, i.e., the languages defined by CFG's.
2. The languages that are accepted by final state by some PDA.
3. The languages that are accepted by empty stack by some PDA.

are all the same class. We have already shown that (2) and (3) are the same. It turns out to be easiest next to show that (1) and (3) are the same, thus implying the equivalence of all three.

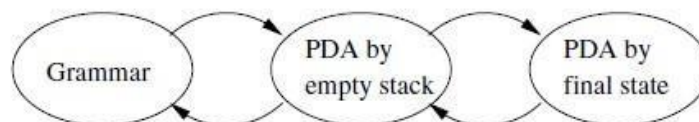


Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

a) From Grammars to Pushdown Automata- By empty stack method:**(Suitable for grammars not in GNF):**

Let $G = (V, T, P, S)$ be a CFG. Construct the PDA P that accepts $L(G)$ by empty stack as follows:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

where transition function δ is defined by:

1. For each variable A ,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is a production of } G\}$$

2. For each terminal a , $\delta(q, a, a) = \{(q, \epsilon)\}$.

Example 6.12: Let us convert the expression grammar of Fig. 5.2 to a PDA. Recall this grammar is:

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \rightarrow & I \mid E * E \mid E + E \mid (E) \end{array}$$

The set of input symbols for the PDA is $\{a, b, 0, 1, (,), +, *\}$. These eight symbols and the symbols I and E form the stack alphabet. The transition function for the PDA is:

- a) $\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$.
- b) $\delta(q, \epsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$.
- c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Note that (a) and (b) come from rule (1), while the eight transitions of (c) come from rule (2). Also, δ is empty except as defined by (a) through (c). $\square$

Method 2:**Converting CFG to PDA by final State (Suitable for CFG in GNF form)**

Algorithm:

Step1: Check whether grammar is in Greibach Normal Form(GNF)

[For GNF, all productions must be of the form, $A \rightarrow a\alpha$, where $a \in T$ and $\alpha \in V^*$]

Step2: Push Start symbol of the grammar to the stack and change state from q_0 to q_1 . The transition is:

$$\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$$

Step3: For each production of the form, $A \rightarrow a\alpha$, where $a \in T$ and $\alpha \in V^*$, add transitions,

$$\delta(q_1, a, A) = (q_1, \alpha)$$

Step4: Finally add a transition to accept the string by final state,

$$\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$$

Ex: Convert the following grammar to PDA by final state:

$$S \rightarrow aSSS \mid a$$

Ans:

Step1: The given grammar is in GNF.

Step2: $\delta(q_0, \epsilon, Z_0) = (q_1, SZ_0)$

Step3:

$S \rightarrow aSSS$	$\delta(q_1, a, S) = (q_1, SSS)$
$S \rightarrow a$	$\delta(q_0, a, S) = (q_1, \epsilon)$

Step4: $\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$

Therefore, $P = (\{q_0, q_1, q_f\}, \{a\}, \{S, Z_0\}, \delta, q_0, Z_0, \{q_f\})$

MODULE-4

NORMAL FORMS FOR CONTEXT-FREE GRAMMARS

The goal of this section is to show that every CFL, without ϵ is generated by a CFG in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A , B , and C are variables, and a is a terminal. This form is called Chomsky Normal Form. To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways.

1. We must eliminate useless symbols, those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.
2. We must eliminate ϵ -productions, those of the form $A \rightarrow \epsilon$ for some variable A .
3. We must eliminate unit productions, those of the form $A \rightarrow B$ for variables A and B .

1. Eliminating Useless Symbols:

We say a symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w$, where w is in T^* . Note that X may be in either V or T , and the sentential form $\alpha X \beta$ might be the first or last in the derivation. If X is not useful, we say it is *useless*. Evidently, omitting useless symbols from a grammar will not change the language generated, so we may as well detect and eliminate all useless symbols.

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \xRightarrow{*} w$ for some terminal string w . Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.
2. We say X is *reachable* if there is a derivation $S \xRightarrow{*} \alpha X \beta$ for some α and β .

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

All symbols but B are generating; a and b generate themselves; S generates a , and A generates b . If we eliminate B , we must eliminate the production $S \rightarrow AB$, leaving the grammar:

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow b \end{aligned}$$

Now, we find that only S and a are reachable from S . Eliminating A and b leaves only the production $S \rightarrow a$. That production by itself is a grammar whose language is $\{a\}$, just as is the language of the original grammar.

Note that if we start by checking for reachability first, we find that all symbols of the grammar

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

are reachable.

Theorem 7.2: Let $G = (V, T, P, S)$ be a CFG, and assume that $L(G) \neq \emptyset$; i.e., G generates at least one string. Let $G_1 = (V_1, T_1, P_1, S)$ be the grammar we obtain by the following steps:

1. First eliminate nongenerating symbols and all productions involving one or more of those symbols. Let $G_2 = (V_2, T_2, P_2, S)$ be this new grammar. Note that S must be generating, since we assume $L(G)$ has at least one string, so S has not been eliminated.
2. Second, eliminate all symbols that are not reachable in the grammar G_2 .

Then G_1 has no useless symbols, and $L(G_1) = L(G)$.

2. Eliminating ϵ -Productions:

Nullable Variable: A variable A is nullable if, $A \Rightarrow \epsilon$

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm. We shall then show that there are no nullable symbols except what the algorithm finds.

BASIS: If $A \rightarrow \epsilon$ is a production of G , then A is nullable.

INDUCTION: If there is a production $B \rightarrow C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Example 7.8: Consider the grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aAA \mid \epsilon \\ B &\rightarrow bBB \mid \epsilon \end{aligned}$$

First, let us find the nullable symbols. A and B are directly nullable because they have productions with ϵ as the body. Then, we find that S is nullable, because the production $S \rightarrow AB$ has a body consisting of nullable symbols only. Thus, all three variables are nullable.

Now, let us construct the productions of grammar G_1 . First consider $S \rightarrow AB$. All symbols of the body are nullable, so there are four ways we could choose present or absent for A and B , independently. However, we are not allowed to choose to make all symbols absent, so there are only three productions:

$$S \rightarrow AB \mid A \mid B$$

Next, consider production $A \rightarrow aAA$. The second and third positions hold nullable symbols, so again there are four choices of present/absent. In this case,

all four choices are allowable, since the nonnullable symbol a will be present in any case. Our four choices yield productions:

$$A \rightarrow aAA \mid aA \mid aA \mid a$$

Note that the two middle choices happen to yield the same production, since it doesn't matter which of the A 's we eliminate if we decide to eliminate one of them. Thus, the final grammar G_1 will only have three productions for A .

Similarly, the production B yields for G_1 :

$$B \rightarrow bBB \mid bB \mid b$$

The two ϵ -productions of G yield nothing for G_1 . Thus, the following productions:

$$\begin{aligned} S &\rightarrow AB \mid A \mid B \\ A &\rightarrow aAA \mid aA \mid a \\ B &\rightarrow bBB \mid bB \mid b \end{aligned}$$

constitute G_1 . $\square$

Eliminating Unit Productions:

“A unit production is a production of the form $A \rightarrow B$, where both A and B are variables.

However, unit productions can complicate certain proofs and they also introduce extra steps into derivations that technically need not be there.

Ex : $A \rightarrow B$

$B \rightarrow a \mid ab$

Can be rewritten as, $A \rightarrow a \mid ab$, by eliminating the unit production $A \rightarrow B$.

Consider the context free grammar given below and remove unit production for the same.

$S \rightarrow 0A \mid 1B \mid C$

$A \rightarrow 0S \mid 00$

$B \rightarrow 1 \mid A$

$C \rightarrow 01$

Step 1:

$S \rightarrow C$ is unit production but while removing $S \rightarrow C$ we have to consider what C gives so we can add a rule to S.

$S \rightarrow 0A \mid 1B \mid 01$

Step 2:

$B \rightarrow A$ is also unit production

$B \rightarrow 1 \mid 0S \mid 00$

Finally, we can write CFG without unit production as follows –

$S \rightarrow 0A \mid 1B \mid 01$

$A \rightarrow 0S \mid 00$

$B \rightarrow 1 \mid 0S \mid 00$

C->01

Therefore, the simplification involves following steps in the same order as mentioned below:

1. Eliminate ϵ -productions
2. Eliminate unit productions
3. Eliminate useless symbols

Chomsky Normal Form:

Every nonempty CFL without ϵ has a grammar G in which all productions are in one of the two simple forms, either:

1. $A \rightarrow BC$, where A , B , and C , are each variable, or
2. $A \rightarrow a$, where A is a variable and a is a terminal.

Also, G has no useless symbols. **Such a grammar is said to be in Chomsky Normal Form or CNF.**

Steps to convert given CFG to CFN:

1. Eliminate ϵ -productions
2. Eliminate unit productions
3. Eliminate useless symbols
4. Put the resulting grammar to CNF
 - Arrange all the bodies of length 2 or more consist only variables.
 - Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

Example :

Convert the following grammar to CNF.

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$

Ans:

Step 1: Eliminate ϵ -productions:

Given grammar doesn't have any ϵ -productions.

Step 2: Eliminate unit productions

Given grammar doesn't have any unit productions.

Step 3: Eliminate useless symbols

Given grammar doesn't have any useless symbols.

Step 4: Convert to CNF

$S \rightarrow ABT_a$	$S \rightarrow AD_1$ $D_1 \rightarrow BT_a$
$A \rightarrow T_a T_a T_b$	$A \rightarrow T_a D_2$ $D_2 \rightarrow T_a T_b$
$B \rightarrow AT_c$	$B \rightarrow AT_c$
$T_a \rightarrow a$ $T_b \rightarrow b$ $T_c \rightarrow c$	$T_a \rightarrow a$ $T_b \rightarrow b$ $T_c \rightarrow c$

Therefore, final grammar in CNF is $G' = (V', T', S, P')$

Where $V' = \{S, A, B\}$

$T' = \{a, b\}$

S is the start symbol

$$\begin{aligned} P' = \{ & S \rightarrow AD_1 \\ & D_1 \rightarrow BT_a \\ & A \rightarrow T_a D_2 \\ & D_2 \rightarrow T_a T_b \\ & B \rightarrow AT_c \\ & T_a \rightarrow a \\ & T_b \rightarrow b \\ & T_c \rightarrow c \} \end{aligned}$$

(Note: For More problems refer classwork)

The Pumping Lemma for Context-Free Languages:

The Size of Parse Trees:

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

BASIS: $n = 1$. Recall that the length of a path in a tree is the number of edges, i.e., one less than the number of nodes. Thus, a tree with a maximum path length of 1 consists of only a root and one leaf labeled by a terminal. String w is this terminal, so $|w| = 1$. Since $2^{n-1} = 2^0 = 1$ in this case, we have proved the basis.

INDUCTION: Suppose the longest path has length n , and $n > 1$. The root of the tree uses a production, which must be of the form $A \rightarrow BC$, since $n > 1$; i.e., we could not start the tree using a production with a terminal. No path in the subtrees rooted at B and C can have length greater than $n - 1$, since these paths exclude the edge from the root to its child labeled B or C . Thus, by the inductive hypothesis, these two subtrees each have yields of length at most 2^{n-2} . The yield of the entire tree is the concatenation of these two yields, and therefore has length at most $2^{n-2} + 2^{n-2} = 2^{n-1}$. Thus, the inductive step is proved. $\square$

Statement of the Pumping Lemma for CFL:

Theorem 7.18: (The pumping lemma for context-free languages) Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n , then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be “pumped,” this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, uv^iwx^iy is in L . That is, the two strings v and x may be “pumped” any number of times, including 0, and the resulting string will still be a member of L .

Proof:

Now, starting with a CNF grammar $G = (V, T, \check{P}, \check{S})$ such that $L(G) = L - \{\epsilon\}$, let G have m variables. Choose $n = 2^m$. Next, suppose that z in L is of length at least n . By Theorem 7.17, any parse tree whose longest path is of length m or less must have a yield of length $2^{m-1} = n/2$ or less. Such a parse tree cannot have yield z , because z is too long. Thus, any parse tree with yield z has a path of length at least $m + 1$.

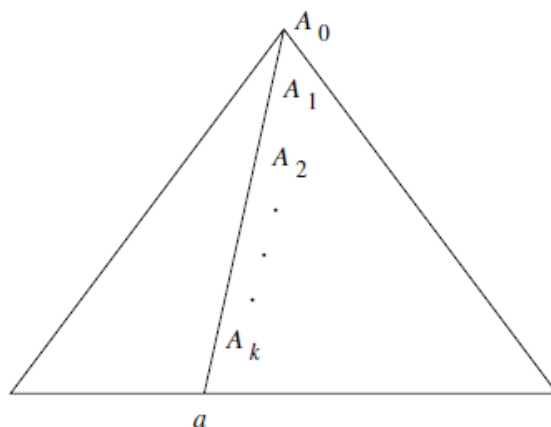


Figure 7.5: Every sufficiently long string in L must have a long path in its parse tree

Figure 7.5 suggests the longest path in the tree for z , where k is at least m and the path is of length $k+1$. Since $k \geq m$, there are at least $m+1$ occurrences of variables $A_0, A_1, \dots, A_k$ on the path. As there are only m different variables in V , at least two of the last $m+1$ variables on the path (that is, A_{k-m}

through A_k , inclusive) must be the same variable. Suppose $A_i = A_j$, where $k - m \leq i < j \leq k$.

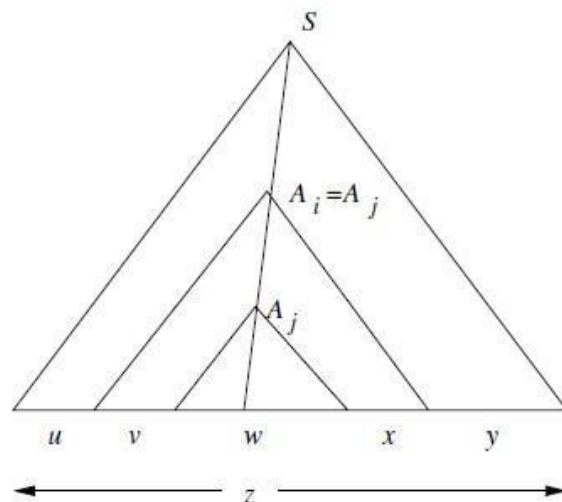


Figure 7.6: Dividing the string w so it can be pumped

Then it is possible to divide the tree as shown in Fig. 7.6. String w is the yield of the subtree rooted at A_j . Strings v and x are the strings to the left and right, respectively, of w in the yield of the larger subtree rooted at A_i . Note that, since there are no unit productions, v and x could not both be ϵ , although one could be. Finally, u and y are those portions of z that are to the left and right, respectively, of the subtree rooted at A_i .

If $A_i = A_j = A$, then we can construct new parse trees from the original tree, as suggested in Fig. 7.7(a). First, we may replace the subtree rooted at A_i , which has yield $vw x$, by the subtree rooted at A_j , which has yield w . The reason we can do so is that both of these trees have root labeled A . The resulting tree is suggested in Fig. 7.7(b); it has yield uwy and corresponds to the case $i = 0$ in the pattern of strings uv^iwx^iy .

Another option is suggested by Fig. 7.7(c). There, we have replaced the subtree rooted at A_j by the entire subtree rooted at A_i . Again, the justification is that we are substituting one tree with root labeled A for another tree with the same root label. The yield of this tree is uv^2wx^2y . Were we to then replace the subtree of Fig. 7.7(c) with yield w by the larger subtree with yield $vw x$, we would have a tree with yield uv^3wx^3y , and so on, for any exponent i . Thus, there are parse trees in G for all strings of the form uv^iwx^iy , and we have almost proved the pumping lemma.

The remaining detail is condition (1), which says that $|vw x| \leq n$. However, we picked A_i to be close to the bottom of the tree; that is, $k - i \leq m$. Thus, the longest path in the subtree rooted at A_i is no greater than $m + 1$. By Theorem 7.17, the subtree rooted at A_i has a yield whose length is no greater than $2^m = n$. $\square$

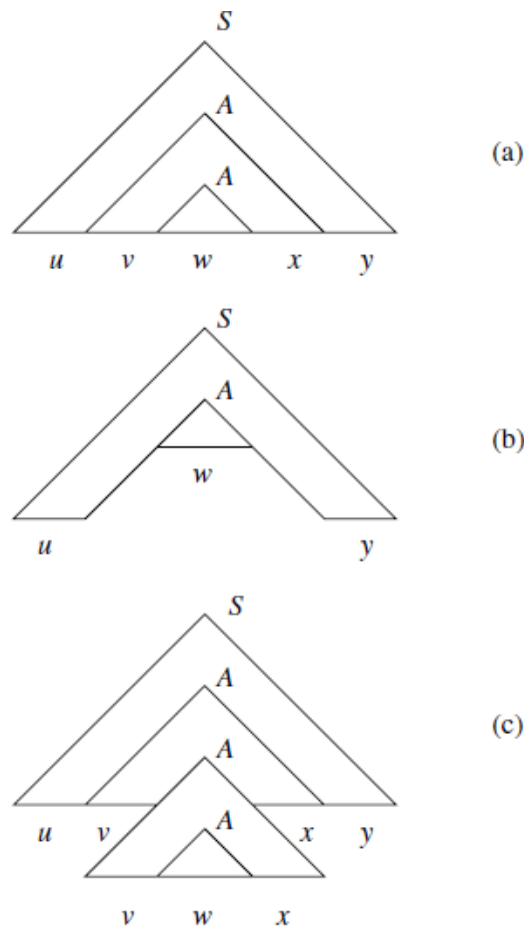


Figure 7.7: Pumping strings v and x zero times and pumping them twice

Application of Pumping Lemma

The pumping lemma for CFL's is used to prove that certain languages are not context free languages.

The general strategy used to prove that a given language is not context free is as follows:

1. Assume that the language L is infinite and is context free.
2. Select the string say z and break it into sub strings u, v, w, x and y such that $z = uvwxy$ where,
 $|vwx| \leq k$ & $vx \neq \epsilon$
3. Find any i such that $uv^iwx^iy \notin L$. According to pumping lemma,

$uv^i wx^i y \in L$. So the result is a contradiction to the assumption that the language is context-free. Thus, we can prove that the given language L is not context-free.

Example:

Show that $L = \{ a^n b^n c^n : n \geq 1 \}$ is not context free.

Ans:

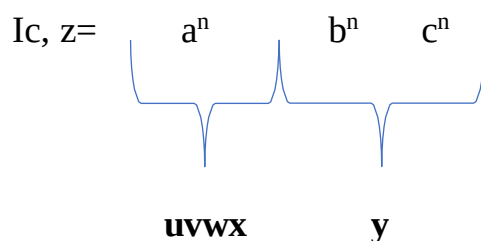
Let L is context free. Let $z = a^n b^n c^n \in L$

Since $|z| = 3n \geq n$, we can split z into $uvwxy$ such that,

$|vwx| \leq n$ & $vx \neq \epsilon$ such that $uv^i wx^i y \notin L$ for all $i = 0, 1, 2, 3, \dots$

Case 1: String vwx is within a^n

Let $v = a^j$, $x = a^k$ where $j+k \geq 1$ and $|vwx| \leq n$.



For $i=2$, $z = uv^2 wx^2 y \in L$

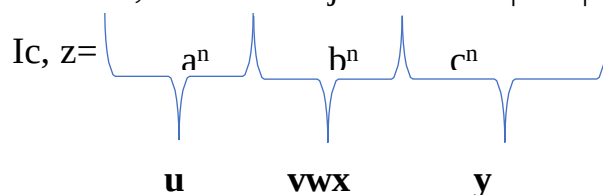
ie, $z = a^{n+j+k} b^n c^n$

but since $a^{n+j+k} b^n c^n \notin L$, our assumption is wrong. Thus, given language is not Context-free.

Case 2: String vwx is within b^n

Similarly we can prove with oth

Let $v = b^j$, $x = b^k$ where $j+k \geq 1$ and $|vwx| \leq n$.



For $i=2$, $z=uv^2wx^2y \in L$

$$\text{ic, } z = a^n b^{n+j+k} c^n$$

but since $a^n b^{n+j+k} c^n \notin L$, our assumption is wrong. Thus, given language is not Context-free.

(Note: More examples refer class work)

Closure Properties of CFLs:

CFLs are closed under:

1. Substitution
2. Union, concatenation and star closure
3. Reverse
4. Homomorphism and inverse homomorphism

CFLs are not closed under

1. Intersection
2. Complement

7.3.1 Substitutions

Let Σ be an alphabet, and suppose that for every symbol a in Σ , we choose a language L_a . These chosen languages can be over any alphabets, not necessarily Σ and not necessarily the same. This choice of languages defines a function s (a *substitution*) on Σ , and we shall refer to L_a as $s(a)$ for each symbol a .

If $w = a_1 a_2 \cdots a_n$ is a string in Σ^* , then $s(w)$ is the language of all strings $x_1 x_2 \cdots x_n$ such that string x_i is in the language $s(a_i)$, for $i = 1, 2, \dots, n$. Put another way, $s(w)$ is the concatenation of the languages $s(a_1)s(a_2) \cdots s(a_n)$. We can further extend the definition of s to apply to languages: $s(L)$ is the union of $s(w)$ for all strings w in L .

Example 7.22: Suppose $s(0) = \{a^n b^n \mid n \geq 1\}$ and $s(1) = \{aa, bb\}$. That is, s is a substitution on alphabet $\Sigma = \{0, 1\}$. Language $s(0)$ is the set of strings with one or more a 's followed by an equal number of b 's, while $s(1)$ is the finite language consisting of the two strings aa and bb .

Theorem 7.23: If L is a context-free language over alphabet Σ , and s is a substitution on Σ such that $s(a)$ is a CFL for each a in Σ , then $s(L)$ is a CFL.

PROOF: The essential idea is that we may take a CFG for L and replace each terminal a by the start symbol of a CFG for language $s(a)$. The result is a single CFG that generates $s(L)$. However, there are a few details that must be gotten right to make this idea work.

More formally, start with grammars for each of the relevant languages, say $G = (V, \Sigma, P, S)$ for L and $G_a = (V_a, T_a, P_a, S_a)$ for each a in Σ . Since we can choose any names we wish for variables, let us make sure that the sets of variables are disjoint; that is, there is no symbol A that is in two or more of V and any of the V_a 's. The purpose of this choice of names is to make sure that when we combine the productions of the various grammars into one set of productions, we cannot get accidental mixing of the productions from two grammars and thus have derivations that do not resemble the derivations in any of the given grammars.

We construct a new grammar $G' = (V', T', P', S)$ for $s(L)$, as follows:

- V' is the union of V and all the V_a 's for a in Σ .
- T' is the union of all the T_a 's for a in Σ .
- P' consists of:
 1. All productions in any P_a , for a in Σ .
 2. The productions of P , but with each terminal a in their bodies replaced by S_a everywhere a occurs.

CFLs are under Union, Concatenation and Star-closure:

Union

Let L_1 and L_2 be two context free languages. Then we can prove that $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n, n > 0 \}$. Corresponding grammar G_1 will have P: $S_1 \rightarrow aAb|ab$

Let $L_2 = \{ c^m d^m, m \geq 0 \}$. Corresponding grammar G_2 will have P: $S_2 \rightarrow cBb| \epsilon$

Union of L_1 and L_2 , $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 | S_2$.

Thus, CFLs are closed under Union.

Concatenation:

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2 , $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 S_2$

Thus, CFLs are closed under concatenation.

Kleene Star or star closure:

If L is a context free language, then L^* is also context free.

Example

Let $L = \{ a^n b^n, n \geq 0 \}$. Corresponding grammar G will have $P: S \rightarrow aAb \mid \epsilon$

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions $S_1 \rightarrow SS_1 \mid \epsilon$.

Thus, CFLs are closed under star closure.

CFLs are closed under Reversal:

Theorem 7.25: If L is a CFL, then so is L^R .

PROOF: Let $L = L(G)$ for some CFL $G = (V, T, P, S)$. Construct $G^R = (V, T, P^R, S)$, where P^R is the "reverse" of each production in P . That is, if $A \rightarrow \alpha$ is a production of G , then $A \rightarrow \alpha^R$ is a production of G^R . It is an easy induction on the lengths of derivations in G and G^R to show that $L(G^R) = L^R$. Essentially, all the sentential forms of G^R are reverses of sentential forms of G , and vice-versa. We leave the formal proof as an exercise. $\square$

CFLs are closed under Homomorphism:

Let L be a CFL with grammar G . Let h be a homomorphism on the terminal symbols of G .

We can construct a grammar for $h(L)$ by replacing each terminal symbol a by $h(a)$.

Example:

G has a production $S \rightarrow 0S1 \mid 01$

h is defined by $h(0)=ab$, $h(1)=\epsilon$

$h(L(G))$ has the grammar with productions $S \rightarrow abS \mid ab$.

Thus, CFLs are closed under homomorphism.

CFLs are closed under Inverse homomorphism:

Theorem 7.30: Let L be a CFL and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

PROOF: Suppose h applies to symbols of alphabet Σ and produces strings in T^* . We also assume that L is a language over alphabet T . As suggested above, we start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ that accepts L by final state. We construct a new PDA

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \quad (7.1)$$

where:

1. Q' is the set of pairs (q, x) such that:
 - (a) q is a state in Q , and
 - (b) x is a suffix (not necessarily proper) of some string $h(a)$ for some input symbol a in Σ .

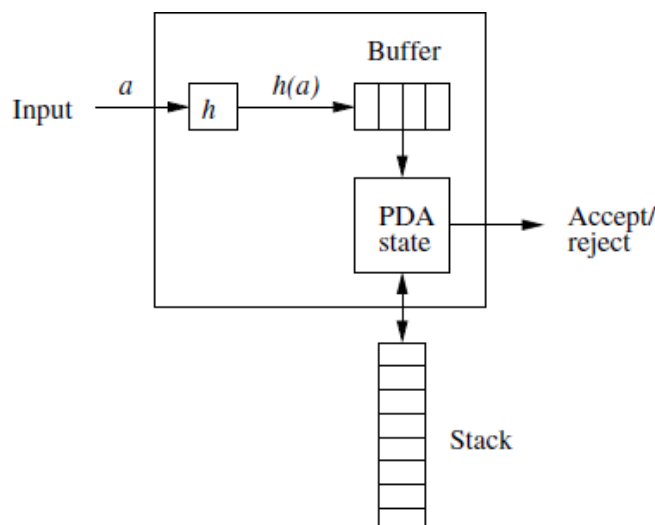


Figure 7.10: Constructing a PDA to accept the inverse homomorphism of what a given PDA accepts

That is, the first component of the state of P' is the state of P , and the second component is the buffer. We assume that the buffer will periodically be loaded with a string $h(a)$, and then allowed to shrink from the front, as we use its symbols to feed the simulated PDA P . Note that since Σ is finite, and $h(a)$ is finite for all a , there are only a finite number of states for P' .

2. δ' is defined by the following rules:

- (a) $\delta'((q, \epsilon), a, X) = \{(q, h(a)), X\}$ for all symbols a in Σ , all states q in Q , and stack symbols X in Γ . Note that a cannot be ϵ here. When the buffer is empty, P' can consume its next input symbol a and place $h(a)$ in the buffer.
- (b) If $\delta(q, b, X)$ contains (p, γ) , where b is in T or $b = \epsilon$, then

$$\delta'((q, bx), \epsilon, X)$$

contains $((p, x), \gamma)$. That is, P' always has the option of simulating a move of P , using the front of its buffer. If b is a symbol in T , then the buffer must not be empty, but if $b = \epsilon$, then the buffer can be empty.

- 3. Note that, as defined in (7.1), the start state of P' is (q_0, ϵ) ; i.e., P' starts in the start state of P with an empty buffer.
- 4. Likewise, the accepting states of P' , as per (7.1), are those states (q, ϵ) such that q is an accepting state of P .

The following statement characterizes the relationship between P' and P :

- $(q_0, h(w), Z_0) \vdash_P^* (p, \epsilon, \gamma)$ if and only if $((q_0, \epsilon), w, Z_0) \vdash_{P'}^* ((p, \epsilon), \epsilon, \gamma)$.

The proofs in both directions are inductions on the number of moves made by the two automata. In the “if” portion, one needs to observe that once the buffer of P' is nonempty, it cannot read another input symbol and must simulate P , until the buffer has become empty (although when the buffer is empty, it may still simulate P). We leave further details as an exercise.

Once we accept this relationship between P' and P , we note that P accepts $h(w)$ if and only if P' accepts w , because of the way the accepting states of P' are defined. Thus, $L(P') = h^{-1}(L(P))$. $\square$

CFLs are not closed under intersection:

Let us prove this with example grammars. Already we know that,

$L = \{a^n b^n c^n : n \geq 1\}$ is not context free.

However, the following two languages are context-free.

$$\begin{aligned} L_1 &= \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\} \\ L_2 &= \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\} \end{aligned}$$

A grammar for L_1 is:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A1 \mid 01 \\ B &\rightarrow 2B \mid 2 \end{aligned}$$

In this grammar, A generates all strings of the form $0^n 1^n$, and B generates all strings of 2's. A grammar for L_2 is:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A \mid 0 \\ B &\rightarrow 1B2 \mid 12 \end{aligned}$$

It works similarly, but with A generating any string of 0's, and B generating matching strings of 1's and 2's.

However, $L = L_1 \cap L_2$. To see why, observe that L_1 requires that there be the same number of 0's and 1's, while L_2 requires the numbers of 1's and 2's to be equal. A string in both languages must have equal numbers of all three symbols and thus be in L .

If the CFL's were closed under intersection, then we could prove the false statement that L is context-free. We conclude by contradiction that the CFL's are not closed under intersection. $\square$

CFLs are not closed under complement:**Theorem: If L is context free, it is not for complement.**

Proof. [Proof 1] Suppose CFLs were closed under complementation. Then for any two CFLs L_1, L_2 , we have

- $\overline{L_1}$ and $\overline{L_2}$ are CFL. Then, since CFLs closed under union, $\overline{L_1} \cup \overline{L_2}$ is CFL. Then, again by hypothesis, $\overline{\overline{L_1} \cup \overline{L_2}}$ is CFL.
- i.e., $L_1 \cap L_2$ is a CFL

i.e., CFLs are closed under intersection. Contradiction!

Thus, CFLs are not closed under complement.

Theorem: If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

PROOF: This proof requires the pushdown-automaton representation of CFL's, as well as the finite-automaton representation of regular languages, and generalizes the proof of Theorem 4.8, where we ran two finite automata "in parallel" to get the intersection of their languages. Here, we run a finite automaton "in parallel" with a PDA, and the result is another PDA, as suggested in Fig. 7.9.

Formally, let

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

be a PDA that accepts L by final state, and let

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

be a DFA for R . Construct PDA

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

where $\delta((q, p), a, X)$ is defined to be the set of all pairs $((r, s), \gamma)$ such that:

1. $s = \delta_A(p, a)$, and
2. Pair (r, γ) is in $\delta_P(q, a, X)$.

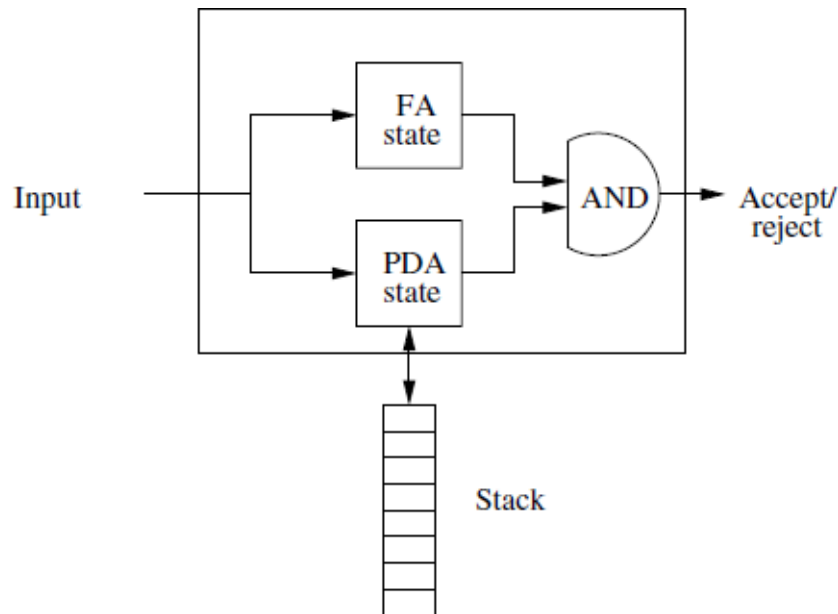


Figure 7.9: A PDA and a FA can run in parallel to create a new PDA

That is, for each move of PDA P , we can make the same move in PDA P' , and in addition, we carry along the state of the DFA A in a second component of the state of P' . Note that a may be a symbol of Σ , or $a = \epsilon$. In the former case, $\hat{\delta}(p, a) = \delta_A(p, a)$, while if $a = \epsilon$, then $\hat{\delta}(p, a) = p$; i.e., A does not change state while P makes moves on ϵ input.

It is an easy induction on the numbers of moves made by the PDA's that $(q_P, w, Z_0) \xrightarrow{*}_P (q, \epsilon, \gamma)$ if and only if $((q_P, q_A), w, Z_0) \xrightarrow{*}_{P'} ((q, p), \epsilon, \gamma)$, where

$p = \hat{\delta}(q_A, w)$. We leave these inductions as exercises. Since (q, p) is an accepting state of P' if and only if q is an accepting state of P , and p is an accepting state of A , we conclude that P' accepts w if and only if both P and A do; i.e., w is in $L \cap R$. $\square$

Module-5

TURING MACHINE

Problems that Computer Cannot Solve

Consider a simple C program with a simple print statement:

```
main()
{
    printf("hello, world\n");
}
```

It is easy to discover that this program prints **hello, world** and terminates.

However, we may have another program that might print **hello, world** as shown below. It takes an input n , and looks for positive integer solutions to the equation $x^n + y^n = z^n$. If it finds one, it prints **hello, world**. If it never finds integers x , y , and z to satisfy the equation, then it continues searching forever and never prints **hello, world**.

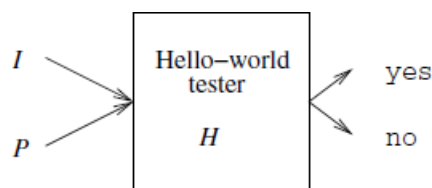
```
int exp(int i, n)
/* computes i to the power n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total = 12`, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program *does* print `hello, world`.

However, for any integer $n > 2$, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$, and thus will fail to print `hello, world`. Interestingly, until a few years ago, it was not known whether or not this program would print `hello, world` for some large integer n . The claim that it would not, i.e., that there are no integer solutions to the equation $x^n + y^n = z^n$ if $n > 2$, was made by Fermat 300 years ago, but no proof was found until quite recently. This statement is often referred to as “Fermat’s last theorem.”

- **Hypothetical tester:** If a problem has an algorithm like H that always tells correctly whether an instance of the problem has answer “yes” or “no” then the problem is said to be “decidable”. Otherwise, the problem is “undecidable”.



A hypothetical program H that is a hello-world detector

The **hypothetical tester** is a conceptual tool introduced in the study of undecidable problems to illustrate why certain problems cannot be solved by any algorithm. In this context, the "hypothetical tester" is designed to determine whether a given program, when executed with specific input, will output a particular result—in this case, whether it prints "**hello, world**" as its first output.

This program, denoted as H , takes two inputs:

1. A program P (the code to be tested).
2. An input I (the input to the program P).

The goal of H is to decide if the program P with input I will produce the output "**hello, world**" as the first set of characters it prints.

Expected Behavior of H :

- If P with input I prints "hello, world" as its first output, H should return "**yes**".
- Otherwise, it should return "**no**".

The challenge arises because:

- A program P may take an **infinite amount of time** to decide its output. For instance:
- If P is searching for a mathematical solution (like Fermat's Last Theorem) before it prints anything, H cannot predict whether it will eventually find a solution and print "hello, world" or loop indefinitely.
- Even for programs that terminate, analyzing the behavior of every possible program P with every possible input I is **non-trivial and undecidable**.

The hypothetical tester is a classic example used to demonstrate:

1. **Limits of Computation:** Some problems, even simple-sounding ones, are fundamentally unsolvable by any algorithm.
2. **Undecidability:** The inability to solve a problem algorithmically is a critical concept in theoretical computer science and is at the core of problems like the Halting Problem.

The **Halting Problem** is a fundamental concept in computer science and computational theory. It addresses whether it's possible to design an algorithm that can determine, for any given program and input, whether the program will eventually stop running (halt) or continue running forever.

Reducing one problem to another: A problem that cannot be solved by computer is called undecidable. A reduction technique is used to prove the **undecidability of** a halting problem. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in the figure given below. The technique is called the reduction of P1 to P2.

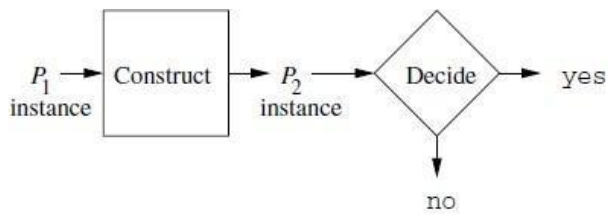


Figure 8.7: If we could solve problem P_2 , then we could use its solution to solve problem P_1

Using this technique, a problem P_1 is reducible to problem P_2 if a solution to the problem P_2 can be used to solve the problem P_1 . Thus,

- if P_1 is reducible to P_2 and P_2 is decidable, then P_1 is decidable
- If P_1 is reducible to P_2 and P_2 is undecidable, then P_1 is undecidable.

What is Turing Machine?

In 1936, **Alan M Turing** proposed the Turing machine as a model of any possible computation. This model is computer like, rather than program like, even though true electronic, or even electromechanical computers were several years in the future.

A Turing machine is a computational model, like **Finite Automata (FA)**, **Pushdown automata (PDA)**, which works on unrestricted grammar. The Turing machine is the most powerful computation model when compared with FA and PDA. Instead of using stack as in PDA, the Turing Machine (TM) uses the tape to store the symbols.

Church-Turing thesis

Any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine. Now it is universally accepted by computer scientists that TM is a Mathematical model of an algorithm. TM has an algorithm and an algorithm has a TM. **If there is an algorithm problem is decidable, TM solves that problem. This is known as Church-Turing thesis.**

Notation of Turing Machine:

(Q. Explain the Turing Machine model working principle. (4M))

The Turing machine consists of a finite control, which can be in any of a finite set of states. There is a tape divided into squares or cells, each cell can hold any one of a finite number of symbols.

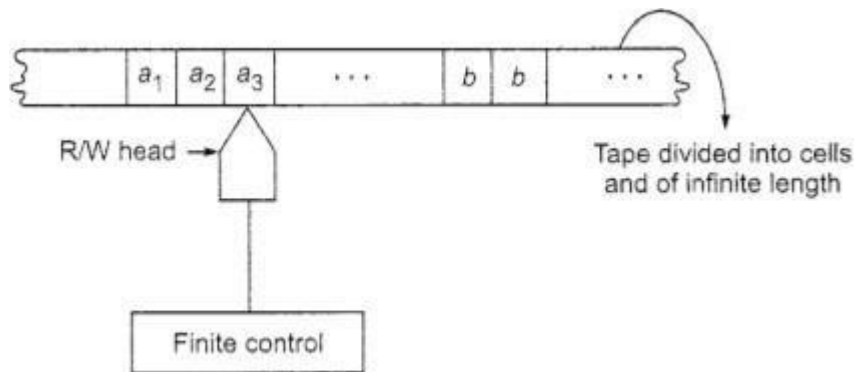


Fig. 9.1 Turing machine model.

- Initially, the input, which is a finite length string of symbols chosen from the input alphabet, is placed on the tape. All other tape cells, extending infinitely to the left and right, initially hold a special symbol called the blank. The blank is a tape symbol, but not an input symbol, and there may be other tape symbols besides the input symbols and the blank, as well.
- There is a tape head that is always positioned at one of the tape cells. The Turing machine is said to be scanning that cell. Initially, the tape head is at the leftmost cell that holds the input.
- A move of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will,
 - a) Change state: The next state optionally may be the same as the current state.
 - b) Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.
 - c) Move the tape head left or right. In our formalism we require a move, and do not allow the head to remain stationary. This

restriction does not constrain what a Turing machine can compute, since any sequence of moves with a stationary head could be condensed, along with the next tape head move, into a single state change, a new tape symbol, and a move left or right.

Definition of Turing Machine:**(Q. Define Turing machine-2M)**

A Turing machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where,

- Q : is the finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,
- $B \in \Gamma$ is the blank.
- Σ is a nonempty set of input symbols and is a subset of Γ and $B \notin \Sigma$.
- δ is the transition function.

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

The arguments of $\delta(q, X)$ are a state q and a tape symbol X .

The value of $\delta(q, X)$ is a triple (p, Y, D) where:

- p is the next state, in Q .
 - Y is the symbol in Γ , written in the cell being scanned, replacing whatever symbol was there.
 - D is the direction, either L or R , standing for “left” or “right” specifying the direction in which the R/W head moves.
- $q_0 \in Q$ is the initial state, and
 - $F \subseteq Q$ is the set of final or accepting states.

Instantaneous Description (IDs) of Turing Machine:

In Turing Machine, the instantaneous description is defined on the whole string and the current state of the machine.

Thus, we shall use the string $X_1X_2,\dots,X_{i-1} \mathbf{q}X_iX_{i+1},\dots,X_n$ to

represent an ID in which,

1. q is the state of the Turing machine.
2. The tape head is scanning the i th symbol from the left.
3. $X_1X_2, \dots, X_n$ is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2, \dots, X_n$ will be blank.

- We use $| \rightarrow_M$ to designate a move of a Turing machine M from one ID to another.

- If $\delta(q, X_i) = (p, Y, L)$, then:

$$X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n \rightarrow_M$$

$$X_1X_2 \cdots X_{i-2}pX_{i-1}YX_{i+1} \cdots X_n$$

- If $\delta(q, X_i) = (p, Y, R)$, then:

$$X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n \rightarrow_M$$

$$X_1X_2 \cdots X_{i-1}YpX_{i+1} \cdots X_n$$

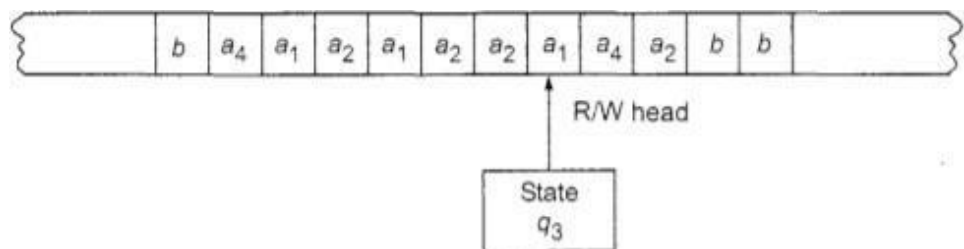


Fig. 9.2 A snapshot of Turing machine.

Thus, the ID is as given in Fig. 9.3.

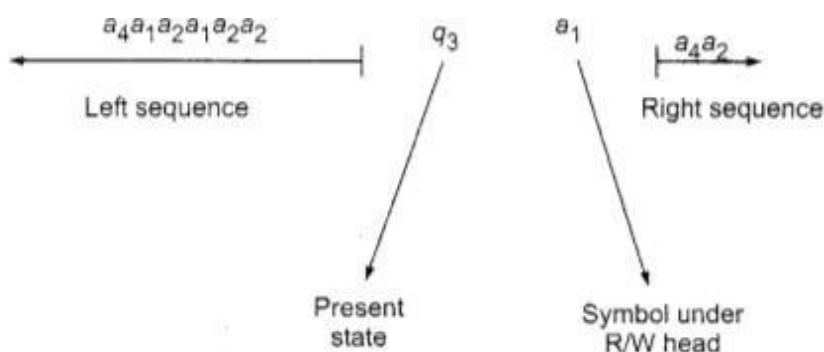


Fig. 9.3 Representation of ID.

Designing Turing Machine:

1. Let us design a Turing machine and see how it behaves on a typical input. The TM we construct will accept the language $L=\{0^n 1^n:n\geq 1\}$

Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's has been matched.

The formal specification of the TM M is,

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

where δ is given by the table in Fig. 8.9.

State	Symbol				
	0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Figure 8.9: A Turing machine to accept $\{0^n 1^n \mid n \geq 1\}$

This table is called as transition table.

Transition Diagrams for Turing Machines:

A transition diagram for TM consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/Y, D where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever $\delta(q,X)=(p, Y, D)$, we find the label X/Y, D on the arc from q to p.

Let $L = \{0^n 1^n : n \geq 1\}$, which is a CFL. We will design a Turing machine M accepting L . Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F \subseteq Q),$$

where $Q = \{q_0, q_1, \dots, q_4\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, X, Y, B\}$, $F = \{q_4\}$, and δ is shown in the following figure.

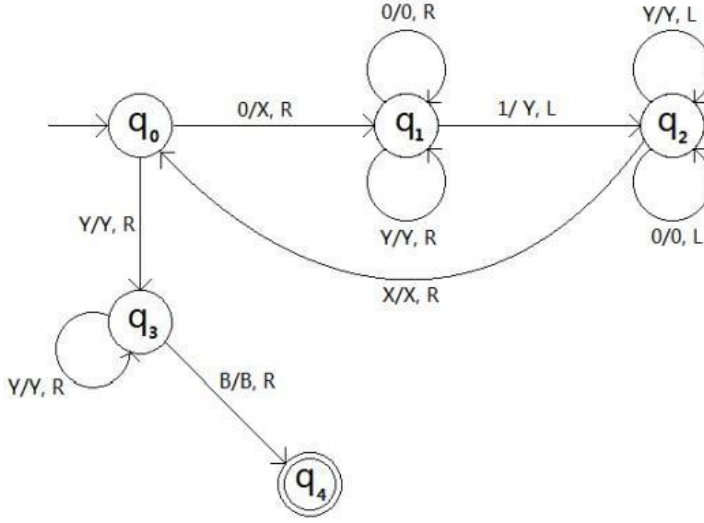


Figure 3: Turing machine accepts $\{0^n 1^n : n \geq 1\}$

Let show the IDs for the input strings 0011 and 0010.

Here is an example of an accepting computation by M . Its input is 0011. Initially, M is in state q_0 , scanning the first 0, i.e., M 's initial ID is $q_0 0011$. The entire sequence of moves of M is:

$$\begin{aligned} q_0 0011 &\vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0Y1 \vdash q_2 X 0Y1 \vdash \\ &X q_0 0Y1 \vdash X X q_1 Y1 \vdash X X Y q_1 1 \vdash X X q_2 Y Y \vdash X q_2 X Y Y \vdash \\ &X X q_0 Y Y \vdash X X Y q_3 Y \vdash X X Y Y q_3 B \vdash X X Y Y B q_4 B \end{aligned}$$

For another example, consider what M does on the input 0010, which is not in the language accepted.

$$\begin{aligned} q_0 0010 &\vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0Y0 \vdash q_2 X 0Y0 \vdash \\ &X q_0 0Y0 \vdash X X q_1 Y0 \vdash X X Y q_1 0 \vdash X X Y 0 q_1 B \end{aligned}$$

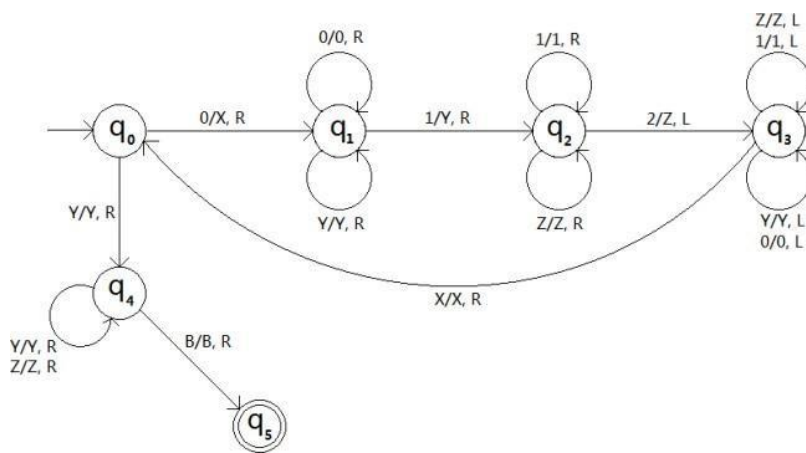
The behavior of M on 0010 resembles the behavior on 0011, until in ID $X X Y q_1 0$ M scans the final 0 for the first time. M must move right, staying in state q_1 , which takes it to the ID $X X Y 0 q_1 B$. However, in state q_1 M has no move on tape symbol B ; thus M dies and does not accept its input. $\square$

Example 2:

Construct a T.M. to accept the language $L = \{0^n 1^n 2^n : n \geq 1\}$.

Ans:

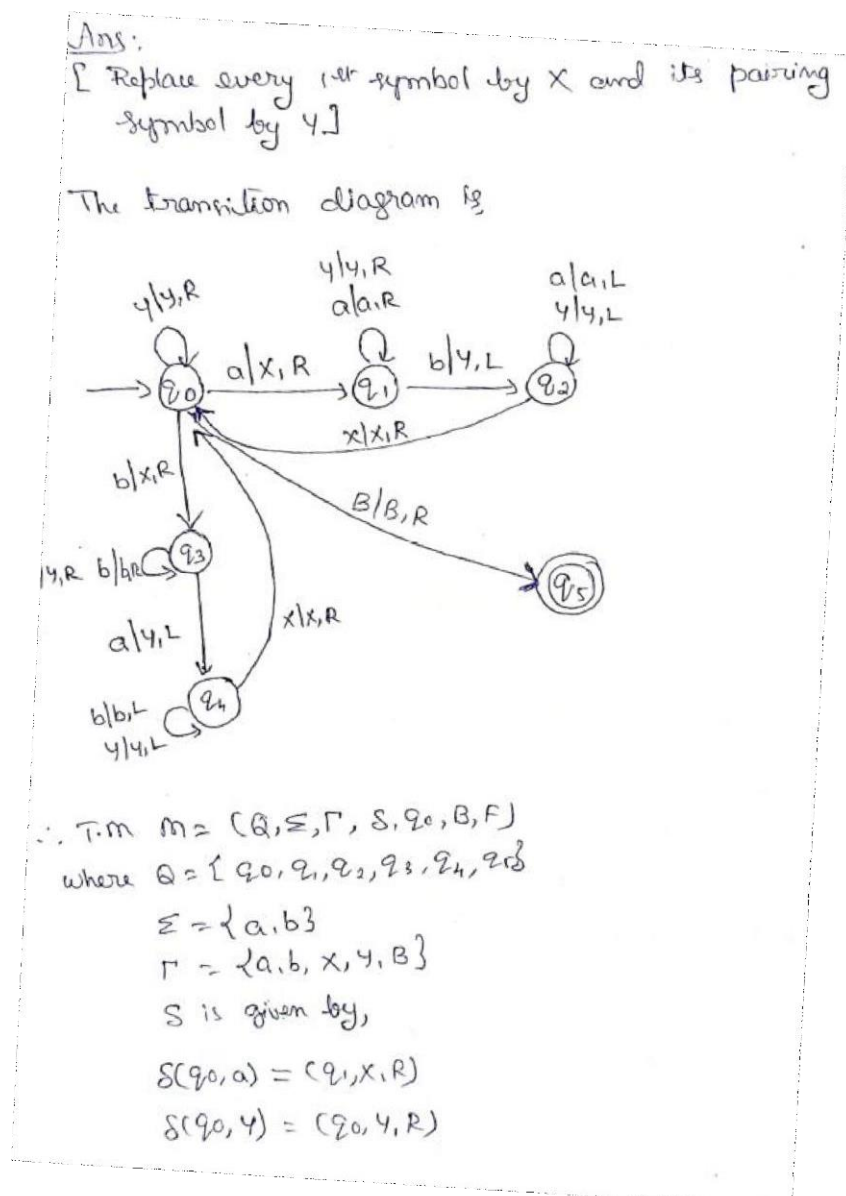
The Turing machine works similar with the above example: it marks the first 0 as X, the first 1 as Y, the first 2 as Z; then it goes back to the second 0, and marks it as X, the second 2 as Y, the second 2 as Z, and so on.

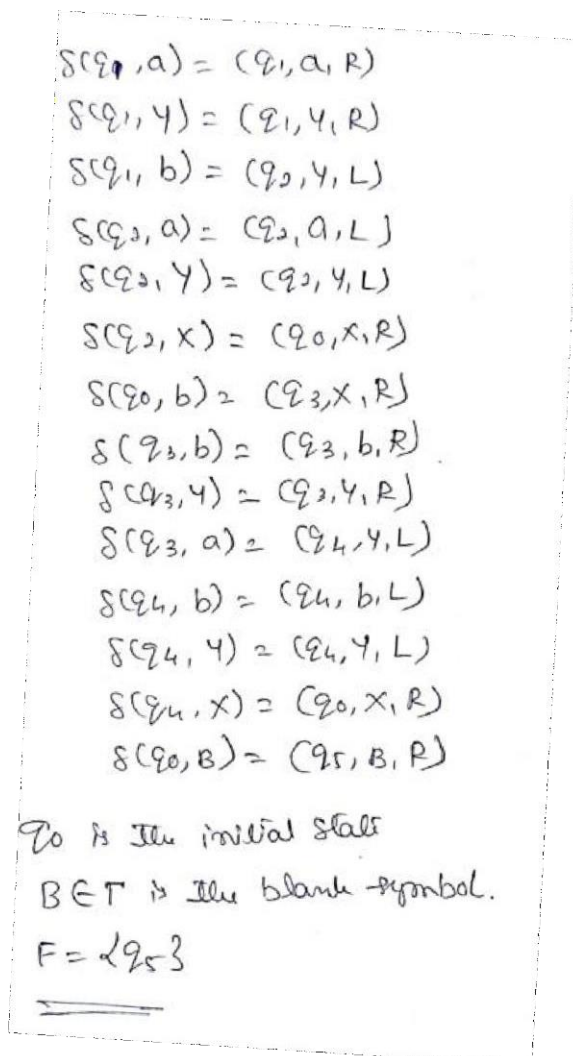
**Example 3:**

Obtain a TM to accept a string of a's and b's such that $n_a(w) = n_b(w)$.

Ans:

The general procedure: When TM is in initial state q_0 , first symbol of the input string can be either a or b. Replace the first symbol a (or b) by X and search for next symbol b (or a) and replace it by symbol Y. Repeat this until blank symbol B is scanned.

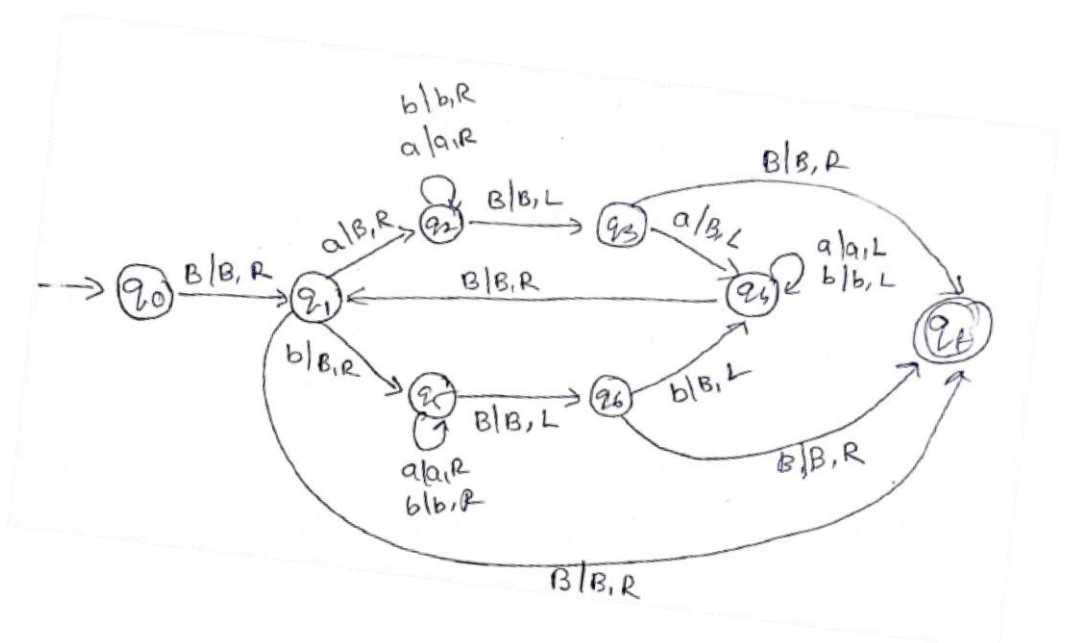




Example 4: Obtain a TM to accept a palindrome consisting of a's and b's of any length.

Ans: General idea is to place the given string in between blank symbols. For string to be palindrome, its first and last, second and last but one and so on should match. If we found match, replaced matched symbols by blank symbol B.

The transition diagram is,



therefore, $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where,

$Q=\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_f\}$

$\Sigma=\{a, b\}$

$\Gamma=\{a, b, B\}$

δ is shown in the transition diagram

q_0 is the initial state

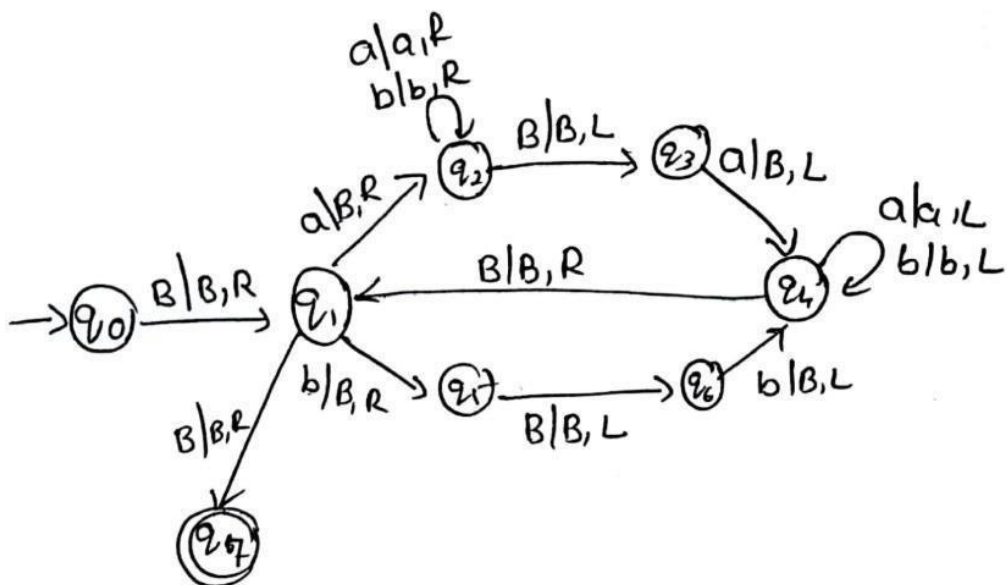
B is the blank symbol

$F=\{q_f\}$

Example 5: Obtain a TM to accept the language $L=\{ww^R: w \in (a+b)^*\}$

Ans: ww^R is also a palindrome problem only, but length of the palindrome must be even. Hence TM is same as previous but no transition required from q_3 and q_6 towards the final state.

Therefore, the transition diagram is,



Example 6: Let x and y are two positive integers. Obtain a turing machine to perform $x+y$. (Addition operation on TM)

Ans: Let us represent the numbers on the tape using unary representation 1'. Let 0 be the separator.

Ex: $x=5$ can be represented as 11111 and $y=3$ can be represented as 111. Then $5+3$ can be represented on the tape together with 0 as a separator,

1 1 1 1 1 0 1 1 1

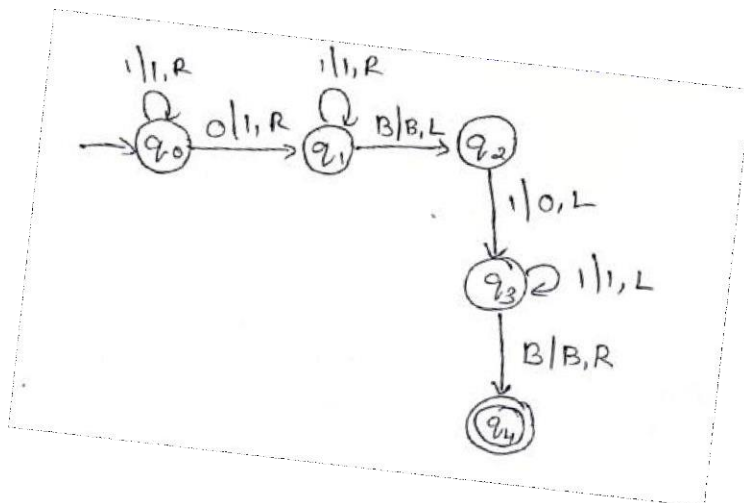
Then output should be of the form:

1 1 1 1 1 1 1 1 0

ic, read all 1's until we encounter the 0. Then change 0 to 1 and keep

moving right until blank symbol is encountered. Once found move left and change 1 to 0 and keep moving right read sum from leftmost blank symbol.

The transition diagram is,



Example 7: Minus operation or proper subtraction on TM.

Let us,

show how a Turing machine might compute the function $\dot{-}$, which is called *minus* or *proper subtraction* and is defined by $m \dot{-} n = \max(m - n, 0)$. That is, $m \dot{-} n$ is $m - n$ if $m \geq n$ and 0 if $m < n$.

TM, M will start with a tape consisting of $0^m 1 0^n$ surrounded by blanks.

M halts with 0^{m-n} on its tape, surrounded by blanks.

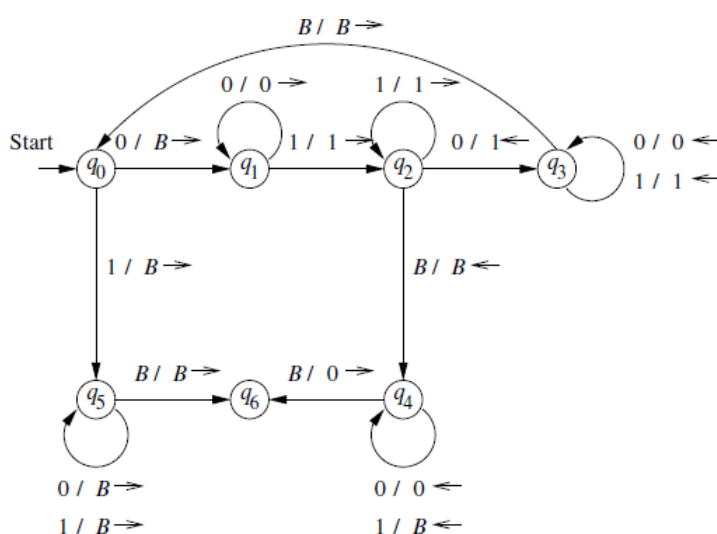
It, for example 5-3 can be represented as,

B000001000B

The result 5-3=2 will be represented on the tape as,

BBBBB00BBBB

The transition diagram is,



Here the symbol $\rightarrow$ indicates R(right)

and $\leftarrow$ indicates L (left)

The Language of a Turing Machine

In TM the input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not.

More formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. Then $L(M)$ is the set of strings w in Σ^* such that $q_0 w \vdash^* \alpha p \beta$ for some state p in F and any tape strings α and β .

The set of languages we can accept using a Turing machine is often called the **recursively enumerable languages or RE languages**.

Turing Machines and Halting Problem:

There is another notion of “acceptance” that is commonly used for Turing machines “acceptance by halting”. We say a TM halts if it enters a state q , scanning a tape symbol X , and there is no move in this situation i.e, $\delta(q, X)$ is undefined.

DECIDABILITY

- The term **decidability** defined using Turing machines. When a Turing machine reaches a final state, it halts.
- We can also say that a Turing machine M halts when M reaches a state q and a current symbol ‘ a ’ to be scanned so that $\delta(q, a)$ is undefined.
- There are TMs that never halt on some inputs in any one of these ways. So, we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

Definition of Recursively Enumerable Language:

“A language $L \subseteq \Sigma^*$ is recursively enumerable if there exists a TM M , such that $L = L(M)$ ”.

Definition of Recursive Language:

A language $L \subseteq \Sigma^*$ is recursive if there exists some TM M that satisfies the following two conditions.

- i) If $w \in L$ then M accepts w (that is, reaches an accepting state on processing w) and halts.
- ii) If $w \notin L$ then M eventually halts without reaching an accepting state

Definition Decidable: A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called **decidable**.

Definition Undecidable: A problem/language is **undecidable** if it is not decidable.

Programming Techniques for Turing Machines

The goal is to give a sense of how a Turing machine can be used to compute in a manner not unlike that of a conventional computer. Eventually, we want to convince you that a TM is exactly as powerful as a conventional computer.

8.3.1 Storage in the State

We can use the finite control not only to represent a position in the “program” of the Turing machine, but to hold a finite amount of data. Figure 8.13 suggests this technique (as well as another idea: multiple tracks). There, we see the finite control consisting of not only a “control” state q , but three data elements A , B , and C . The technique requires no extension to the TM model; we merely think of the state as a tuple. In the case of Fig. 8.13, we should think of the state as $[q, A, B, C]$. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

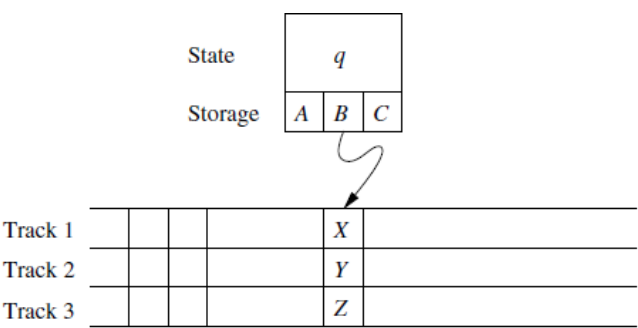


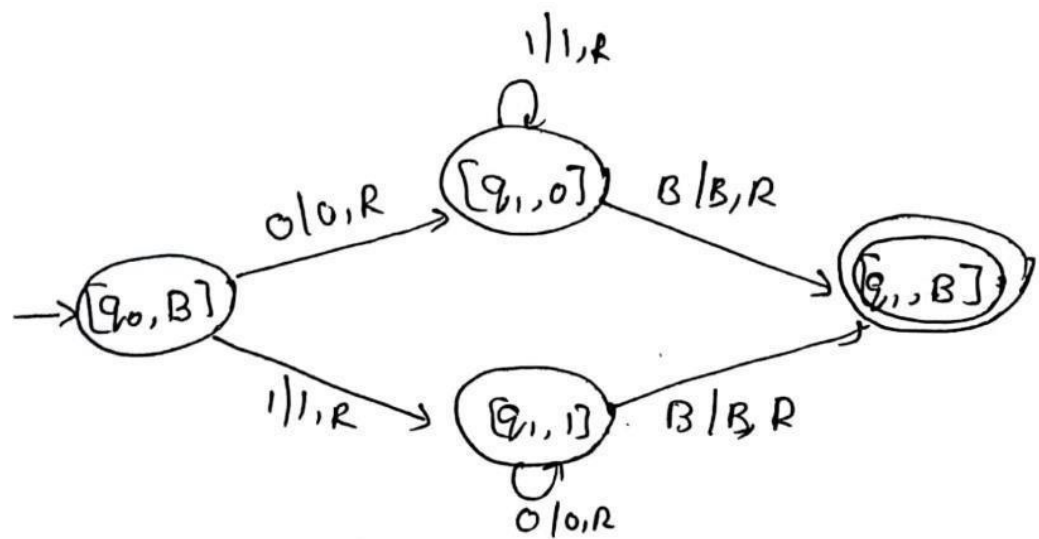
Figure 8.13: A Turing machine viewed as having finite-control storage and multiple tracks

Example 8.6: We shall design a TM

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $01^* + 10^*$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The transition diagram is,



Multiple Tracks

Another useful “trick” is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each “track”. Thus, for instance, the cell scanned by the tape head in Fig. 8.13 contains the symbol $[X, Y, Z]$. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do.

Extensions to the Basic Turing Machine:

a) Multi-tape Turing Machines

(Q. Write a short note on Multitape TM. (06M))

A multi-tape TM is as shown below. The device has a finite control “state”, and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet.

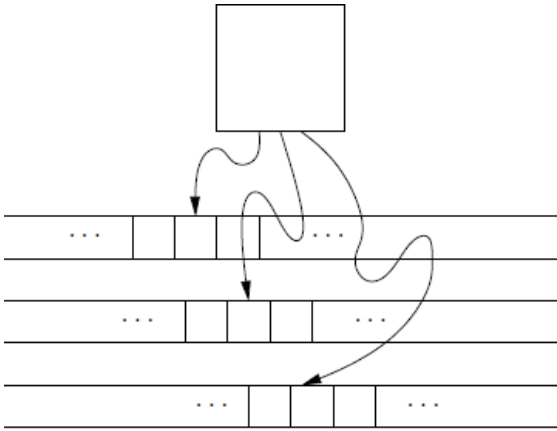


Figure 8.16: A multitape Turing machine

As in the single tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a

member. The set of states includes an initial state and some accepting states. Initially,

- The input, a finite sequence of input symbols, is placed on the first tape.
- All other cells of all the tapes hold the blank.
- The finite control is in the initial state.
- The head of the first tape is at the left end of the input
- All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes, “look” the same.

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

- The control enters a new state, which could be the same as the previous state.
- On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
- Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

Equivalence of one-tape TM and Multi-tape TM:

(Q. Prove that every language accepted by a multi-tape TM is recursively enumerable.5M)

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. Suppose language L is accepted by a k -tape TM M . We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks. Half these tracks hold the tapes of M , and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure 8.17 assumes $k = 2$. The second and fourth tracks hold the contents of the first and second tapes of M , track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

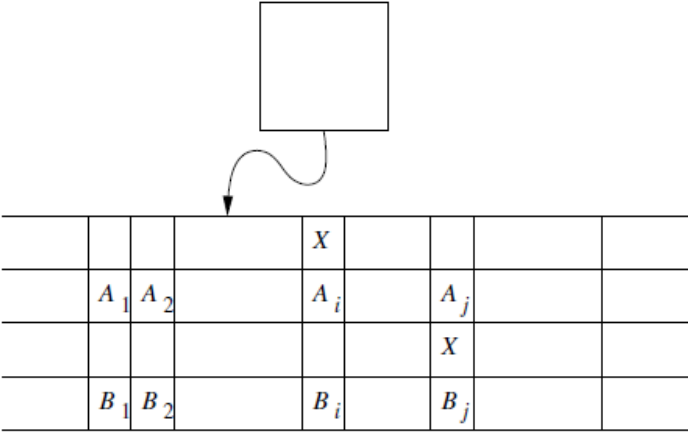


Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

To simulate a move of M , N 's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N 's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M 's heads. N also knows the state of M , which it stores in N 's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M , and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M .

We select as N 's accepting states all those states that record M 's state as one of the accepting states of M . Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise. $\square$

b) NONDETERMINISTIC TURING MACHINES

(Q. Write a short note on Nondeterministic TM)

In the case of standard Turing $\delta(q1, a)$ was defined for some elements of $Q \times \Gamma$ as an element of $Q \times \Gamma \times \{L, R\}$. Now we extend the definition of δ . In a nondeterministic TM, $\delta(q1, a)$ is defined as a subset of $Q \times \Gamma \times \{L, R\}$.

Definition: A nondeterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where,

- Q is a finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,

- $B \in \Gamma$ is the blank.
- Σ is a nonempty set of input symbols and is a subset of Γ and $B \notin \Sigma$.
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.
- δ is a partial function from $Q \times \Gamma$ into the power set of $Q \times \Gamma \times \{L, R\}$.

Undecidability

We then divide problems that can be solved by a Turing machine into two: classes, those that have an algorithmic, a Turing machine that halts whether or not it accepts its inputs, and those that are only solved by Turing machines that may run forever on inputs they do not accept.

A Language That Is Not Recursively Enumerable:

- A Language L is recursively enumerable (RE) if $L=L(M)$ for some TM M . A language is recursively enumerable if there exists a Turing machine that will enumerate all the strings in the language or accept strings in the language but may not halt for strings not in the language.
- A language L is recursive iff exists a Turing machine M such that
 - i) $L = L(M)$ and
 - ii) M always halt (even if it does not accept).

Definition: A language L is **undecidable** iff L is not recursive.

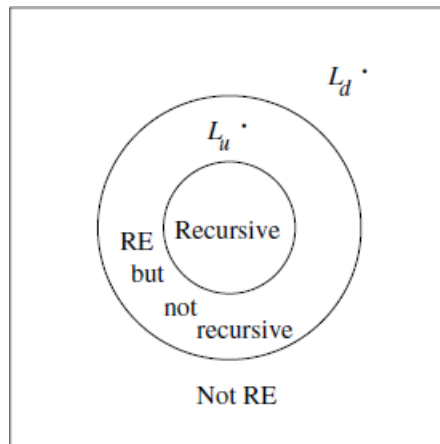
Note: The term “recursive” as synonym for “decidable” comes from Mathematics (prior to computers).

Non-Recursively Enumerable (Non-RE) Languages

Definition: A language is non-recursively enumerable if no Turing machine exists that can enumerate or accept all strings in the language. In other words, it is not possible to construct an algorithm to semi-decide membership.

- These are the most complex and "unsolvable" problems.
- The language and its complement are both non-recursively enumerable.

Following figure suggests the relationship among three classes of languages:



1. The recursive languages
2. The languages that are recursively enumerable but not recursive.
3. The non-recursively-enumerable(non-RE) languages.

$$\text{Recursive} \subseteq \text{RE} \subseteq \text{Non-RE}.$$

Undecidability often involves proving that certain problems cannot be solved by any algorithm (or equivalently, any Turing machine).

- To analyze such problems, we need a formal way to represent them as inputs to Turing machines. Binary coding provides a way to encode:
 - Strings in a language.
 - Descriptions of Turing machines.
 - Complex decision problems.

Example: The Halting Problem

- The Halting Problem asks whether a Turing machine halts on a given input.
- To analyze it, both the Turing machine (its states and transitions) and the input string are encoded into a single binary string. This encoding allows us to reason about whether a universal Turing machine (UTM) can solve the problem.

Codes for Turing Machines:

Let us devise a binary code for Turing machines so that each TM with input alphabet $\{0, 1\}$ may be thought of as a binary string.

Code for Turing Machine

To represent a TM, $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ as a binary string, we need to assign integers to the states, tape symbols and directions.

- Assume $Q = \{q_1, q_2, \dots, q_k\}$ are states for some k .
Let q_1 is the initial state. (we will not use q_0 , as numbering starts from 1).

- Assume $\Gamma = \{X_1, X_2, \dots, X_m\}$ for some m .

$$\text{Let } X_1 \Rightarrow 0$$

$$X_2 \Rightarrow 1$$

$$X_3 \Rightarrow B$$

- Assume directions of tape-head as,

$$D_1 \Rightarrow L$$

$$D_2 \Rightarrow R$$

Suppose T.M's transition rule is,

$$\delta(q_i, X_j) = (q_k, X_l, D_m) \text{ for some integers } i, j, k, l \text{ \& } m.$$

The code is : $\boxed{0^i 1 0^j 1 0^k 1 0^l 1 0^m}$ here 1 acts as separator.

Then the code for entire TM is:

$$C_1 \parallel C_2 \parallel \dots \parallel C_n$$

Example :

$$\text{Let } M = (\underbrace{\{q_1, q_2, q_3\}}_Q, \underbrace{\{0, 1\}}_\Sigma, \underbrace{\{0, 1, B\}}_\Gamma, \delta, q_1, B, \underbrace{\{q_3\}}_F)$$

$$\text{where } \delta = \{ \delta(q_1, 1) = (q_3, 0, R)$$

$$\delta(q_3, 0) = (q_1, 1, R)$$

$$\delta(q_3, 1) = (q_2, 0, R)$$

$$\delta(q_3, B) = (q_3, 1, L) \}$$

Let us convert individual δ into binary code as follows:

$$\bullet \delta(q_1, 1) = (q_3, 0, R) \Rightarrow \delta(q_1, X_2) = (q_3, X_1, D_2)$$

$$\therefore X_2 \Rightarrow 1$$

$$D_2 \Rightarrow R$$

$$X_1 \Rightarrow 0$$

$$X_3 \Rightarrow B$$

$$\therefore \text{code for } \delta = \overset{1}{0} \overset{2}{1} \overset{3}{0} \overset{1}{0} \overset{2}{1} \overset{2}{0}$$

- $\delta(q_3, 0) = (q_1, 1, R) \Rightarrow \delta(q_3, x_1) = (q_1, x_2, D_2)$
 $\therefore \text{code } c_2 = 0^3 1 0^1 1 0^1 1 0^2 1 0^2$
- $\delta(q_3, 1) = (q_2, 0, R) \Rightarrow \delta(q_3, x_2) = (q_2, x_1, D_2)$
 $\therefore \text{code } c_3 = 0^3 1 0^2 1 0^2 1 0^1 1 0^2$
- $\delta(q_3, B) = (q_3, 1, L) \Rightarrow \delta(q_3, x_3) = (q_3, x_2, D_1)$
 $\therefore \text{code } c_4 = 0^3 1 0^3 1 0^3 1 0^2 1 0^1$

$\therefore$ The code for entire T.M is,
 $c_1 || c_2 || c_3 || c_4$

i.e. 01001000101001100010101010010011
 00010010010100110001000100010010

Complements of Recursive and RE languages:

(Q. Prove that complement of recursive language is recursive)

Theorem: If L is a recursive language, so is $\bar{L}$

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\bar{M}$ such that $\bar{L} = L(\bar{M})$ by the construction suggested in Fig. 9.3. That is, $\bar{M}$ behaves just like M . However, M is modified as follows to create $\bar{M}$:

1. The accepting states of M are made nonaccepting states of $\bar{M}$ with no transitions; i.e., in these states $\bar{M}$ will halt without accepting.
2. $\bar{M}$ has a new accepting state r ; there are no transitions from r .
3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r .

Since M is guaranteed to halt, we know that $\bar{M}$ is also guaranteed to halt. Moreover, $\bar{M}$ accepts exactly those strings that M does not accept. Thus $\bar{M}$ accepts $\bar{L}$. $\square$

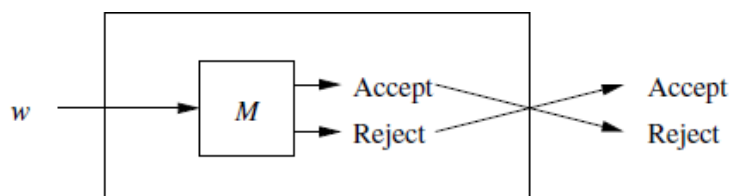


Figure 9.3: Construction of a TM accepting the complement of a recursive language

Q. Prove that complement of recursively enumerable language is

recursive.

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\bar{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\bar{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M . We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1 , while the other tape of M simulates the tape of M_2 . The states of M_1 and M_2 are each components of the state of M .

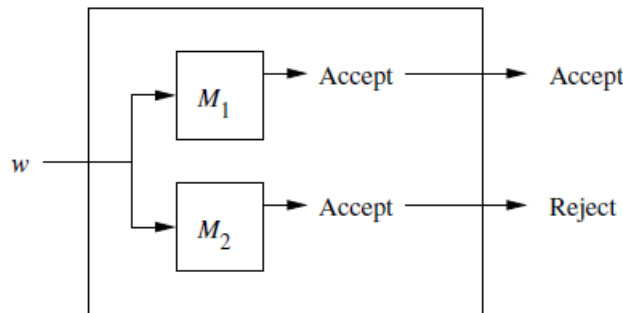


Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L , then M_1 will eventually accept. If so, M accepts and halts. If w is not in L , then it is in $\bar{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L . Since M always halts, and $L(M) = L$, we conclude that L is recursive. $\square$

The Universal Language:

A Turing machine could be used to simulate a computer that had been loaded with an arbitrary program. That is to say, a single TM can be used as a “stored program computer”, taking its program as well as its data from one or more tapes on which input is placed.

We define L_u , the universal language, to be the set of binary strings that encode, a pair (M, w) , where M is a TM with, the binary input alphabet, and w is a string in $(0+1)^*$ such that w is in $L(M)$.

That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U , often called the **universal Turing machine**, such that $L_u = L(U)$.

It is easiest to describe U as a multi-tape Turing machine. In the case of U , the transitions of M are stored initially on the first tape, along with the string w . A second tape will be used to hold the simulated tape of M , using the same format as for the code of M . That is, tape symbol X_i of M will be represented by 0^i and tape symbols will be separated by single 1's. The third tape of U holds the state of M , with state q_i

represented by i 0's. A sketch of U is in Fig. 9.5

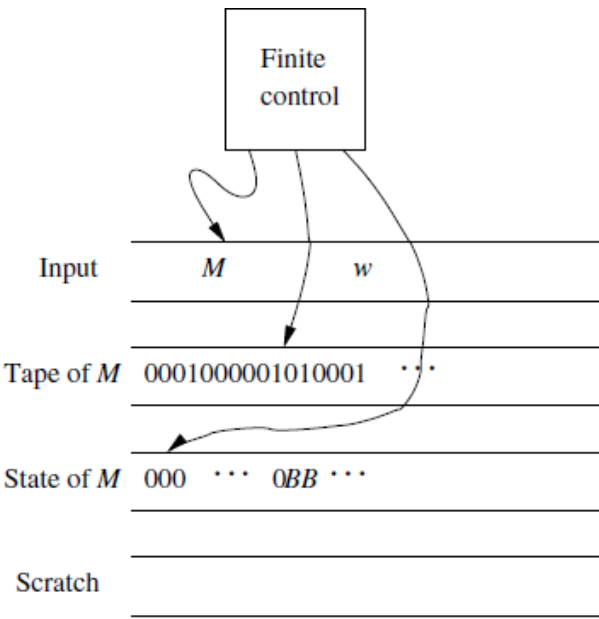


Figure 9.5: Organization of a universal Turing machine

The operation of U can be summarized as follows:

1. Examine the input to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting. Since invalid codes are assumed to represent the TM with no moves, and such a TM accepts no inputs, this action is correct.
2. Initialize the second tape to contain the input w , in its encoded form. That is, for each 0 of w , place 10 on the second tape, and for each 1 of w , place 100 there. Note that the blanks on the simulated tape of M , which are represented by 1000, will not actually appear on that tape; all cells beyond those used for w will hold the blank of U . However, U knows that, should it look for a simulated symbol of M and find its own blank, it must replace that blank by the sequence 1000 to simulate the blank of M .
3. Place 0, the start state of M , on the third tape, and move the head of U 's second tape to the first simulated cell.
4. To simulate a move of M , U searches on its first tape for a transition $0^i 10^j 10^k 10^l 10^m$, such that 0^i is the state on tape 3, and 0^j is the tape

symbol of M that begins at the position on tape 2 scanned by U . This transition is the one M would next make. U should:

- (a) Change the contents of tape 3 to 0^k ; that is, simulate the state change of M . To do so, U first changes all the 0's on tape 3 to blanks, and then copies 0^k from tape 1 to tape 3.
 - (b) Replace 0^j on tape 2 by 0^l ; that is, change the tape symbol of M . If more or less space is needed (i.e., $i \neq l$), use the scratch tape and the shifting-over technique of Section 8.6.2 to manage the spacing.
 - (c) Move the head on tape 2 to the position of the next 1 to the left or right, respectively, depending on whether $m = 1$ (move left) or $m = 2$ (move right). Thus, U simulates the move of M to the left or to the right.
5. If M has no transition that matches the simulated state and tape symbol, then in (4), no transition will be found. Thus, M halts in the simulated configuration, and U must do likewise.
 6. If M enters its accepting state, then U accepts.

In this manner, U simulates M on w . U accepts the coded pair (M, w) if and only if M accepts w .
