Lecture 1

Deterministic Context Free Languages(DCFLs) and Properties of DCFL

Lecture-21	**PO 1 & PSO 1**

The languages recognized by the deterministic pushdown automata (DPDAs) are known as deterministic context-free languages (DCFLs). This class of language is a subclass of the context-free languages (CFLs). So, we can say that the DPDAs accept a class of languages that is between the class of regular languages and the CFLs. The application area of DCFLs include the design of parsers in compilers for programming languages (because the parsing problem is generally easier for DCFLs than for CFLs) as they can be parsed in linear time, and various restricted forms of DCFGs admit simple practical parsers as well.

1.0.1 PROPERTIES OF DCFLs

Closure and non-closure properties of the class of DCFLs

THEOREM: The class of DCFLs is closed under complementation. That means, if L is a DCFL then the complement of L i.e., L^c is also a DCFL.

Proof Idea: The approach used in DFA i.e., swapping the accept and non-accept states of a DFA generates a new DFA that recognizes the complementary language, thereby proving that the class of regular languages is closed under complementation. The same approach can be used to prove this aspect of DPDAs, except for one problem. The DPDA may accept its input by entering both accept (acceptance by accept state) and non-accept states (acceptance by empty stack) in a sequence of moves at the end of the input string. Interchanging accept and non-accept states would still accept in this case.

To fix this problem, we modify the DPDA to limit when acceptance can occur. For each symbol of the input, the modified DPDA can enter an accept state only when it is about to read the next symbol. In other words, only reading states– *the states that*

always read an input symbol— may be accept states. Then, by swapping acceptance and non-acceptance only among these reading states, we invert the output of the DPDA.

Lecture 2

Deterministic context-free grammars, Relationship of Deterministic PDAs and DCFGs

Lecture-22	**PO 1 & PSO 1**

- Deterministic context-free grammars (DCFGs) are always unambiguous and are an important subclass of unambiguous CFGs. So, DCFLs are always unambiguous.
- However, unambiguous grammars do not always generate a DCFL.
- For example, the language of even-length palindromes on the alphabet of 0 and 1 i.e., $\{ww^r|w \in (0,1)^*\}$ has the unambiguous context-free grammar $S \rightarrow 0S0|1S1|\epsilon$.

  But, an arbitrary string of this language cannot be parsed without reading all its letters first. This means that a PDA has to try alternative state transitions to accommodate for the different possible lengths of a semi-parsed string. So, clearly this is not a DCFL.

2.0.1 Relationship of DCFGs and DPDAs

- DCFGs are the counterparts of DPDAs. That means, these two models are equivalent in power, provided that the language is an endmarked language where all strings are terminated with $\dashv$
- Consider an example:

  Let L = $a^* \bigcup\{\mathrm{a}^n b^n|n > 0\ \}$.

  When the DPDA begins reading a's, it must push them onto the stack.

  In case there are going to be b's it runs out of input without seeing b's, it needs a way to pop the a's from the stack before it can accept. In this case the end-of-string marker $\dashv$ allows the popping to happen, when all the input has been read.

- So, adding $\dashv$ at the end of the string makes it easier to build DPDAs, because we can take advantage of knowing when the input string ends. However, it does not add power to DPDA. DPDA for L = $a^* \bigcup \{a^n b^n | n > 0\}$ is shown below.

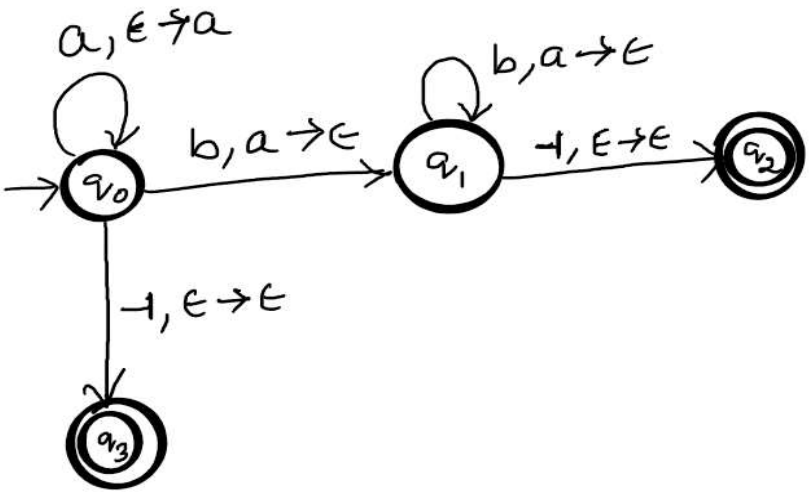

Figure 2.1: DPDA for L = $a^* \bigcup \{a^n b^n | n > 0\}$

Lecture 3

Deterministic PDAs

Lecture-23	**PO 1 & PSO 1**

3.0.1 DPDA

PDAs are by definition allowed to be non-deterministic but its deterministic subcase is also important. Basically, parsers in compilers behave like the deterministic PDAs. So, the class of languages that can be accepted by DPDAs are interesting since it gives us the insights about various constructs suitable for use in programming languages, i.e., the in compilers. So now we will discuss deterministic PDAs and discuss some of the things they can and cannot do.

3.0.2 Definition of DPDA

Basically, a PDA with deterministic computation is known as a DPDA i.e., there is never a choice of more than one move in any situation. Meaning, the DPDA has at most one way to proceed at each step of its computation just like the DFA. However, differently from DFAs they may haveϵ transitions from any state Q at any moment. And theseϵ-moves can be a ϵ-input moves ((q,ϵ, x)) ,ϵ-stack moves ((q, a,ϵ)) or may be the combination of both the moves ((q,ϵ,ϵ)). So, when a DPDA makes aϵ-move, at that point it is prohibited from making any other move that involves processing an input symbol other thanϵ in order to ensure deterministic nature is preserved.

Thus, we can formally define a DPDA D with a 6-tuple notation (Q, Σ, Γ, δ, q0, F), in which

1. Q is its set of states,
2. Σ its input alphabet,
3. Γ its stack alphabet,
4. δ its transition function, where δ is defined as: Q $\times$ $\Sigma_{\{}\epsilon\}\times\Gamma_{\{}\epsilon\}\rightarrow (Q\times\Gamma_{\{}\epsilon\})\bigcup\{\phi\}$
5. q0 the start state, and

6. F $\subseteq$ Q the set of accept states.

And it satisfies the following condition;

For every $q \in Q$,$a \in \Sigma$ and $x \in \Gamma$, exactly one of the values (q, a, x), (q, a, ϵ), (q, ϵ, x), and (q, ϵ, ϵ) is not ϕ. That means if the value of the transition (q, a, x) is not ϕ for any state q, input symbol a and stack symbol x, the value for other transitions (q, a, ϵ), (q, ϵ, x), and (q, ϵ, ϵ) should be empty or ϕ.

The acceptance mechanism for DPDAs similar to the acceptance of PDAs. If a DPDA has finished reading the last input symbol of an input string and enters to an accept state, then it accepts that string. In all other cases, it rejects that string. Basically, rejection in a DPDA occurs if the it reads the entire input but doesn't enter an accept state when it is at the end of the string, or if the DPDA fails to read the entire input string. The latter case may arise if the DPDA tries to pop an empty stack or if the DPDA makes an endless sequence of -input moves without reading the input past a certain point.

3.0.3 Examples of DPDA

1. Consider the language L=$\{0^n1^n|n \geq 0\}$ and construct a DPDA

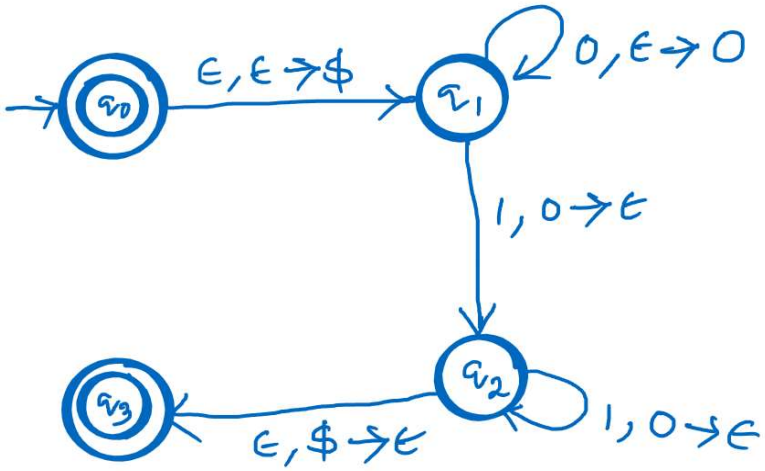

Figure 3.1: DPDA for L=$\{0^n1^n|n \geq 0\}$

2. Consider the language L=$\{wcw^r|w \in (0,1)^*\}$ and construct a DPDA

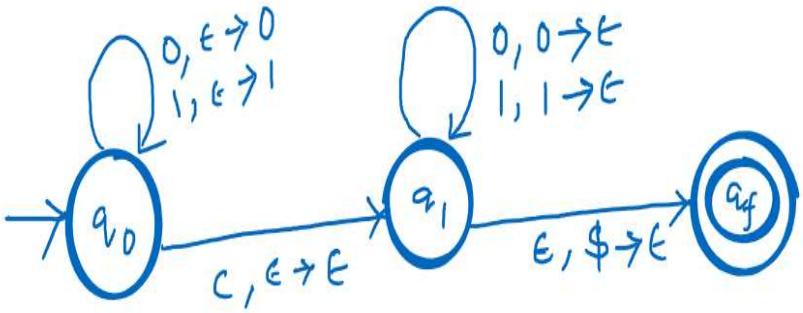

Figure 3.2: DPDA for L=$\{wcw^r | w \in (0,1)^*\}$

Lecture 4

Introduction to general-purpose computing

Lecture-24	**PO 1 & PSO 1**

Till now, we have discussed several models of computing devices. Firstly, we saw Finite Automata (FAs), basic model that works for devices that have a small amount of memory. Further, we have shown that some very simple tasks are beyond the capabilities of FAs. Then, we discussed the Pushdown automata (PDAs), the models for devices that have an unlimited memory that is usable only in the last in, first out manner of a stack. We have discussed some of the tasks which are beyond the capabilities of PDAs. Hence, they are too restricted to serve as models of general-purpose computers.

4.0.1 Turing Machine

We will now discuss about a much more powerful model i.e., the Turing Machine. It was first proposed by Alan Turing in 1936. A TM is just like a finite automaton but with an unlimited and unrestricted memory. So it is a much more accurate model of a general purpose computer and can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems in the field of computation (In reality, these problems are beyond the theoretical limits of computation).

The Turing machine uses an infinite tape as its unlimited memory, a tape head that can read and write symbols and move around (in left and right directions) on the tape. It uses a finite length of the tape to store the input string to be processed. So, initially the tape contains only the input string within finite number of cells of the infinite tape and all other cells of the tape are blanks. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing while moving back and forth until it decides to produce an output. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will loop forever, i.e, it will never halt.

We may visualize a Turing machine as in Fig4.1. The machine consists of a finite

control, which can be in any of a finite set of states. There is a tape divided into cells, where each cell can hold any one of a finite number of symbols

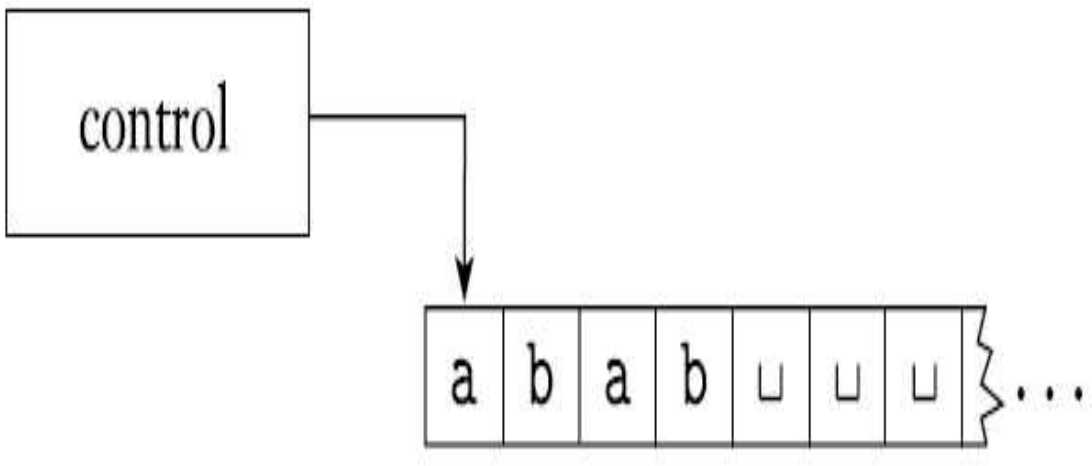

Figure 4.1: Schematic of a Turing machine

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.

The following list summarizes the differences between finite automata and Turing machines.

1. A Turing machine can both write on the tape and read from it.
2. The read–write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take effect immediately.

The tape head 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

1. Changes the state. The next state may be the same as the current state
2. Writes a tape symbol in the cell scanned (pointed by the tape head). This tape symbol replaces whatever symbol was in that cell and the symbol written may be the same as the symbol currently there

3. Moves the tape head left or right

Let's introduce a Turing machine M_1 for testing membership in the language L= $\{w\#w|w \in (0,1)^*\}$. We want M_1 to accept if its input is a member of L and to reject otherwise. To understand M_1 better, put yourself in its place by imagining that you are standing on a mile-long input consisting of millions of characters. Your goal is to determine whether the input is a member of L—that is, whether the input comprises two identical strings separated by a symbol. The input is too long for you to remember it all, but you are allowed to move back and forth over the input and make marks on it. The obvious strategy is to zig-zag to the corresponding places on the two sides of the and determine whether they match. Place marks on the tape to keep track of which places correspond.

The machine M_1 makes multiple passes over the input string with the read–write head. On each pass it matches one of the characters on each side of the # symbol. To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined. If it crosses off all the symbols, that means that everything matched successfully, and M_1 goes into an accept state. If it discovers a mismatch, it enters a reject state. In summary, M_1's algorithm is as follows. "On input string w: the TM M_1

1. Zig-zag across the tape to corresponding positions on either side of the symbol to check whether these positions contain the same symbol. If they do not, or if no is found, reject . Cross off symbols as they are checked to keep track of which symbols correspond

2. When all symbols to the left of the have been crossed off, check for any remaining symbols to the right of the . If any symbols remain, *reject*; otherwise, *accept*."

The following figure contains several nonconsecutive snapshots of M_1's tape after it is started on input 01100#011000.

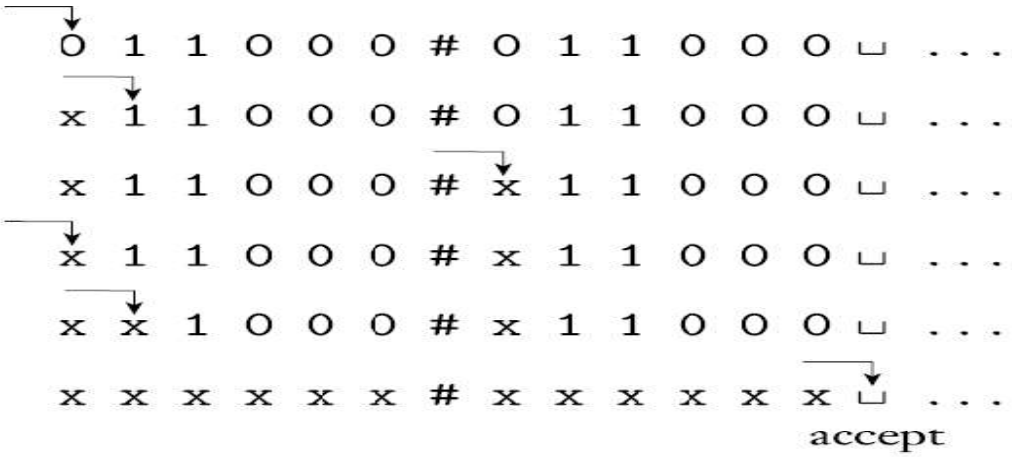

Figure 4.2: Snapshots of Turing machine M_1 computing on input 011000#011000

This description of Turing machine M_1 sketches the way it functions but does not give all its details. We can describe Turing machines in complete detail by giving formal

descriptions analogous to those introduced for FA and PDA. The formal descriptions specify each of the parts of the formal definition of the Turing Machine model to be presented shortly. In actuality, we almost never give formal descriptions of Turing Machines because they tend to be very big.

FORMAL DEFINITION OF A TURING MACHINE

We can formally define a TM M with a 7-tuple notation (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}), in which

1. Q is its set of states,
2. Σ its input alphabet not containing the blank symbol,
3. Γ its Tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. δ its transition function, where δ is defined as: $Q \times \Gamma \rightarrow Q \times \Gamma$ { L, R },
5. $q_0 \in Q$ the start state,
6. $q_{accept} \in Q$ the accept states, and,
7. $q_{reject} \in Q$ the reject state, where $q_{reject} \neq q_{accept}$

A Turing machine M = (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}) computes as follows.

Initially, M receives its input $w = w_1 w_2 ... w_n$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape. Note that does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a configuration of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet , we write u q v for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v. For example, $1011q_701111$ represents the configuration when the tape is 101101111, the current state is q_7, and the head is currently on the second 0. Following figure depicts a Turing machine with that configuration.

Here we formalize our intuitive understanding of the way that a Turing machine computes. Say that configuration C_1 yields configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We define this notion formally as follows.

Suppose that we have a, b, and c in Γ, as well as u and v in Γ^* and states q_i and q_j. In that case, $ua\ q_i\ bv$ and $u\ q_j\ acv$ are two configurations. Say that $ua\ q_i\ bv$ yields $u\ q_j\ acv$ if in the transition function $\delta(q_i, b) = (q_j, c, L)$. That handles the case where the Turing machine moves leftward. For a rightward move, say that $ua\ q_i\ bv$ yields $uac\ q_j\ v$ if $\delta(q_i, b) = (q_j, c, R)$.

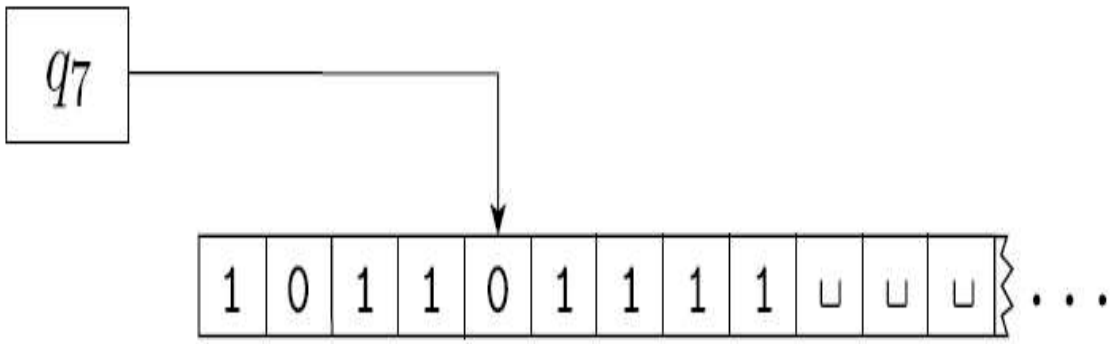

Figure 4.3: A Turing machine with configuration 1011q701111

Special cases occur when the head is at one of the ends of the configuration. For the left-hand end, the configuration $q_i\ bv$ yields $q_j\ cv$ if the transition is left moving (because we prevent the machine from going off the left-hand end of the tape), and it yields $c\ q_j\ v$ for the right-moving transition. For the right-hand end, the configuration $ua\ q_i$ is equivalent to $ua\ q_i$ because we assume that blanks follow the part of the tape represented in the configuration. Thus, we can handle this case as before, with the head no longer at the right-hand end.

The start configuration of M on input w is the configuration q0 w, which indicates that the machine is in the start state q0 with its head at the leftmost position on the tape. In an accepting configuration, the state of the configuration is q_{accept}. In a rejecting configuration, the state of the configuration is q_{reject}. Accepting and rejecting configurations are halting configurations and do not yield further configurations. Because the machine is defined to halt when in the states q_{accept} and q_{reject}, we equivalently could have defined the transition function to have the more complicated form δ is defined as: Q' $\times\ \Gamma \rightarrow$ Q $\times\ \Gamma\ \times$ { L, R }, where Q' is Q without q_{accept} and q_{reject}. A Turing machine M accepts input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

4.0.2 The Language of a Turing Machine

The collection of strings that M accepts is the language of M, or the language recognized by M, denoted L(M).

Definition 1

A language is ***Turing-recognizable*** if some Turing machine recognizes it.

When we start a Turing machine on an input, three outcomes are possible. The machine may accept, reject, or loop. By loop we mean that the machine simply does not halt.

Looping may entail any simple or complex behavior that never leads to a halting state. A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called deciders because they always decide to accept or reject.

It is also called a ***recursively enumerable language (REL)***.

Definition 2

A language is ***Turing-decidable*** or simply decidable if some Turing machine decides it. It is also called a ***recursive*** language.

Every decidable language is Turing recognizable. We present examples of languages that are Turing recognizable but not decidable after we develop a technique for proving undecidability. Next, we will discuss some of the decidable languages.

Lecture 5

Examples of Turing Machines

Lecture-25	PO 1 & PSO 1

5.1 Example #1

The following is a formal description of $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, the Turing machine that we informally described for deciding the language $A = \{w\#w | w \in (0,1)^*\}$. $Q = \{q_1, \ldots, q_8, q_{accept}, q_{reject}\}$, $\Sigma = \{0, 1, \#\}$, and $\Gamma = \{0, 1, \#, x, \sqcup\}$. We describe δ with a state diagram (refer the figure). The start, accept, and reject states are q_1, q_{accept} and q_{reject}, respectively.

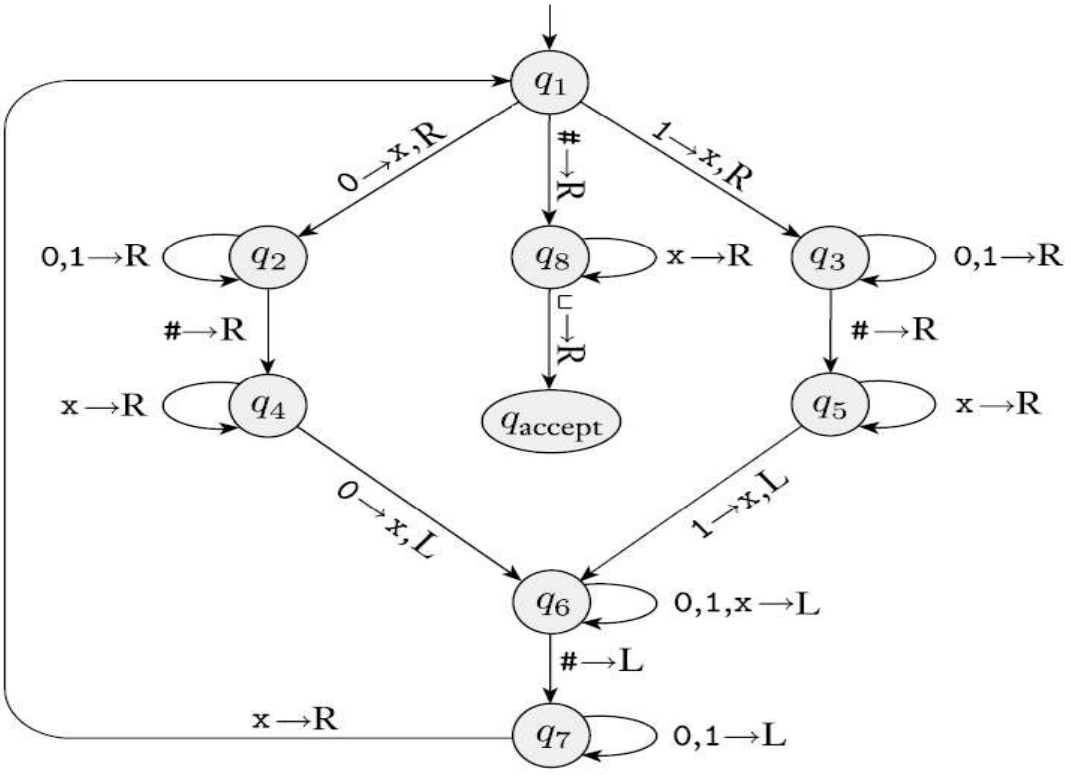

Figure 5.1: State diagram for Turing machine M_1

In the state diagram of TM M_1, you will find the label 0,1→R on the transition going

from q_3 to itself. That label means that the machine stays in q_3 and moves to the right when it reads a 0 or a 1 in state q_3. It doesn't change the symbol on the tape.

Stage 1 is implemented by states q_1 through q_7, and stage 2 by the remaining states. To simplify the figure, we don't show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol. Thus, because in state q_5 no outgoing arrow with a is present, if a occurs under the head when the machine is in state q_5, it goes to state q_{reject}. For completeness, we say that the head moves right in each of these transitions to the reject state.

5.2 Example #2

The following is a formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, the Turing machine that we informally described for deciding the language $B = \{0^{2^n} | n >= 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

$Q = \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$, $\Sigma = \{0\}$, and $\Gamma = \{0, x, \sqcup\}$.We describe δ with a state diagram (refer the figure). The start, accept, and reject states are q_1, q_{accept} and q_{reject}, respectively.

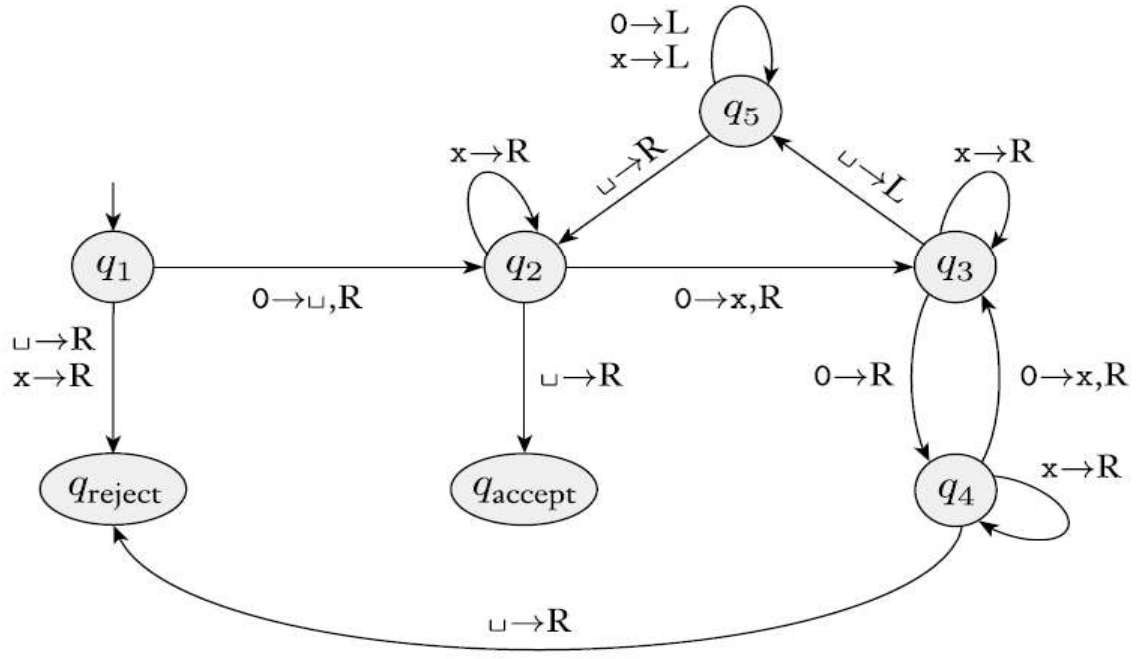

Figure 5.2: State diagram for Turing machine M_2

M_2 = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.

2. If in stage 1 the tape contained a single 0, accept.

3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.

4. Return the head to the left-hand end of the tape.

5. Go to stage 1."

Each iteration of stage 1 cuts the number of 0s in half. As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd. If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2. Therefore, the machine rejects in this instance. However, if the number of 0s seen is 1, the original number must have been a power of 2. So, in this case, the machine accepts.

In this state diagram, the label $0 \to \sqcup$,R appears on the transition from q_1 to q_2. This label signifies that when in state q_1 with the head reading 0, the machine goes to state q_2, writes $\sqcup$, and moves the head to the right. In other words, $\delta(q_1,0) = (q_2,\sqcup,R)$. For clarity we use the shorthand $0 \to R$ in the transition from q_3 to q_4, to mean that the machine moves to the right when reading 0 in state q_3 but doesn't alter the tape, so $\delta(q_3,0) = (q_4,0,R)$.

This machine begins by writing a blank symbol over the leftmost 0 on the tape so that it can find the left-hand end of the tape in stage 4. Whereas we would normally use a more suggestive symbol such as for the left-hand end delimiter, we use a blank here to keep the tape alphabet, and hence the state diagram, small.

Next, we give a sample run of this machine on input 0000. The starting configuration is $q_1 0000$. The sequence of configurations the machine enters appears as follows; read down the columns and left to right.

$q_1 0000$	$\sqcup q_5 \mathrm{X} 0 \mathrm{X} \sqcup$	$\sqcup \mathrm{X} q_5 \mathrm{X} \mathrm{X} \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup \mathrm{X} 0 \mathrm{X} \sqcup$	$\sqcup q_5 \mathrm{X} \mathrm{X} \mathrm{X} \sqcup$
$\sqcup \mathrm{X} q_3 00$	$\sqcup q_2 \mathrm{X} 0 \mathrm{X} \sqcup$	$q_5 \sqcup \mathrm{X} \mathrm{X} \mathrm{X} \sqcup$
$\sqcup \mathrm{X} 0 q_4 0$	$\sqcup \mathrm{X} q_2 0 \mathrm{X} \sqcup$	$\sqcup q_2 \mathrm{X} \mathrm{X} \mathrm{X} \sqcup$
$\sqcup \mathrm{X} 0 \mathrm{X} q_3 \sqcup$	$\sqcup \mathrm{X} \mathrm{X} q_3 \mathrm{X} \sqcup$	$\sqcup \mathrm{X} q_2 \mathrm{X} \mathrm{X} \sqcup$
$\sqcup \mathrm{X} 0 q_5 \mathrm{X} \sqcup$	$\sqcup \mathrm{X} \mathrm{X} \mathrm{X} q_3 \sqcup$	$\sqcup \mathrm{X} \mathrm{X} q_2 \mathrm{X} \sqcup$
$\sqcup \mathrm{X} q_5 0 \mathrm{X} \sqcup$	$\sqcup \mathrm{X} \mathrm{X} q_5 \mathrm{X} \sqcup$	$\sqcup \mathrm{X} \mathrm{X} \mathrm{X} q_2 \sqcup$
		$\sqcup \mathrm{X} \mathrm{X} \mathrm{X} \sqcup q_{\text{accept}}$

Figure 5.3: Sample run for Turing machine M_2 on input 0000