14 TM Extensions and Languages

'Increment in knowledge results in the enhanced versions of the systems concerned'

Introduction

In the previous chapter, we discussed the more powerful model of computation through Turing machine, along with a number of examples pertaining to the cases:

a. Turing machine as an accepting device for the languages
b. Turing machine as a computer of functions.

In this chapter, first we discuss a number of extensions, rather than the enhanced versions of the standard TM and later, the languages accepted by a TM and also its properties through **recursive** and **recursively enumerable** languages. Each of the extensions discussed here is no doubt a powerful model, but is equivalent to the standard TM.

14.1 Extensions of TM

In the previous chapter, the Turing machine defined is referred to as the *Standard turing machine*, because the tape is assumed to be arbitrarily extensible to the left as well as to the right and the machine is supplied with enough tape required for computation.

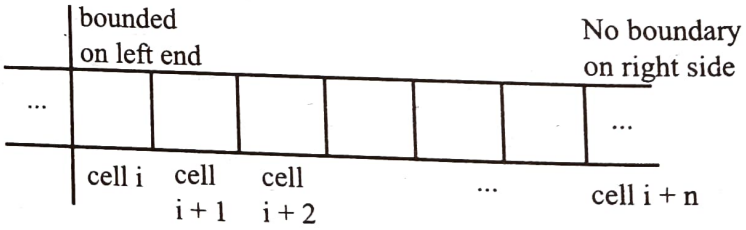

Figure 14.1. *Tape in a Standard TM*

The cell *i*, in figure 14.1, indicates that it is the leftmost cell provided to the machine for the current computation, but there is no boundary on the right side of the tape.

If, for the standard TM, the following characteristics are considered, then we get different types of TM and these are called the extensions of TM. The characteristics are:

a. The tape may be allowed to be infinite in both directions.

b. There may be more than one head, processing various cells of the tape.

c. There may be several tapes, each having its own independent head.

d. The tape may be k-dimensional, instead of only one-dimensional.

e. The machine may adopt the nondeterministic approach.

Accordingly, the following are the extensions of a TM:

14.1.1 *Two-Way Infinite Tape TM*

A TM in which there is an infinite number of sequences of blanks on either side of the tape, is said to be two-way infinite tape TM. The advantage of having this type of TM is that there is no possibility of jumping off the left end of the tape.

The initial configurations of the tape for the input string 101, with respect to a standard TM and two-way infinite tape TM, are shown below:

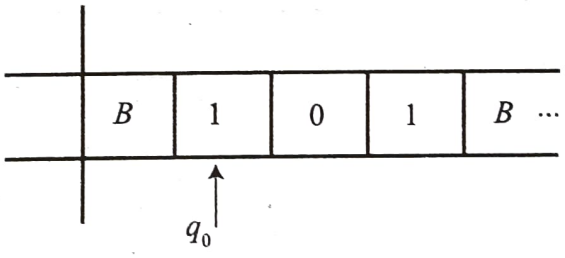

Figure 14.2. *Standard TM with Boundary on Left-Side*

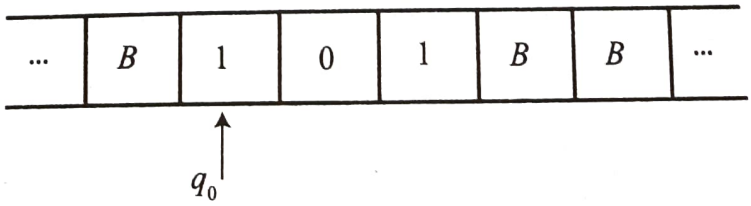

Figure 14.3. *Two-Way TM with No Boundaries*

Formally, a two-way infinite tape TM is represented as 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, h)$$

where

Q is a finite set of states.
Σ is the finite set of non-blank symbols.
Γ is the set of tape characters.
$q_0 \in Q$ is the initial state.
B is the blank character.
$h \sqsubseteq Q$ is the final state and
$\delta : Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, N\}$.

14.1.2 *Multi-Tape Turing Machine*

A TM with more than one tape and each tape having its own independent head is said to be a multi-tape TM.

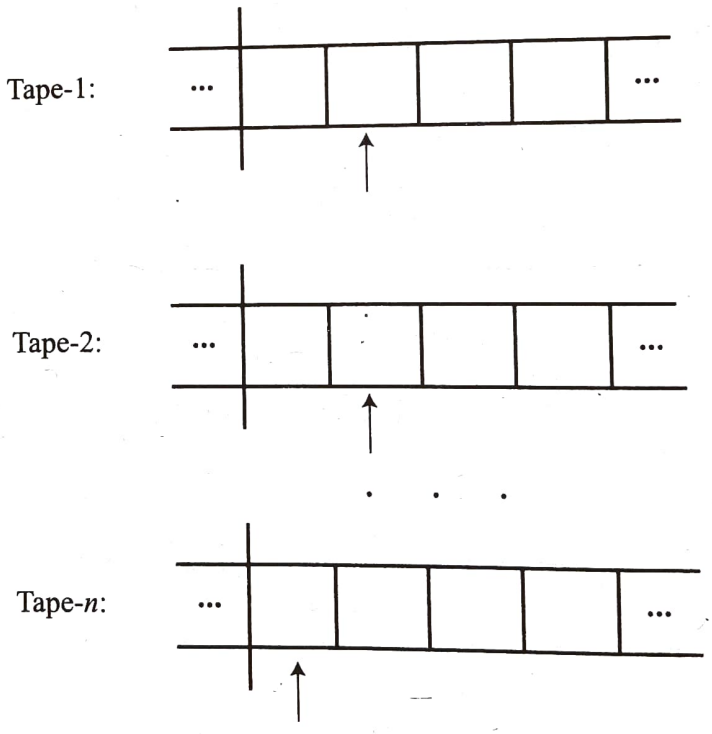

Figure 14.4. *A Multi-Tape TM*

Initially, the input appears on tape 1 and the others start out blank. The transition function is changed to allow for reading, writing, and moving the heads on all the tapes simultaneously.

Formally, a multi-tape TM is represented as 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, h)$$

where

Q is the finite set of states.

Σ is the finite set of non-blank symbols.

Γ is the set of tape characters.

$q_0 \in Q$ is the initial state.

B is the blank character.

$h \sqsubseteq Q$ is the final state and

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

(k is the number of tapes).

The advantage of having multi-tape TM is that the design of some functions like copying, reversing, verifying whether a string is a palindrome or not etc., can be carried out in much easier way as compared to the design of the corresponding standard TMs.

14.1.3 *Multi-Head TM*

A TM with one tape and several heads on it is said to be a multi-head turing machine.

In a multi-head TM having k heads ($k \geq 2$ – the heads are numbered 1 through k), a move of TM depends on the state and on the symbol scanned by each head. In one move, each head can move independently to left or right or remain stationary.

Also, the use of multi-heads (like multi-tapes) can sometimes simplify the construction of complex TMs drastically.

14.1.4 *K - Dimensional TM*

A TM having its tape with an infinite k-dimensional grid, is said to be a k-dimensional TM.

In this type of TM, the tape consists of a k-dimensional array of cells, infinite in all 2-k directions, for some fixed k. Depending on the state and symbol scanned, the machine changes its states, writes a new symbol and moves its tape - head in one of the 2-k directions, either positively or negatively along one of the k-axes. Initially, the input is along one axis and the head is at the left end of the input.

At any time, only a finite number of rows in any dimension contain non-blank symbols and each of these rows has only a finite number of non-blank symbols.

Figure 14.5 shows a two-dimensional tape, having boundaries on the left and the bottom. Each cell is given an address (x, y) where x is the row number of the cell and y is the column number of the cell.

14.2.1 *A Random Access TM*

A random access TM has a fixed number of registers and a one-way infinite type. Each register and each tape cell is capable of containing an arbitrary natural number. A sequence of instructions, called *program*, acts on its tape cell and its registers, with a program counter that indicates the address of the instruction to be executed next. The operations of random access TM will end, when a 'halt' instruction is executed.

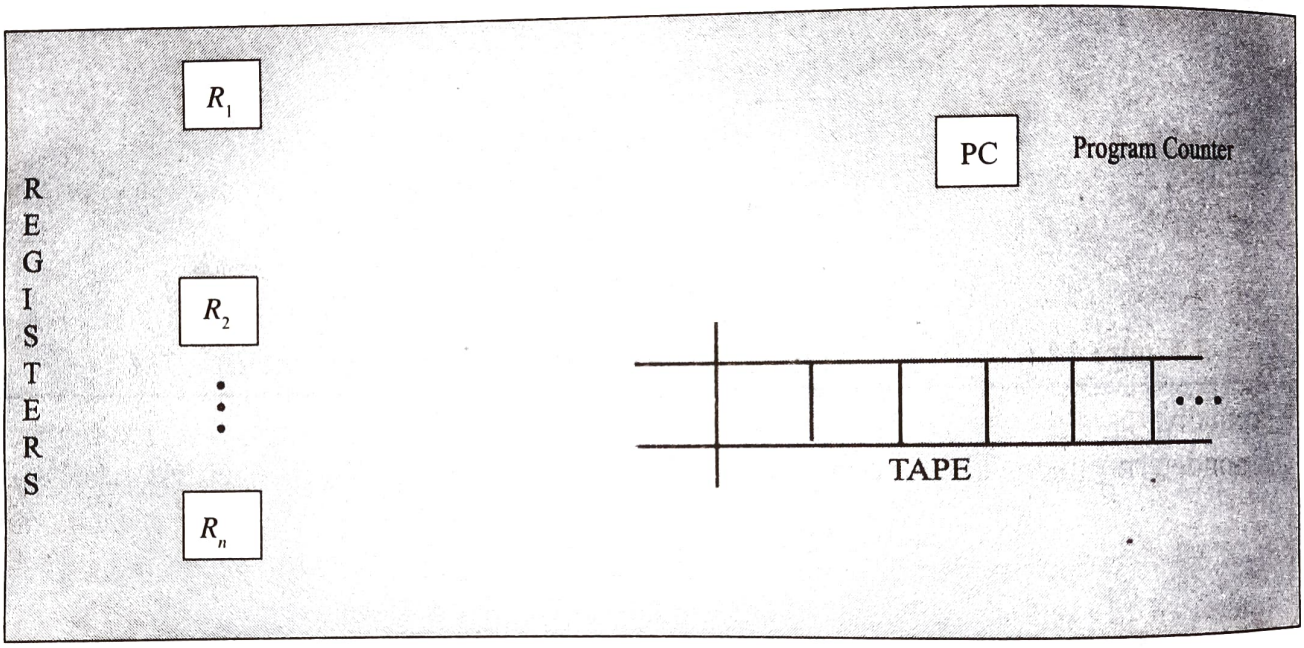

Figure 14.7. *Random Access TM*

14.2.2 *Universal Turing Machine*

A universal Turing machine is one that is capable of mimicking the action of any other turing machine.

Consider a fixed Turing machine U, with the property that for each and every other TM, T_f (that computes $f(x_1, \ldots, x_n)$), there is a string of symbols λ_T that describes the states and transitions of T_f, such that the unary number $x_1, \ldots, x_n$ is written on the tape followed by a delimiter symbol and the symbol λ_T. If U is started in the state Q_0 on the leftmost symbol of $\lambda_T, f(x_1, \ldots, x_n)$ will be written on the U tape when U stops, where $f(x_1, \ldots, x_n)$ is the output of T_f.

A universal TM does exactly what a real modern computer does. In other words, a universal TM executes some algorithms on an input $(x_1, \ldots x_n)$ and outputs the results.

14.2.3 *Alternating Turing Machine*

A nondeterministic TM, which consists of universal states, is said to be an alternating TM. The idea is that if all possible transitions from a universal state leads to an acceptance, then TM accepts.

14.2.4 *Probabilistic Turing Machine*

A turing machine that contains randomly chosen transitions is a probabilistic TM.

14.2.5 *Oracle Turing Machine*

It is a special kind of TM that consists of an extra tape and 3 additional states $q_?$, q_y and q_n. Upon entering the state $q_?$, an oracle is consulted to produce an answer (in a single step) to a given input on the tape. If the input is in the oracle set, then the control goes to the state q_y, otherwise it goes to q_n.

14.3 Linear Bounded Automata

A linear bounded automaton (LBA) is a restricted form of a non-deterministic TM. It possesses

a. a tape made up of cells, that can contain symbols from a finite alphabet
b. a head that can read from or write to one cell on the tape at a time (and can be moved)
c. a finite number of states.

It differs from a TM in the sense that (while the tape is initially considered infinite) only a finite continuous portion of tape (the length of which is a linear function of the length of the initial input) can be accessed by the read/write head. This limitation makes an LBA a more accurate model of computers that actually exists, than a TM in some respects.

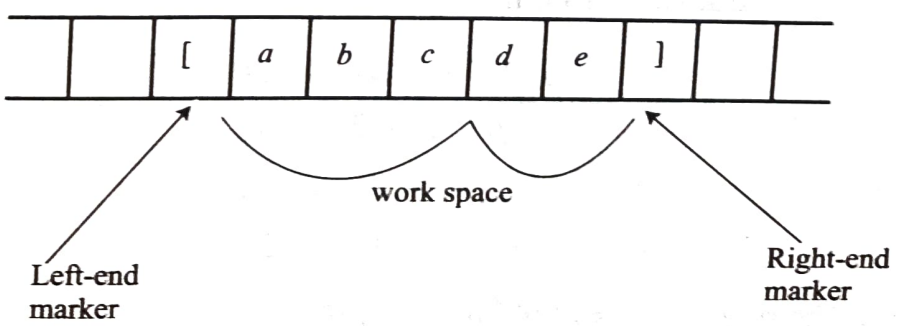

Figure 14.8. *LBA: All Computation is Done Between End Markers*

Formally, the LBA is a TM:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, h)$$

where

Q is the set of finite states.
Σ is the set of input alphabets which also has two special symbols '[' and ']'.
Γ is the set of tape symbols.
δ is the transition function from $Q \times \Gamma$ to $Q \times 2^{Q \times \Gamma \times \{L,R\}}$ with two more transitions of the form $\delta(q_i, [) = (q_j, [, R)$ and $\delta(q_i,]) = (q_j,], L)$ forcing the read/write head to be within the boundaries '[' and ']'
q_0 is the start state.
B is a special symbol indicating blank character.
$h \in Q$ is the set of final states.

The LBAs are same as TM, except that the input string tape space is the only tape space allowed to use. The LBA are acceptors for the class of **context-sensitive languages**.

14.4 Encoding Turing Machines

There are many ways to encode a turing machine. One of the simplest method is **binary encoding**. The following are the steps required in binary encoding:

Step 1: Assume that the Turing machine T consists of $\{0, 1, B\}$ as the tape alphabets.
Step 2: Rename all states in T as $q_1, \ldots, q_i$.
Step 3: Rename all tape alphabets as $x_1, \ldots, x_j$.
Step 4: Rename all tape directions as D_1, D_2 and D_3, indicating left, right and no moves.
Step 5: Redefine, each transition rule in T as

$$\delta(q_j, x_L) = (q_x, x_y, D_z)$$
$$j, x \in \{1, 2, \ldots, i\}$$
$$L, y \in \{1, 2, \ldots, j\}.$$
$$z \in \{1, 2, 3\}.$$

Step 6: Encode the rules of T as $O^j|O^L|O^x|O^y|O^z$.

EXAMPLE 14.4.1: Consider a TM to accept $r = aba^*$, whose transitions are shown in figure 14.9:

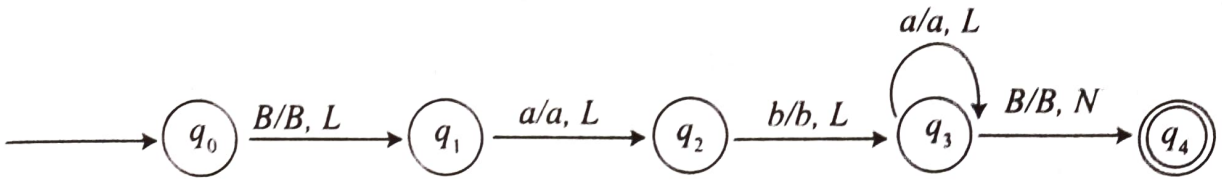

Figure 14.9. *Transition Diagram for TM to Accept $r = aba^*$*

The encoding of the above TM, using binary encoding, is as follows:

Step 1: Let the tape alphabet be $\{0, 1, B\}$, instead of $\{a, b, B\}$.
Step 2: $\{q_0, q_1, q_2, q_3, q_5\} = \{q_1, q_2, q_3, q_4, q_5, q_6\}$.
Step 3: $\{0, 1, B\} = \{x_1, x_2, x_3\}$.
Step 4: $\{L, R, N\} = \{D_1, D_2, D_3\}$.
Step 5: Redefine each transition rule in T and encode.

a. $\delta(q_0, B) = (q_1, B, L)$.
$\Rightarrow \delta(q_1, x_3) = (q_2, x_3, D_1)$.

For q_1, $j = 1$, for x_3, $L = 3$, for q_2, $x = 2$,
for x_3, $y = 3$ and for D_1, $z = 1$.

Thus, using unary number conventions, we have

$$0^j 10^L 10^x 10^y 10^z$$
$$\Rightarrow 0^1 10^3 10^2 10^3 10^1$$
$$\Rightarrow 01000100100010.$$

b. $\delta(q_1, a) = (q_2, a, L)$.

$$\Rightarrow \delta(q_1, 0) = (q_2, 0, L).$$
$$\Rightarrow \delta(q_2, x_1) = (q_3, x_1, D_1)$$
$$\Rightarrow 001010001010.$$

c. $\delta(q_2, b) = (q_3, b, L)$.

$$\Rightarrow \delta(q_2, 1) = (q_3, 1, L).$$
$$\Rightarrow \delta(q_3, x_2) = (q_4, x_2, D_1)$$
$$\Rightarrow 0001001000010010$$

d. $\delta(q_3, a) = (q_3, a, L)$.

$\Rightarrow \delta(q_3, 0) = (q_3, 0, L)$.

$\Rightarrow \delta(q_4, x_1) = (q_4, x_1, D_1)$.

$\Rightarrow 000010100001010$

e. $\delta(q_3, B) = (q_4, B, N)$.

$\Rightarrow \delta(q_4, x_3) = (q_5, x_3, D_3)$.

$\Rightarrow 00001000100000010001000$

Thus, the encoded transitions of TM are:

01000100100010
001010001010
0001001000010010
000010100001010
00001000100000010001000.

14.5 TM vs. Real Machines

a. A TM can compute anything a real computer computes.

For example: A TM can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms.

b. A TM has the ability to manipulate an unbounded amount of data. However, given a finite amount of time, a TM (like a real machine) can only manipulate a finite amount of data.

c. Like a TM, a real machine can have its storage space enlarged if required, by acquiring more disks or other storage media. If the supply of these runs short, the TM may become less useful as a model. But the fact is that neither a TM nor a real machine needs astronomical amounts of storage space in order to perform useful computations. The processing time required usually poses a bigger challenge.

d. TMs describe algorithms, independent of how much memory they utilise. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. TM allows us to make statements about algorithms, which will hold forever, regardless of advances in conventional computing machine architecture.

e. Real machines are much more complex than a TM.

For example, a TM describing an algorithm may have a few hundred states, while the equivalent DFA on a given real machine has quadrillions of them. This makes the analysis of DFA representation infeasible.

f. One way in which TMs are poor models for programs is that many real programs, such as operating systems and word processors, are written to receive unbounded input over time and, therefore, do not halt. In other words, TMs do not model such ongoing computation well. However, they can still model portions of it, such as individual procedures.

14.5.1 *The Power of TMs*

TMs represent more than an incremental increase in power, over a pushdown automaton. A TM can recognise a larger set of languages than any of the machines we have studied so far. It can recognise regular and context-free languages and a lot more. Once it recognises a language pattern, it can perform some action associated with the pattern. This is the essence of a computer's actions.

14.6 TM Languages

A TM halts if the current state is q, current tape symbol is 'a' and $\delta(q, a)$ is undefined.

We can make sure that a TM always halts, when it enters a final state, by removing all exit transitions. A TM rejecting the string w means that it does not accept w. A TM can reject by running forever, without entering a final state.

This creates an interesting and important problem that it cannot be ascertained whether the TM has rejected the input or just has not accepted it yet. Thus for a Turing machine, there are two types of languages:

- Turing acceptable Language or Recursively Enumerable
- Turing Decidable Language or Recursive Language

Before we discuss these languages, we first define the following:

- *Decider* - A machine that always halts is called a decider. As it always halts, the machine is able to decide whether a given string is a member of a formal language, hence the name.
- *Decision problem* - A decision problem is any arbitrary yes - or - no question on an infinite set of inputs.

For example, the problem 'given two numbers x and y, does x evenly divide y?' is a decision problem. The answer can be either 'yes' or 'no' and depends upon the values of x and y.

- *Decision procedure* - Methods used to solve decision problems are called decision procedures or algorithms.
  For example, for the problem 'given two numbers x and y, does x evenly divide y?', the algorithm would explain how to determine whether x evenly divides y, given x and y.
- *Decidable* - A decision problem, which can be solved by some algorithm, is called decidable.
- *Decidable problem/algorithmically solvable/recursively solvable* - A decision problem that can be solved by an algorithm and halts on all inputs in a finite number of steps, is called a decidable problem.
- *Decidable language* - A language for which membership can be decided by an algorithm, that halts on all inputs in a finite number of steps - equivalently, it can be recognised by a TM that halts for all inputs.
- *Undecidable* - A problem, that cannot be solved for all cases by an algorithm, is called an undecidable problem.
- *Undecidable language* - A language for which the membership cannot be decided by an algorithm - equivalently, it cannot be recognised by a TM that halts for all inputs.
- *Solvable* - A computational problem that can be solved by a TM is called a solvable problem. The problem may have a non-binary output.
- *Computable* - A function that can be computed by an algorithm is called computable (equivalently, it may be computable by a TM).
- *Tractable* - A problem which has an algorithm, that computes all instances of it in polynomial time, is called a tractable problem.
- *Intractable* - A problem, for which no algorithm exists which computes all instances of it in polynomial time, is intractable.

14.6.1 *Turing Acceptable/Recursively Enumerable Language*

Definition: A language is said to be *recursively enumerable* if there exists a TM that will halt and accept, when presented with any string $w \in \Sigma^*$ in the language as input, but may either halt and reject or loop forever, when presented with a string not in the language.

Thus, for Turing acceptable language L, there may be a Turing mahicne TM which halts on large number of input strings w, where $w \in L$, but there must be atleast one string $w \notin L$ on which TM does not halt.

For the recursively enumerable language, a decision problem is called partially semi-decidable, if there is a recursively enumerable set.

EXAMPLE 14.6.1: Consider a TM that will search for and locate a symbol A on its tape, if there is one, and then halt.

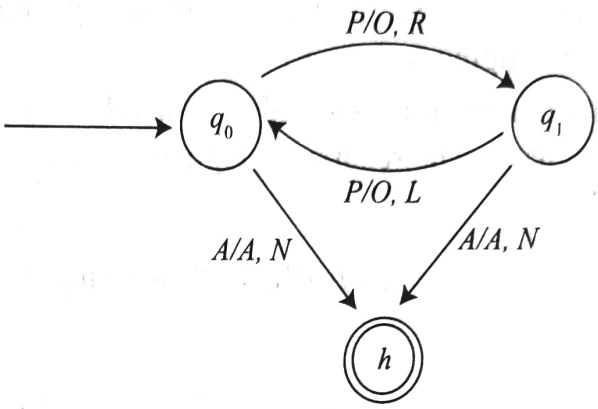

Figure 14.10. *A TM to Search for Symbol A on its Tape. P is Any Symbol Apart from A*

- For the input string $w = PPAP$, the TM operates from left to right searching for 'A'. When it finds 'A', it halts.
- For the input string $w = PPP$, the TM will go on forever looking for A i.e., it will not halt. This is known as 'semidecidability'. It can find an answer, if there is one, but does not know when to stop if there isn't any.

EXAMPLE 14.6.2: Consider a TM that accepts the language aa^*, over $\Sigma = \{a, b\}$.

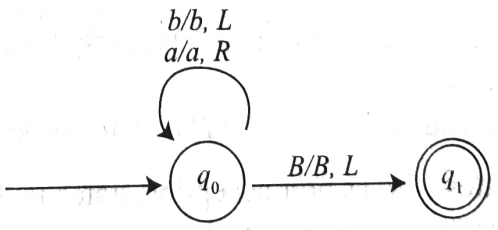

Figure 14.11. *A TM to Accept aa^**

- For the input string $w = aaa$, the TM halts and hence $w \in L$.
- For the input string $w = aba$, the TM loops forever hence $w \notin L$.

14.6.2 *Turing Decidable/Recursive Language*

Definition: A language is said to be recursive, if there exists a TM which will halt and accept when presented with any input string $W \in \Sigma^*$, only if the string is in the language, otherwise will halt and reject the string.

Thus, for Turing decidable language L, there is a Turing machine TM which halts for a large number of inputs W belonging to L.

A TM that always halts is known as a **decider** or a **total turing machine** and is said to decide the recursive language. The recursive language is also called as **recursive set** or **decidable**.

EXAMPLE 14.6.3: The language $L = \{a^n b^n c^n | n \geq 0\}$ is turing decidable.

Note-2:

- All recursive languages are also recursively enumerable.
- There may be languages which are recursively enumerable but not recursive.
- Set of all possible words, over the alphabet of the recursive language, is a recursive set.
- Set of all possible words, over the alphabet of the recursive enumerable language, is a recursively enumerable set.

14.6.3 *Properties of Recursive and Recursively Enumerable Languages*

Theorem I: Recursive languages are closed under complementation.

Proof. If a language L is recursive, there is a TM, T that accepts it and always halts. Thus, $L = L(T)$.
Now we construct a TM, T', such that $L' = L(T')$, as follows:

- Make all accepting states of T as non-accepting states in T'.
- Add a new accepting state in T' as h'.
- Change each non-accepting state of T which is undefined, to go to a new final state h'.

Thus, if T enters a final state on input W, then T' halts without accepting. If T halts without accepting, T' enters a final state.

$$\Rightarrow L' = L(T')$$

Theorem II: If L and L' are both recursively enumerable, then L and L' are recursive.

Proof. Let T_1 and T_2 be two turing machines, accepting languages L and L' respectively. Now construct a TM, T to simulate T_1 and T_2 in parallel for the input w.

- T accepts w, if T_1 accepts w and halts.
- T rejects w, if T_2 accepts w.

Thus $L = L(T)$ is recursive and since the complement of a recursive language is recursive, L' is also recursive.

Theorem III: Union of two recursive languages is recursive.

Proof. Let T_1 and T_2 be two Turing machines, accepting languages L_1 and L_2 respectively. Now construct a TM, T to simulate T_1 and T_2.

- If T_1 rejects, the TM, T simulates T_2 and accepts, iff T_2 accepts and finally halts.

Since T always halts, T accepts $L_1 \cup L_2$.

Theorem IV: Recursively enumerable languages are closed under union.

Proof. Let T_1 and T_2 be two Turing machines, accepting languages L_1 and L_2 respectively. Now construct a TM, T to simulate T_1 and T_2 in parallel.

- T accepts W, iff either T_1 or T_2 accepts W and halts. Thus, T accepts $L_1 \cup L_2$.

We discuss recursive and recursively enumerable language further, in chapter 15.

14.7 Undecidable Problem

An undecidable problem is a problem whose language is not a recursive one. Such problems cannot be solved by computers, in general. The following is the list of undecidable problems:

- **The halting problem**

  It is a decision problem which can be informally stated as:

  'Given a description of a program and a finite input, decide whether the program finishes running or will run forever'.

  Formally, it is stated as

'Given an arbitrary machine M and a string W, does M halt with W as the input string?'

The halting problem is undecidable over TM.

- **Rice's theorem**
  It states that all non-trivial properties of computer programs are undecidable.
- Determine, if a context-free grammar generates all possible strings or whether it is ambiguous.
- Given two context-free grammars, determine whether they generate the same set of strings or whether one generates a subset of strings generated by the other or whether there is any string at all that is generated by both.
- The post correspondence problem.
- The word problem for groups.
- The word problem for certain formal languages.
- Determine whether two finite simplicial complexes are homeomorphic.
- Determine whether the fundamental group of finite simplicial complex is trivial.

14.8 P and NP Classes of Languages

A class of languages is a set of languages that share some characteristic features. Since a language is a set of strings from a finite alphabet, a class of languages is a set of sets.

The language class P is the set of languages for which there exists a deterministic TM that accepts each language in a number of transitions, bounded by a fixed polynomial in the length of the input string.

The language class NP is the set of languages for which there exists a nondeterministic finite state control and which have a TM that does not have to reject a string in any prescribed number of moves.

14.8.1 *NP-Completeness*

A decision problem C is **NP-complete**, if

a. It is in NP

b. It is NP-hard i.e., every other problem in NP is reducible to it.

By 'reducible' it means that, for every problem L, there is a polynomial-time many-one reduction, a deterministic algorithm which transforms instances l and L into instances c and L, such that the answer to c is YES, iff the answer to l is YES.

The list below shows some well-known problems that are NP-complete, when expressed as decision problems:

- Boolean satisfiability problem
- N-puzzle
- Knapsack problem
- Hamiltonian cycle problem
- Travelling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Graph colouring problem

14.9 Church Turing Thesis

The thesis was first proposed by *Stephen C. Kleene* in 1943, but named after *Alonzo Church and Alan Turing*.

It can be stated as:

'Every 'function which would naturally be regarded as computable, can be computed by a Turing machine'.

Due to the vagueness of the concept of a 'function which naturally be regarded as computable', the thesis cannot be proven formally. Disproof would be possible only if humanity found ways of building hypercomputers whose results should 'naturally be regarded as computable'.

Any non-interactive computer program can be translated into a Turing machine, and any Turing machine can be translated into any *Turing complete programming language*. So the thesis is equivalent to saying that any Turing-complete programming language is sufficient to express any algorithm.

Variations of the thesis exist – for example, the **Physical Church-Turing thesis (PCTT)** states:

'Every function that can be physically computed can be computed by a Turing machine'.

Another variation is the **Strong Church-Turing Thesis (SCTT)**, which is not due to Church or Turing, but was rather realised gradually in the development of complexity theory. It states:

'Any 'reasonable' model of computation can be efficiently simulated on a probabilistic Turing machine.

The word 'efficiently' here means upto polynomial-time reductions. The Strong Church-Turing Thesis, then, posits that all 'reasonable' models of computation, yield the same class of problems that can be computed in polynomial time. Assuming the conjecture that probabilistic polynomial time (BPP) equals deterministic polynomial time (P), the word 'probabilistic' is optional in the Strong Church-Turing Thesis.

14.10 Exercises

1. Explain the follwing variations of TM:
   a. Multi-head
   b. K-dimensional
   c. offline
2. Present the formal definition of a non-deterministic TM.
3. Construct a non-deterministic TM to accept the language $\{0^n 1^m : n \geq 1, m \geq 1\}$.
4. Present the formal definition of a two-way infinite tape TM.
5. Design a two-tape TM to convert the input $\#w\#$ into $\#ww^R\#$.
6. Explain the following types of TM.
   a. Universal
   b. Random Access
   c. Oracle
7. Define linear bounded automaton. Present its formal definition.
8. Given the general procedure to encode a TM.
9. Encode a TM that accepts the language of all strings, which contain *aba* as a substring.
10. Compare TM with real machines.
11. Define recursively enumerable and recursive languages, with examples.
12. Prove that recursive languages are closed under complementation.