CHAPTER 1

Introduction

An automation is mathematical model for a finite state machine (FSM). A FSM is a machine that has a set of input symbols and transitions and jumps through a series of states according to a transition function. Automata play a major role in compiler design and parsing. Turing Machines are the most powerful computational machines. They possess an infinite memory in the form of a tape, and a head which can read and change the tape, and move in either direction along the tape or remain stationary. Turing Machines are equivalent to algorithms and are the theoretical basis for modern computers.

JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. JFLAP is extremely useful in constructing Turing Machine with multiple inputs. Complex Turing Machines can also be built by using other Turing Machines as components or building blocks for the same. The implementation of a Turing Machine and Universal Turing Machine for the JFLAP platform has been described. JFLAP is most successful and widely used tool for visualizing and simulating automata such as finite state machines, pushdown automata, and Turing Machines. By executing our Universal Turing Machine in JFLAP, everyone get a direct and interactive experience of how this Turing Machine is capable of emulating other Turing Machines

1.1 Objective

The objective of this project is designing a Turing Machine and Universal Turing Machine for Recursively Enumerable Language and Unrestricted Grammar for JFLAP platform. The Turing Machine differs from all other automata as it can work with Recursively Enumerable Languages and Unrestricted Grammar. If any language is Recursively Enumerable Language then it cannot be implemented using Finite Automata or a PDA. Turing Machines are the most powerful computational machines. The Turing Machines provide an abstract model to all the problems.

1.2 Overview OF JFLAP

JFLAP (Java Formal Languages and Automata Package) is instructional software for experimenting with automata and grammars, but goes further in allowing one to experiment with proofs and applications related to these topics. JFLAP's main feature is the ability to experiment with theoretical machines and grammars. With JFLAP one can build and run user-defined input on finite automata, pushdown automata, multi-tape Turing machines, regular grammars, context-free grammars (CFG), unrestricted grammars, and L-systems. After constructing the automaton or grammar, one can trace through a single input string or receive automatic feedback on multiple inputs.

JFLAP's second feature is the ability to construct in steps the proof of the transformation of one form to another form.

For example, after constructing nondeterministic finite automata (NFA), one can step through its conversion to deterministic finite automata (DFA), then to a minimal state DFA, and then to regular grammar. As another example, one can build a CFG and construct in steps the equivalent nondeterministic PDA.

JFLAP's third feature is the experimentation with applications of theoretical material. One example is experimenting with parsing by constructing the SLR (1) parse table in steps, and then stepping through the parsing of input strings and the construction of the equivalent parse tree. The construction of the parse table includes building a special DFA that models the parsing process, thus seeing a use for a DFA. Another application is building an L-system grammar of a plant, and rendering it to watch a simulation of the plant growing.

1.3 TURING MACHINE IN JFLAP

Among all the machines invented by the ingenuity of humankind, the computer stands alone as being a general-purpose device. Unlike the steam engine or the telephone, the computer is a **programmable** device, capable of being used for a vast number of different purposes. Alan Turing was one of the first to recognize this fundamental property when he defined the **Turing Machine**. According to the classic Church-Turing thesis, the Turing Machine captures the essence of “computation”. This thesis states that any problem that is computable can be carried out by a Turing Machine. The Church-Turing thesis is to Computer Science what Darwin’s theory of evolution is to Biology.

Any particular Turing Machine, denoted here as T_M, consists of a set of rules, or instructions. The execution of T_M involves applying those rules to the given input string. Thus a special-purpose Turing Machine can be written that will add two binary numbers, or another Turing Machine that will move all a’s in front of all b’s, etc. But the uniquely programmable nature of computers is not that one can write a set of rules to accomplish one particular task, but that one can write rules that can carry out any other set of rules. Thus the “programmability” of the Turing Machine can be seen by the existence of one particular Turing Machine, T_U, known as the universal Turing Machine. T_U reads, as input, the description of arbitrary Turing Machine T_M, together with the input string of T_M, and then T_U performs the same computation that T_M would have done. T_U effectively emulates T_M.

In our work, such a Turing Machine is implemented. This machine was developed for the **JFLAP** environment. The JFLAP software provides a GUI environment where users can design and simulate different types of automata. JFLAP has been downloaded over 35,000 times by people from more than 150 countries.

.

1.3.1 Turing Machine

A Turing machine is an automation whose temporary storage is tape. This tape is divided into cells, each of which is capable of holding one symbol. Associated with the tape is read-write head that can travel right or left on the tape and that can read and write a single symbol on each move. Turing Machines are the most powerful computational machines. The Turing Machine (TM) is the solution for the halting problem and all other problems that exist in the domain of computer science. Turing Machines provide an abstract model to all problems. It can work with Recursively Enumerable Language and Unrestricted Grammar.

In the beginning, a brief review of Turing Machines is mentioned. Figure 1 shows a small Turing Machine built using the JFLAP environment. This machine will serve as an example to illustrate how to use our universal TM. The function of this TM is very simple. It performs the “two’s complement” function on the input string, where each ‘a’ is considered as a binary zero, and each ‘b’ is considered a binary one.

JFLAP represents a Turing Machine as a directed graph. Any TM consists of a set of states, each of which is a node in the graph. Each state is labeled with its state id number. The initial state, designated by a triangle, is where execution begins. The TM also has a set of final states, denoted by double-circles.

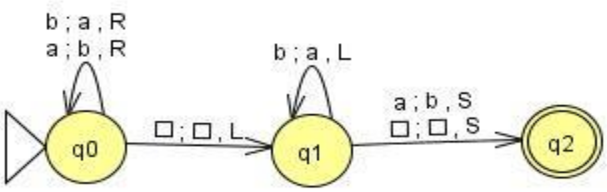


Figure: 1 Sample Turing Machine

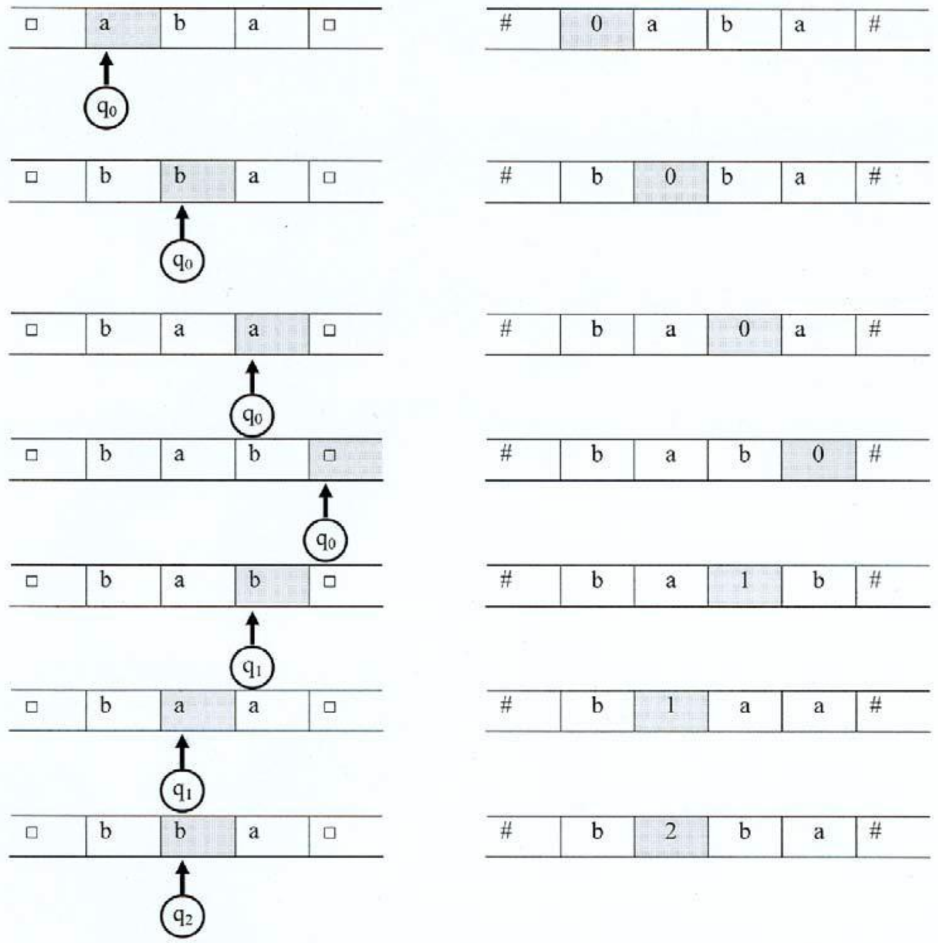
A Turing Machine contains a memory tape that extends infinitely in both directions. Each cell of the memory contains one symbol from the tape alphabet. In the example above, the alphabet consists of the characters {a, b, G}. JFLAP uses the G symbol to represent a blank character. Initially the tape contains the input string, denoted as w, surrounded by blank characters. The input string w may not have any embedded blanks.

When the machine is executing, a “read-head” moves along the tape, indicating the next character to be read. When execution begins the read-head is aligned at the leftmost character of the input.

The execution of any Turing Machine is specified by its **transitions**, or delta rules. Each transition is of the form:

□ (current state, current read symbol) □ ☎next state, write symbol, direction)

Each transition is represented as an edge in the directed graph presentation of the Turing Machine. The edge is directed from the current state to the next state, and is labeled with the read symbol, the write symbol and the direction, respectively. The direction must be L, R, or S, representing left, right, and stationary. The sample Turing Machine of Figure 1 has 6 transitions.



[2a: execution trace of T_M] [2b: partial view of corresponding execution trace of T_U]

[Figure 2]

When a transition is being executed, the Turing Machine will read the character from the tape located at the position marked by the read-head. If the character matches that required by the transition, the character on the tape will be replaced by the corresponding write symbol, and the machine will shift the read-head position one cell to the left, one cell to the right, or no shift will occur at all, as specified by the direction entry of that delta-rule. If the Turing Machine ever enters a final state, then it can be said that the TM “accepts” the input. If at any time the TM is in a configuration where there is no applicable transition, then it can be said that the TM has “crashed” and the input is “rejected”. The third possibility is that the TM might continue its execution indefinitely, endlessly looping.

The left column of Figure 2 presents the execution trace of the sample T_M on the input string w = “aba”. Each entry in the column indicates the current state and the position of the read-head of T_M. Execution continues until T_M enters a final state.

1.4 Recursively Enumerable Language

A language L is said to be recursively enumerable if there exists a Turing Machine that accepts it. It implies that there exists a Turing Machine M, such that, for every w € L

$$q_0w \vdash^* M \ x_1q_f x_2$$

with q_f a final state. The definition says nothing about what happens for w not in L; it may be that machine halts in a non final state or that it never halts and goes into an infinite loop.

Regular languages form a proper subset of Context Free Languages. So PDA is more powerful than finite automata. But CFLs are limited in scope because many of the simple language like $a^nb^nc^n$ are not context free. So to incorporate the set of all languages that are not accepted by PDAs and hence that are not context free, more powerful language families has been formed. This creates the class of Recursively Enumerable Languages (REL).

1.5 Unrestricted Grammars

A grammar G = (V,T,S,P) is called unrestricted if all the productions are of form

$$u \to v,$$

Where u is in $(V \text{ U } T)^+$ and v is in $(V \text{ U } T)^*$.

In an unrestricted grammar, essentially no conditions are imposed on the productions. Any number of variables and terminals can be on left or right, and these can occur in any order. There is only one restriction: λ (empty string) is not allowed as the left side of a production. Unrestricted grammars are much more powerful than restricted forms like the regular and context free grammars.

In fact, unrestricted grammars correspond to the largest family of languages, so there is a hope to recognize by mechanical means; that is, unrestricted grammars generate exactly the family of recursively enumerable languages.

The Turing Machine and its other models, Formal Language (Recursively Enumerable Language and Unrestricted Grammar) are described in Chapter 2, Chapter 3, and Chapter 4 respectively.

CHAPTER 2

Turing Machine

A **Turing machine** is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The "Turing" machine was invented in 1936 by Alan Turing who called it an "a-machine" (automatic machine). The Turing machine is not intended as practical computing technology, but rather as a hypothetical device representing a computing machine. Turing machines help computer scientists understand the limits of mechanical computation.

Turing gave a succinct definition of the experiment in his 1948 essay, "Intelligent Machinery". Referring to his 1936 publication, Turing wrote that the Turing machine, here called a Logical Computing Machine, consisted of: ...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.

2.1 Definition of a Turing Machine

A Turing machine is an automation whose temporary storage is tape. This tape is divided into cells, each of which is capable of holding one symbol. Associated with the tape is read – write head that can travel right or left on the tape and that can read and write a single symbol on each move.

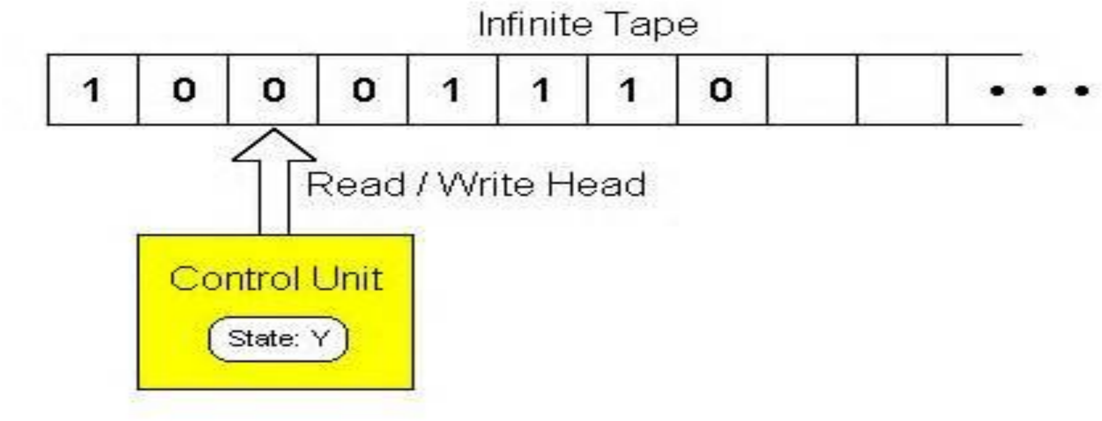


Figure : 3 Turing Machine

A Turing Machine M is defined by

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

Where

Q is the set of internal states $\{q_i \mid i$ is a nonnegative integer$\}$
Σ is the input alphabet
Γ is the finite set of symbols in the tape alphabet
δ is the transition function
$\square$ is the blank symbol.
q_0 (is member of Q) is the initial state
F (is a subset of Q) is the set of final states

In the definition of a Turing Machine, it is assumed that Σ is the subset of $\Gamma - \{\square\}$, that is, that the input alphabet is a subset of the tape alphabet, not including the blank. Blanks are ruled out as input for reasons that will become apparent shortly.

Since one can make several different definitions of a Turing machine, it is worthwhile to summarize the main features of our model, which will be called a standard Turing machine:

1. The Turing machine has a tape that is unbounded in both directions, allowing any number of left and right moves.
2. The Turing machines is deterministic in the sense that δ defines at most one move for each configuration.
3. There is no special input file. It is assumed that at the initial time the tape has some specified content. Some of this may be considered input. Similarly, there is no special output device. Whenever the machine halts, some or all of the contents of the tape may be viewed as output.

2.2 Turing Machines as Language Accepters

Turing Machines can be viewed as accepters in the following sense. A string w is written on the tape, with blanks filling out the unused portions. The machine is started in the initial state q_0 with the read – write head positioned on the leftmost symbol of w. If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

2.2.1 Definition

Let M = (Q, Σ, Γ, δ, q_0, $\square$, F) be a Turing machine. Then the language accepted by M is

$$L(M) = \left\{ w \in \sum^{+} : q_0 w \vdash^{*} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^{*} \right\}$$

This definition indicates that the input w is written on the tape with blanks on either side. The reason for excluding blanks from the input now becomes clear: it assures us that all the input is restricted to a well – defined region of the tape, bracketed by blanks on the right and left. Without this convention, the machine could not limit the region in which it must look for the input; no matter how many blanks it saw, it could never be sure that there was not some nonblank input somewhere else on the tape.

This definition tells us what must happen when w € L(M). It says nothing about the outcome for any other input. When w is not in L(M), one of two things can happen: the machine can halt in a non final state or it can enter an infinite loop and never halt. Any string for which M does not halt is by definition not in L(M).

For ∑ = {a,b}, design a Turing machine that accepts

$$L = \{ a^n b^n : n \geq 1\}.$$

Intuitively, the problem is solved in the following fashion. Starting at the leftmost a, it is checked off by replacing it with some symbol, say x. The read-write head is made to travel right to find the leftmost which in turn is checked off by replacing it with another symbol, say y. After that, it went left again to the leftmost a, replace it with an x, then move to the leftmost b and replaces it with y, and so on. Traveling back and forth this way, each a with a corresponding b is matched. If after some time no a's or b's remain, then the string must be in L.

Working out the details, it is arrived at a complete solution for which

$$Q = \{q_0,q_1,q_2,q_3,q_4\},$$
$$F = \{q_4\},$$
$$\sum = \{a, b\},$$
$$\Gamma = \{a, b, x, y, \square\} .$$

The transitions can be broken into several parts. The set

$$\delta\ (q_0, a) = (q_1, x, R),$$
$$\delta\ (q_1, a) = (q_1, a, R),$$
$$\delta\ (q_1, y\ ) = (q_1, y\ , R),$$
$$\delta\ (q_1, b\ ) = (q_2, y\ , L),$$

replaces the leftmost a with an x, then causes the read-write head to travel right to the first b, replacing it with a y. When the y is written, the machine enters a state q_2, indicating that an a has been successfully paired with a b.

The next set of transitions reverses the direction until an x is encountered, repositions the read-write head over the leftmost a, and returns control to the initial state.

$$\delta\ (q_2, y) = (q_2, y, L),$$
$$\delta\ (q_2, a) = (q_2, a, L),$$

$$\delta\ (q_2, x\ ) = (q_0, x\ , R).$$

It is now back in the initial state go, ready to deal with the next a and b. After one pass through this part of the computation, the machine will have carried out the partial computation

$$q_0aa \ldots.. abb \ldots b \vdash^* xq_0a \ldots ayb \ldots. b,$$

So that a single a has been matched with a single b. After two passes, the partial computation is completed

$$q_0aa \ldots.. abb \ldots b \vdash^* xxq0 \ldots ayy \ldots. b,$$

And so on, indicating that the matching process is being carried out properly.

When the input is a string a^nb^n, the rewriting continues this way, stopping only when there are no more a's to be erased. When looking for the leftmost a, the read-write head travels left with the machine in state q_2. When an x is encountered, the direction is reversed to get the a. But now, instead of finding an a it will find a y. To terminate, a final check is made to see if all a's and b's have been replaced (to detect input where an a follows a b). This can be done by

$$\delta\ (q_0, y) = (q_3, y, R),$$
$$\delta\ (q_3, y) = (q_3, y, R),$$
$$\delta\ (q_3, \square) = (q_4, \square\ , R).$$

If a string not in the language is input, the computation will halt in a nonfinal state. For example, if a string a^nb^m, with $n > m$, is applied to the machine, it will eventually encounter a blank in state q_1. It will halt because no transition is specified for this case. Other input not in the language will also lead to a nonfinal halting state.

The particular input $aabb$ gives the following successive instantaneous descriptions

$$q_0aabb \vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb$$
$$\vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb$$
$$\vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy$$

$$\vdash xxq_0yy \vdash xxyq_3y \vdash xxyq_3\square$$
$$\vdash xxyy\square\, q_4\square$$

At, this point the Turing machine halts in a final state, so the string $aabb$ is accepted.

2.3 Turing Machine as Transducers

Turing Machines are not only interesting as language accepters; they provide us with a simple abstract model for digital computers in general. Since the primary purpose of a computer is to transform into output, it acts as a transducer. To model computers using Turing machines, this aspect to be looked more closely.

The input for a computation will be all the nonblank symbols on the tape at the initial time. At the conclusion of the computation, the output will be whatever is then on the tape. Thus, one can view a Turing machine transducer M as an implementation of a function f defined by

$$\hat{w} = f(w),$$

provided that

$$q_0w \vdash^*_M q_f\hat{w},$$

for some final state q_f.

2.3.1 Definition

A function f with domain D is said to be Turing-computable or just computable if there exists some Turing machine M = (Q, Σ, Γ, δ, q_0, □, F) such that

$$q_0w \vdash^*_M q_f w, \qquad q_f € F,$$

for all w € D.

2.4 Limitation of Turing Machines

Some problems that cannot be solved by Turing machines. The argument that the power of mechanical computations is limited is not surprising. Intuitively it is known that many vague and speculative questions require special insight and reasoning well beyond the capacity of any computer that one can now construct or even plausibly foresee. What is more interesting to computer scientists is that there are questions that can be clearly and simply stated, with an apparent possibility of an algorithmic solution, but which are known to be unsolved by any computer.

2.4.1 Computability and Decidability

Our concern here will be the somewhat simplified setting where the result of a computation is a simple “yes” or “no”. In this case, a problem being decidable or undecidable is discussed. By a problem one can understand a set of related statements, each of which must be either true or false. For example, consider the statement, “For a context-free Grammar G, the language L (G) is ambiguous”. For some G, this is true, for others it is false, but clearly one must have one or the other. The problem is to decide whether the statement is true for any given G. Again, there is an underlying domain, the set of all context-free grammars. A problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

When decidability or undecidability results are stated, one must always know what the domain is, because this may affect the conclusion. The problem may be decidable on some domain but not on another. Specifically, a single instance of a problem is always decidable, since the answer is either true or false. In the first case, a Turing machine that always answers “true” gives the correct answer, while in the second case one that always answers “false” is appropriate. This may seem like a facetious answer, but it emphasizes an important point. The fact that one does not know, what the correct answer is, makes no difference, what matters is that there exists some Turing machine that does give the correct response.

2.4.2 The Turing Machine Halting Problem

Starting with some problems that have some historical significance and that at the same time gives a starting point for developing later results. The best-known of these is the Turing machine halting problem. Simply stated, the problem is: given the description of a Turing machine M and an input w, does M, when started in the initial configuration q_0w, perform a computation that eventually halts? Using an abbreviated way of talking about the problem, one may ask whether M applied to w, or simply (M, w), halts or does not halt. The domain of this problem is to be taken as the set of all Turing machines and all w; that is, a single Turing machine that give the description of an arbitrary M and w, will predict whether or not the computation of M applied to w will halt.

One cannot find the answer by simulating the action of M on w, say by performing it on a universal Turing machine, because there is no limit on the length of the computation. If M enters an infinite loop, then no matter how long to wait, one can never be sure that M is in fact in a loop. It may simply be a case of a very long computation. An algorithm that can determine the correct answer for any M and w by performing some analysis on the machine’s description and the input is needed. But as it is shown, no such algorithm exists.

For subsequent discussion, it is convenient to have a precise idea of what is meant by the halting problem; for this reason, one make a specific definition of Turing machine’s halting problem.

2.4.2.1 Definition

Let w_M be a string that describes a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, and let w be a string in M’s alphabet. It is assumed that w_M and w are encoded as a string of 0’s and 1’s. A solution of the halting problem is a Turing machine H, which for any w_M and w performs the computation

$$q_0 w_M w \vdash^* x_1 q_y x_2,$$

if M applied to w halts, and

$q_0 w_M w \vdash^* y_1 q_n y_2$,

If M applied to w does not halt. Here q_y and q_n are both final states of H.

CHAPTER 3

Other Models of Turing Machines

Our definition of standard Turing machine is not the only possible one; there are alternative definitions that could serve equally well. The conclusions one can draw about the power of a Turing machine are largely independent of the specific structure chosen for it. In this chapter several variations are looked at, showing that the standard Turing machine is equivalent to other more complicated models.

If Turing's thesis is accepted, one can expect that complicating the standard Turing machine by giving it a more complex storage device will not have any effect on the power of the automation. Any computation that can be performed on such a new arrangement will still fall under the category of mechanical computation and, therefore, can be done by a standard model. It is nevertheless instructive to study more complex models, if more no other reason that an explicit demonstration of the expected result will demonstrate the power of the Turing Machine and thereby increase our confidence in Turing's thesis. Many variations on the basic model of Definition 2.1 are possible.

3.1 Minor Variations on the Turing Machine Theme

Whenever a definition is changed, a new type of automata is introduced and the question is raised whether these new automata are in any real sense different from those already encountered. What is meant by an essential difference in their definitions, these differences may not have any interesting consequences. There is example of this in the case of deterministic and nondeterministic finite automata. These have quite different definitions, but they are equivalent in the sense that they both are identified exactly with the family of regular languages.

3.1.1 Equivalence of classes of Automata

Whenever equivalence for two automata or classes of automata is defined, one must carefully state what is to be understood by this equivalence.

3.1.1.1 Definition

Two automata are equivalent if they accept the same language. Consider two classes of automata C_1 and C_2. If for every automation M_1 in C_1 there is an automation M_2 in C_2 such that

$$L\ (M_1) = L\ (M_2),$$

Say that C_2 is at least as powerful as C_1. If the converse also holds and for every M_2 in C_2 there is an M_1 in C_1 such that $L\ (M_1) = L\ (M_2)$, therefore, C_1 and C_2 are equivalent.

3.2 Turing Machines with a stay-option

In our definition of a standard Turing machine, the read-write head must move either to the right or to the left. Sometimes it is convenient to provide a third option, to have the read-write head stay in place after rewriting the cell content. Thus, one can define a Turing machine with stay-option by replacing δ in definition by

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

With the interpretation that S signifies no movement of the read-write head. This option does not extend the power of the automation.

3.3 Turing Machines with Semi – Infinite Tape

Turing Machine is used with a tape that is unbounded only in one direction. One can visualize this as a tape that has a left boundary (Figure 4). This Turing Machine is otherwise identical to our standard model, except that no left move is permitted when the read-write head is at the boundary.

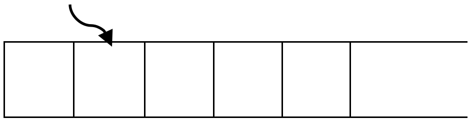


Figure: 4

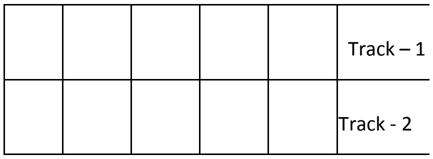


Figure: 5

The simulating machine $\hat{M}$ has a tape with two tracks. On the upper one, keep the information to the right of some reference point on M's tape. The reference point could be, for example, the position of the read- write head at the start of the computation. The lower track contains the left part of M's tape in reverse order. $\bar{M}$ is programmed so that it will use information on the upper track only as long as M's read-write head is to the right of the reference point, and work on the lower track as M moves into the left part of its tape. The distinction can be made by partitioning the state set of $\bar{M}$ into two parts, say Qu and $Ql,:$ the first to be used when working on the upper track, the second, to be used on the lower one. Special end markers # is put on the left boundary of the tape to facilitate switching from one track to the other. For example, assume that the machine to be simulated and the simulating machine are in the respective configurations shown in Figure 6 (a & b) and that the move to be simulated is generated by

δ (q_i, a) = (q_j,c,L).

The simulating machine will first move via the transition

$\hat{\delta}$ ($\hat{q}_i$, (a, b)) == ($\hat{q}_j$ (c,b),L),

Where $\hat{q}_i$ € Qu- Because $\hat{q}_i$ belongs to Qu-, only information in the upper track is considered at this point. Now, the simulating machine sees (#, #).

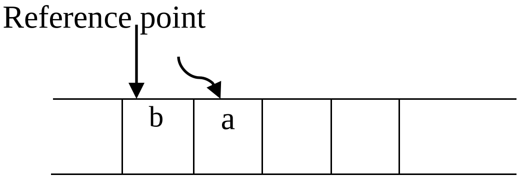


Figure: 6 (a) Machine to be simulated

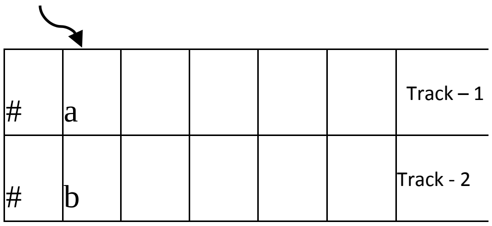


Figure: 6 (b) Simulating machine

3.4 The off- Line Turing Machine

By putting the input file back into the picture, one get what is known as an ***off-line Turing machine***. In such a machine, each move is governed by the internal state, what is currently read from the input file, and what is seen by the read-write head. A schematic representation of an off-line machine is shown in Figure 7. A formal definition of an off-line Turing machine is easily made here in this work. It is required to indicate why the class of off-line Turing machines is equivalent to the class of standard machines.

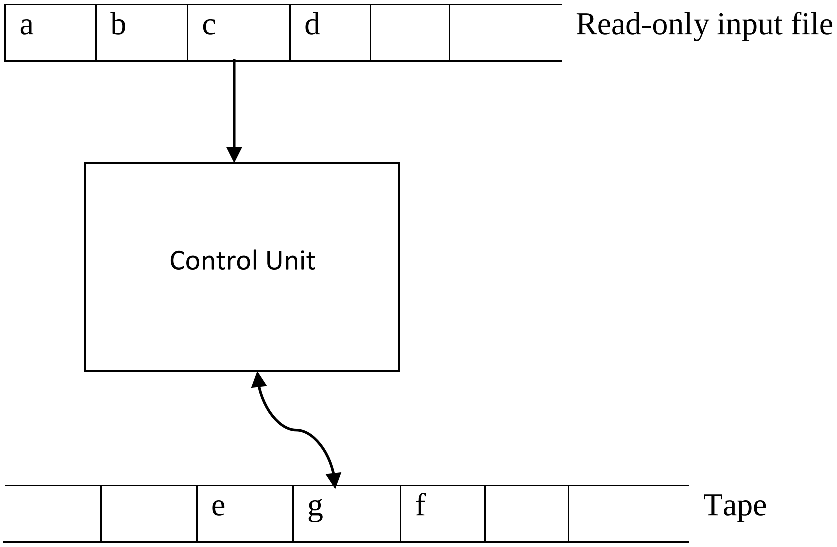
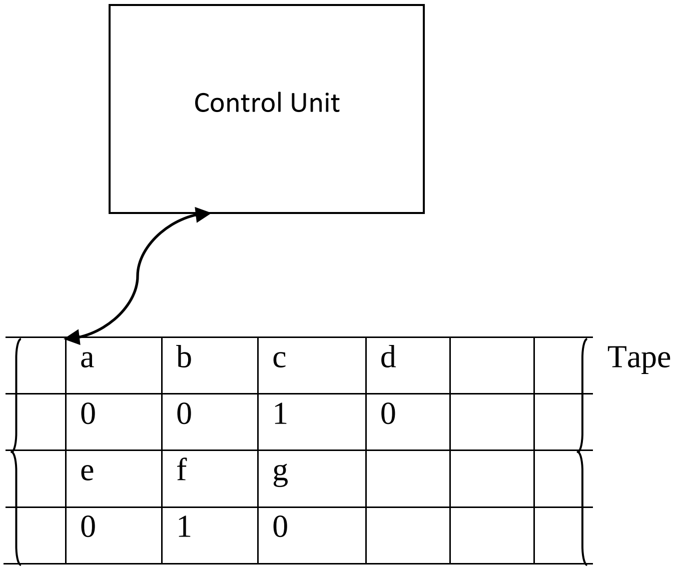


Figure: 7 The Off-Line Turing Machine

First, the behavior of any standard Turing machine can be simulated by some off-line model. All that needs to be done by the simulating machine is to copy the input from the input file to the tape. Then it can proceed in the same way as the standard machine.

The simulation of an off-line machine M by a standard machine $\hat{M}$ requires a lengthier description. A standard machine can simulate the computation of an off-line machine by using the four-track arrangement shown in Figure 8. In that picture, the tape contents shown represent the specific configuration of Figure 7. Each of the four tracks of $\hat{M}$ plays a specific role in the simulation. The first track has the input, the second marks the position at which the input is read, the third represents the tape of M, and the fourth shows the position of M's read-write head.



	a	b	c	d		
	0	0	1	0		
	e	f	g			
	0	1	0			

Figure: 8

The simulation of each move of M requires a number of moves of $\hat{M}$. Starting from some standard position, say the left end, and with the relevant information marked by special end markers, $\hat{M}$ searches track 2 to locate the position at which the input file of M is read. The symbol found in the corresponding cell on track 1 is remembered by putting the control unit of $\hat{M}$ into a state chosen for this purpose. Next, track 4 is searched for the position of the read-write head of M. With the remembered input and the symbol on track 3, it is now known that M is to do. This information is again renumbered by $\hat{M}$ with an appropriate internal state. Next, all four tracks of $\hat{M}$'s tape are modified to reflect the move of M. Finally, the read-write head of $\hat{M}$ returns to the standard position for the simulation of the next move.

3.5 Turing Machines with More Complex Storage

The storage device of a standard Turing machine is so simple that one might think it possible to gain power by using more complicated storage devices. But this is not the case as illustrated in these two examples.

3.5.1 Multitape Turing Machines

A multitape Turing machine is a Turing machine with several tapes, each with its own independently controlled read-write head.

The formal definition of a multitape Turing machine goes beyond Definition 2.1, since it requires a modified transition function. Typically, an n-tape machine is defined by $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ where Q, E, T, q_0, F are as in Definition 2.1,

But where

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

specifies what happens on all the tapes. For example, if $n = 2$, with a current configuration shown in Figure 9, then

$$\delta\ (q_0, a, e) = (q_1, x, y, L, R)$$

is interpreted as follows. The transition rule can be applied only if the machine is in state q_0 and the first read-write head sees an a and the second an e. The symbol on the first, tape will then be replaced with an x and its read-write head will move to the left. At the same time, the symbol on the second tape is rewritten as y and the read-write head moves right. The control unit then changes its state to qi and the machine goes into the new configuration shown in Figure 10.

To show the equivalence between multitape and standard Turing machines, it is argued that any given multitape Turing machine M can be simulated by a standard Turing machine $\hat{M}$ and,

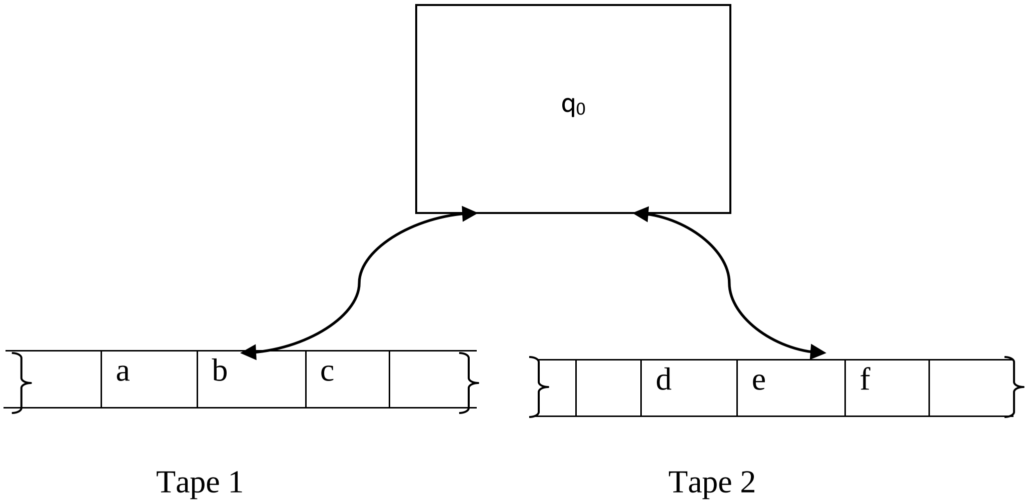


Figure : 9

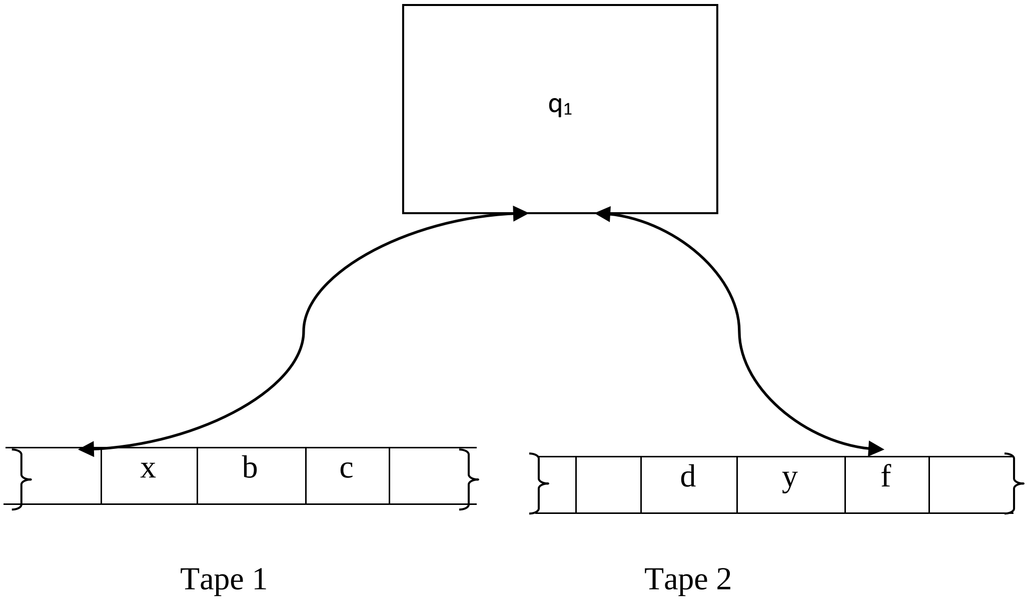


Figure: 10

conversely, that any standard Turing machine can be simulated by a multitape one. The second part of this claim needs no elaboration, since one can always elect to run a multitape machine with only one of its tapes doing useful work. The simulation of a multitape machine by one with a single tape is a little more complicated, but conceptually straightforward.

Consider, for example, the two-tape machine in the configuration depicted in Figure 11.

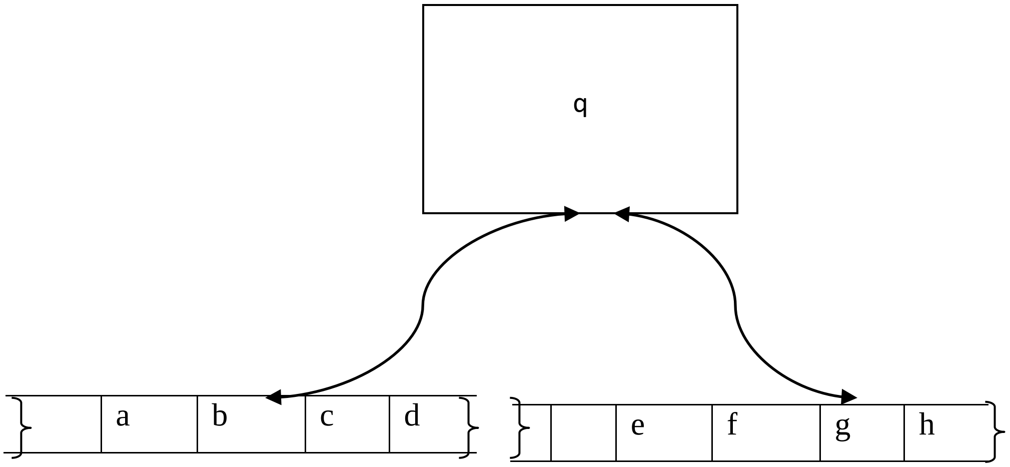
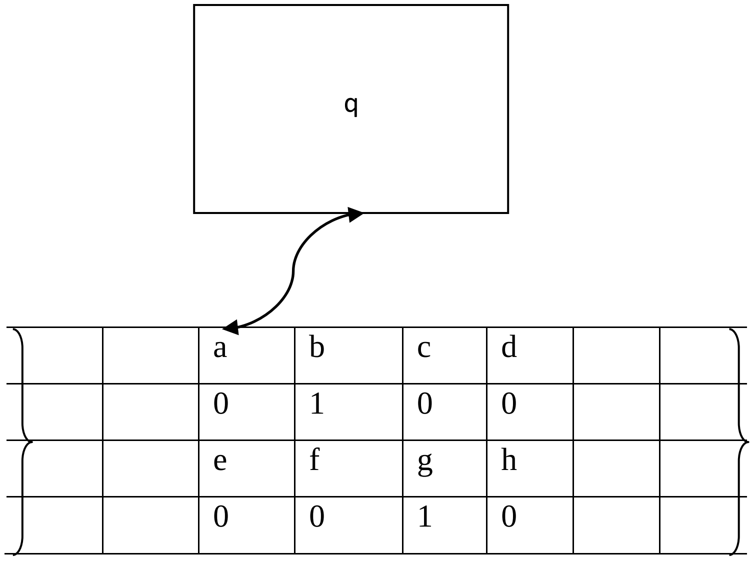


Figure: 11

The simulating single-tape machine will have four tracks (Figure 12). The first track represents the contents of tape 1 of M. The nonblank part of the second track has all zeros, except for a single 1 marking the position of M's read-write head. Tracks 3 and 4 play a similar role for tape 2 of M. Figure 12 makes it clear that, for the relevant configurations of $\hat{M}$ (that is, the ones that have the indicated form), there is a unique corresponding configuration of M.



		a	b	c	d		
		0	1	0	0		
		e	f	g	h		
		0	0	1	0		

Figure: 12

The representation of a multitape machine by a single-tape machine is similar to that used in the simulation of an off-line machine. The actual steps in the simulation are also much the same, the only difference being that there are more tapes to consider. The outline given for the simulation of offline machines carries over to this case with minor modifications and suggests a procedure by which the transition function $\hat{\delta}$ of $\hat{M}$ can be constructed from the transition function δ of M. While it is not difficult to make the construction precise, it takes a lot of writing. Certainly, the computations of $\hat{M}$ given the appearance of being lengthy and elaborate, but this has no bearing on the conclusion. Whatever can be done of *M* can also be done on $\hat{\mathrm{M}}$.

It is important to keep in mind the following point. When it is claimed that a Turing machine with multiple tapes is no more powerful than a standard one, a statement is being made only about what can be done by these machines, particularly, what languages can be accepted.

3.5.2 Multidimensional Turing Machines

A multidimensional Turing machine is one in which the tape can be viewed as extending infinitely in more than one dimension. A diagram of a two- dimensional Turing machine is shown in Figure 13.

The formal definition of a two-dimensional Turing machine involves a transition function δ of the form

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D),$$

Where *U* and *D* specify movement of the read-write head up and down, respectively.

To simulate this machine on a standard Turing machine, one can use the two-track model depicted in Figure 14. First, associate on ordering or address with the cells of the two-dimensional tape. This can be done in a number of ways, for example, in the two-dimensional fashion indicated in Figure 13. The two-track tape of the simulating machine will use one track to store cell contents and the other one to keep the associated address. In the scheme of Figure 13, the configuration in which cell (1, 2) contains *a* and cell (10, -3) contains *b* is shown in Figure 14. Note one complication: the cell address can involve arbitrarily large integers, so the address track cannot use a fixed-size

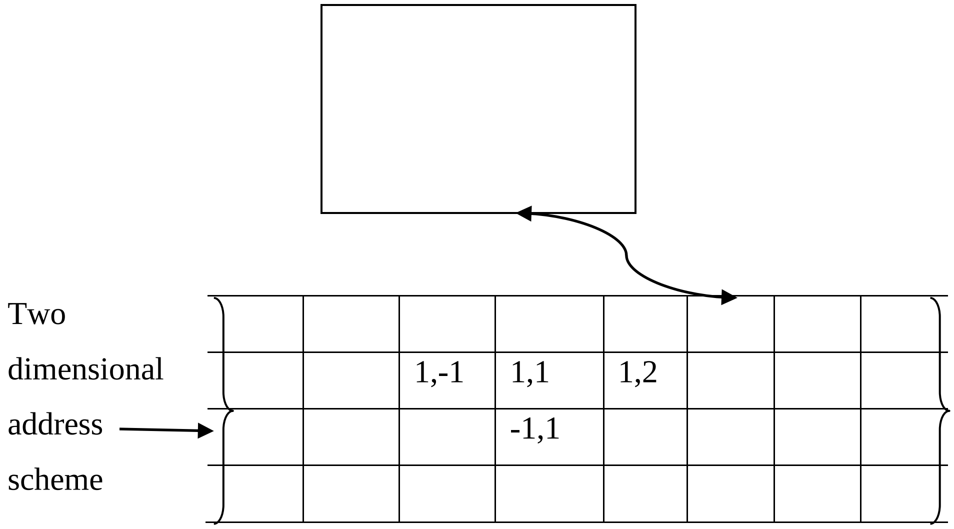


Figure: 13

field to store addresses. Instead, one must use a variable field-size arrangement, using some special symbols to delimit the fields, as shown in the picture.

	a				b						
	1	#	2	#	1	0	#	-	3	#	

Figure: 14

Let us assume that, at the start of the simulation of each move, the read- write head of the two-dimensional machine M and the read-write head of the simulating machine $\hat{M}$ are always on corresponding cells. To simulate a move, the simulating machine $\hat{M}$ first computes the address of the cell to which M is to move. Using the two-dimensional address scheme, this is a simple computation. Once the address is computed, $\hat{M}$ finds the cell with this address on track 2 and then changes the cell contents to account for the move of M. Again M, there is a straightforward construction for M.

3.6 Nondeterministic Turing Machines

While Turing's thesis makes it plausible that the specific tape structure is immaterial to the power of the Turing machine, the same cannot be said of nondeterminism. Since nondeterminism involves an element of choice and so has a no mechanistic flavor, an appeal to Turing's thesis is inappropriate. The effect of nondeterminism must be looked in more detail if one wants to argue that nondeterminism adds nothing to the power of a Turing machine. Again one resort to simulation, showing that nondeterministic behavior can be handled deterministically.

3.6.1 Definition

A nondeterministic Turing machine is an automaton as given by Definition 2.1, except, that δ is now a function

$$\delta: Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\}}$$

As always when nondeterminism is involved, the range of δ is a set of possible transitions, any of which can be chosen by the machine.

Since it is not clear what role nondeterminism plays in computing functions, nondeterministic automata are usually viewed as accepters. A non- deterministic Turing machine is said to accept w if there is any possible sequence of moves such that

$$q_0 w \vdash^* x_1 q_f x_2,$$

with qf ∈ F. A nondeterministic machine may have moves available that lead to a nonfinal state or to an infinite loop. But, as always with nondeterminism, these alternatives are irrelevant; the existence of some sequence of moves leading to acceptance is of great interest.

To show that a nondeterministic Turing machine is no more powerful than a deterministic one, one needs to provide a deterministic equivalent for the nondeterminism. Nondeterminism can be viewed as a deterministic backtracking algorithm, and a deterministic machine can simulate a nondeterministic one as long as it can handle the bookkeeping involved in the backtracking. To see how this can be done simply, let us consider an alternative view of nondeterminism, one which is useful in many arguments: a nondeterministic machine can be seen as one that has the ability to replicate itself whenever necessary. When more than one move is possible, the machine produces as many replicas as needed and gives each replica the task of carrying out one of the alternatives.

This view of nondeterminism may seem particularly no mechanistic. Unlimited replication is certainly not within the power of present-day computers. Nevertheless, the process can be simulated by a deterministic Turing machine. Consider a Turing machine with a two-dimensional tape (Figure 15). Each pair of horizontal tracks represents one machine; the top track containing the machine's tape, the bottom one for indicating its internal state and the position of the read-write head. Whenever a new machine is to be created, two new tracks are started with the appropriate information.

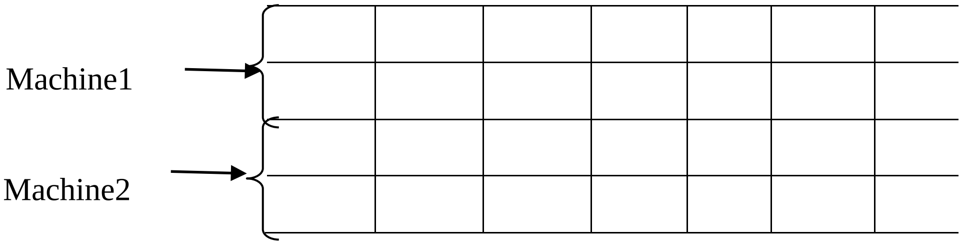


Figure: 15

3.7 A Universal Turing Machine

Consider the following argument against Turing's thesis: "A Turing machine as presented in Definition 9.1 is a special purpose computer. Once δ is defined, the machine is restricted to carrying out one particular type of computation. Digital computers, on the other hand, are general purpose machines that can be programmed to do different jobs at different times. Consequently, Turing machines cannot be considered equivalent to general purpose digital computers."

This objection can be overcome by designing a reprogrammable Turing machine, called a universal Turing machine. A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w, can simulate the computation of M on w. To construct such an M_u, first a standard way of describing Turing machines is selected. Without loss of generality, assume that

$$Q = \{q_1, q_2, \ldots, q_n\},$$

With q_1 the initial state, q_2 the single final state, and

$$\Gamma = \{a_1,a_2,\ldots,a_m\},$$

where a_1 represents the blank. Then select an encoding in which q_1 is represented by 1, q_2 is represented by 11, and so on. Similarly, a_1 is encoded as 1, a_2 as 11, etc. The symbol 0 will be used as a separator between the 1's. With the initial and final state and the blank defined by this convention, any Turing machine can be described completely with δ only. The transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence. For example, $\delta\ (q_1,a_2,) = (q_1,a_3, L)$ might appear as

$$\ldots 1\ 0\ \ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1 \ldots .$$

It follows from this that any Turing machine has a finite encoding as a string on $\{0,1\}^+$, and that, given any encoding of M, one can decode it uniquely. Some strings will not represent any Turing machine (e.g., the strong 00011), but one can easily spot these, so they are of no concern.

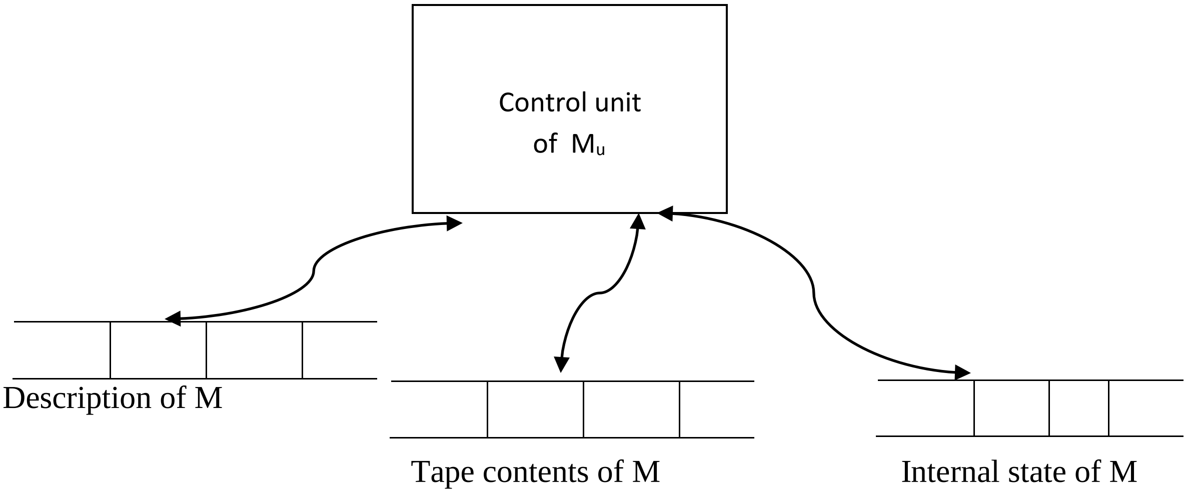


Figure: 16 Universal Turing Machine

A universal Turing machine M_u then has an input alphabet that includes (0,1} and the structure of a multitape machine, as shown in Figure 16.

For any input M and w, tape 1 will keep an encoded definition of M. Tape 2 will contain the tape contents of M, and tape 3 the internal state of M. M_u looks first at the contents of tapes 2 and 3 to determine the configuration of M. It then consults tape 1 to see what M would do in this configuration. Finally, tapes 2 and 3 will be modified to reflect the result of the move.

One can justify in claiming the existence of a Turing machine that, given any program, can carry out the computations specified by that program and that is therefore a proper model for a general purpose computer.

The observation that every Turing machine can be represented by a string of 0's and l's has important implications. But before exploring these implications, one need to review some results from set theory.

Some sets are finite, but most of the interesting sets (and languages) are infinite. For infinite sets, distinguish between sets that are countable and sets that are uncountable. A set is said to be countable if its elements can be put into a one-to-one correspondence with the positive integers. By this, it means that the elements of the set can be written in some order, say, x_1, x_2, x_3..., so that every element of the set has some finite index. For example, the set of all even integers can be written in the order 0, 2, 4...

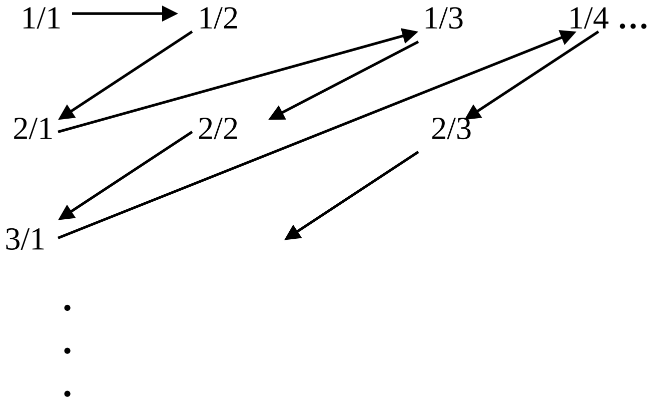


Figure: 17

Since any positive integer 2n occurs in position n + 1, the set is countable. This should not be too surprising, but there are more complicated examples, some of which may seem counterintuitive. Take the set of all quotients of the form p/q, where p and q are positive integers. How should one order this set to show that it is countable? One cannot write

$$\frac{1}{1},\frac{1}{2},\frac{1}{3},\frac{1}{4},....$$

because then $\frac{2}{3}$ would never appear. This does not imply that the set is uncountable; in this case, there is a clever way of ordering the set to show that it is in fact countable. Look at the scheme depicted in Figure 17, and write down the element in the order encountered following the arrows. This gives us

$$\frac{1}{1},\frac{1}{2},\frac{2}{1},\frac{1}{3},\frac{2}{2},....$$

Here the element $\frac{2}{3}$ occurs in the eighth place, and every element has some place in the sequence. The set is therefore countable.

From this example, it can be proved that a set is countable if a method is produced by which its elements can be written in some sequence. Such a method is called an enumeration procedure. Since an enumeration procedure is some kind of mechanical process, a Turing machine model can be used to define it formally.

3.7.1 Definition

Let S be a set of strings on some alphabet then an enumeration procedure for S is a Turing machine that can carry out the sequence of steps

$$q_0\square \vdash^{*} q_s x_1 \# s_1 \vdash^{*} q_s x_2 \# s_2 ...,$$

with x_1 € Γ * — {#}, S_i ∈ S, in such a way that any s in S is produced in a finite number of steps. The state q_s is a state signifying membership in *S*; that is, whenever q_s is entered, the string following # must be in S.

Not every set is countable. There are some uncountable sets, but any set for which an enumeration procedure exists is countable because the enumeration gives the required sequence.

Strictly speaking, an enumeration procedure cannot be called an algorithm, since it will not terminate when S is infinite. Nevertheless, it can be considered a meaningful process, because it produces well-defined and predictable results.

Let $\sum = \{a, b, c\}$. It can be shown that the $S = \sum^{+}$ is countable and find an enumeration procedure that produces its elements in some order, say in the order in which they would appear in a dictionary. However, the order used in dictionaries is not suitable without modification. In a dictionary, all words beginning with a are listed before the string b. But when there are an infinite number of a words, one will never reach b, thus violating the condition of Definition 3.7.1 that any given string be listed after a finite number of steps.

Instead, one can use a modified order, in which the length of the string can be taken as the first criterion, followed by an alphabetic ordering of all equal-length strings. This is an enumeration procedure that produces the sequence

a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa,....

As there will be several uses for such an ordering, it will be called in the **proper order.**

An important consequence of the above discussion is that Turing machines are **countable**.

Theorem 3.1

The set of all Turing machines, although infinite, is countable.

Proof: One can encode each Turing machine using 0 and 1. With this encoding, one can construct the following enumeration procedure.

1. Generate the next string in $\{0,1\}^{+}$ in proper order.
2. Check the generated string to see if it defines a Turing machine. If so, write it on the tape in the form required by Definition 3.7.1. If not, ignore the string.
3. Return to Step 1.

Since every Turing machine has a finite description, any specific machine will eventually be generated by this process.

The Particular ordering of Turing machines depends on the encoding used; if a different encoding is used, a different ordering is expected. This is of no consequence, however, and shows that the ordering itself is unimportant. What matters is the existence of some ordering.

CHAPTER 4

Formal Languages

Our immediate goal will be to examine the languages associated with Turing machines and some of their restrictions. Because Turing machines can perform any kind of algorithmic computation, it is expected to find that the family of languages associated with them is quite broad. Recursive and Recursively Enumerable Languages have been discussed in this chapter.

4.1 Recursive and Recursively Enumerable Language

Starting with some terminology for the languages associated with Turing machines, the important distinction between languages is made for which there exists an accepting Turing machine and languages for which there exists a membership algorithm. Because a Turing machine does not necessarily halt on input that it does not accept, the first does not imply the second.

4.1.1 Definition

A language L is said to be recursively enumerable if there exists a Turing machine that accepts it.

This definition implies only that there exists a Turing machine M, such that, for every w $\in L,$

$q_0w \vdash^*_M x_1q_f x_2$

with q_f a final state. The definition says nothing about what happens for w not in L; it may be that the machine halts in a nonfinal state or that it never halts and goes into an infinite loop. It can be more demanding if it is asked that the machine tells whether or not, any given input is in its language.

4.1.2 Definition

A language L on $\sum$ is said to be recursive if there exists a Turing machine M that accepts L and that halts on every w in $\sum^{+}$. In other words, a language is recursive if and only if there exists a membership algorithm for it.

If a language is recursive, then there exists an easily constructed enumeration procedure. Suppose that M is a Turing machine that determines membership in a recursive language L. First, construct another Turing machine, say $\hat{\mathrm{M}}$, which generates all strings in $\sum$+ in proper order, let us say w_1, w2..... As these strings are generated, they become the input to M, which is modified so that it writes strings on its tape only if they are in L.

That there is also an enumeration procedure for every recursively enumerable language is not as easy to see. One cannot use the above argument as it stands, because if some W_j is not in L, the machine M, when started with W_j on its tape, may never halt and therefore never get to the strings in L that follow w_j in the enumeration. To make sure that this does not happen, the computation is performed in a different way. First get M to generate w_1 and let M execute one move on w_2. Then, let M generate w_2 and let M execute one move on w_2, followed by the second move on w_1. After this, generate w_3 and do one step on w_3, the second step on w_2, the third step on w_1 and so on. The order of performance is depicted in Figure 18. From this, it is clear that M will never get into an infinite loop. Since any w € L is generated by M and accepted by M in a finite number of steps, every string in L is eventually produced by M.

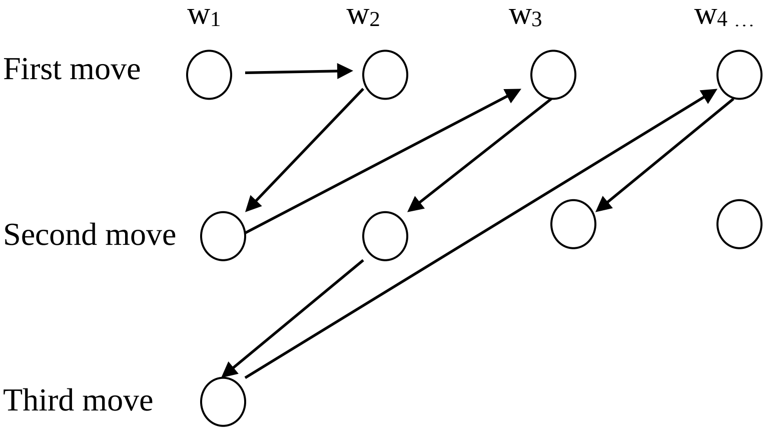


Figure: 18

It is easy to see that every language for which an enumeration procedure exists is recursively enumerable. Compare the given input string against successive strings generated by the enumeration procedure. If w € L, eventually got a match, and the process can be terminated.

Definitions 4.1.1 and 4.1.2 give us very little insight into the nature of either recursive or recursively enumerable languages. These definitions attach names to language families associated with Turing machines, but shed no light on the nature of representative languages in these families. Nor do they tell us much about the relationships between these languages or their connection to the language families encountered before. A question is faced such as "Are there languages that are recursively enumerable but not recursive?" and "Are there languages, describable somehow, that are not recursively enumerable?" While anyone will be able to supply some answers, anyone will not be able to produce very explicit examples to illustrate these questions, especially the second one.

4.1.3 Languages That Are Not Recursively Enumerable

The existence of languages can be established that are not recursively enumerable in a variety of ways. One is very short and uses a very fundamental and elegant result of mathematics.

Theorem 4.1

Let S be an infinite countable set. Then its power set 2^s is not countable.

Proof: Let $S = \{s_1, s_2, s_3...\}$. Then any element $t.$ of 2^s can be represented by a sequence of 0's and l's, with a 1 in position i if and only if S_i is in t. For example, the set $\{s_2,s_3,s_6\}$ is represented by 01100100..., while $\{s_1,s_3,s_5,...\}$ is represented by 10101.... Clearly, any element of 2^s can be represented by such a sequence, and any such sequence represents a unique element of 2^s. Suppose that 2^s were countable; then its elements could be written in some order, say $t_1,t_2\ldots$, and enter these into a table, as shown in Figure 19. In this table, take the elements in the main diagonal, and complement each entry, that is, replace 0 with 1, and vice versa. In the example in Figure 19, the elements are 1100..., so one gets 0011... as the result. The new sequence represents some element of 2^s, say t_1 for some *i*. But it cannot be t_1 because it differs from s_1 through s_1. For the same reason it cannot be t_2, t_3, or any other t_i. This contradiction creates a logical impasse that can be removed only by throwing out the assumption that 2^s is countable.

t_1	1	0	0	0	0	. . .
t_2	1	1	0	0	0	. . .
t_3	1	1	1	1	0	. . .
t_4	1	1	0	1	1	. . .
.						
.						
.						

Figure: 19

This kind of argument, because it involves a manipulation of the diagonal elements of a table, is called ***diagonalization***. The technique is attributed to the mathematician G. F. Cantor, who used it to demonstrate that the set of real numbers is not countable. Theorem 4.1 is diagonalization in its purest form.

As an immediate consequence of this result, it can be shown that, in some sense, there are fewer Turing machines than there are languages, so that there must be some languages that are not recursively enumerable.

Theorem 4.2

For any nonempty ∑, there exist languages that are not recursively enumerable.

Proof: A language is a subset of $\sum^*$, and every such subset is a language. Therefore the set of all languages is exactly 2^{Σ^*}. Since $\sum^*$ is infinite, Theorem 4.1 tells us that the set of all languages on $\sum$ is not countable. But the set of all Turing machines can be enumerated, so the set of all recursively enumerable languages is countable.

This proof, although short and simple, is in many ways unsatisfying. *It* is completely nonconstructive and, while it tells us of the existence of *some* languages that are not recursively enumerable, it gives us no feeling all for what these languages might look like. In the next set of results, investigate the conclusion more explicitly.

4.1.4 A Language That Is Not Recursively Enumerable

Since every language that can be described in a direct algorithmic fashion can *be* accepted by a Turing machine and hence is recursively enumerable, the description of a language that is not recursively enumerable must be indirect. Nevertheless, it is possible. The argument involves a variation on the diagonalization theme.

Theorem 4.3

There exists a recursively enumerable language whose complement is not recursively enumerable.

4.1.5 A Language That Is Not Recursively Enumerable But Not Recursive

Next, there are some languages that are recursively enumerable but not recursive.

Theorem 4.4

If a Language L and its complement $\overline{L}$ are both recursively enumerable, then both languages are recursive. If L is recursive, then $\overline{L}$ is also recursive, and consequently both are recursively enumerable.

Theorem 4.5

There exists a recursively enumerable language that is not recursive; that is, the family of recursive languages is proper subset of the family of recursively enumerable languages.

4. 2 Unrestricted Grammars

To investigate the connection between recursively enumerable languages and grammars, return to the general definition of a grammar G.

A grammar G is defined as a quadruple

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

Where V is finite set of objects called **variables,**

T is a finite set of objects called **terminal** symbols,

S € V is a special symbol called the **start** variable,

P is a finite set of **productions**.

In this definition the production rules were allowed to take any form, but various restrictions were later made to get specific grammar types. Taking the general form and imposing no restrictions, one gets unrestricted grammars.

4.2.1 Definition

A grammar $G = (V, T, S, P)$ is called unrestricted if all the productions are of the form

$$u \rightarrow v,$$

Where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$.

In an unrestricted grammar, essentially no conditions are imposed on the productions. Any number of variables and terminals can be on the left or right, and these can occur in any order. There is only one restriction: A is not allowed as the left side of a production.

Unrestricted grammars are much more powerful than restricted forms like the regular and context-free grammars. In fact, unrestricted grammars correspond to the largest family of languages so one can hope to recognize by mechanical means; that is, unrestricted grammars generate exactly the family of recursively enumerable languages. This is shown in two parts; the first is quite straightforward, but the second involves a lengthy construction.

Theorem 4.6

Any language generated by an unrestricted grammar is recursively enumerable.

Proof: The grammar in effect defines a procedure for enumerating all strings in the language systematically. For example, it can be listed all w in L such that

$$S => w,$$

that is, w is derived in one step. Since the set of the productions of the grammar is finite, there will be a finite number of such strings. Next, list all w in L that can be derived in two steps

$$S => x => w,$$

and so on. These derivations can be simulated on a Turing machine and, therefore, have an enumeration procedure for the language. Hence it is recursively enumerable.

This part of the correspondence between recursively enumerable languages and unrestricted grammars is not surprising. The grammar generates strings by a well-defined algorithmic process, so the derivations can be done on a Turing machine. To show the converse, describe how any Turing machine can be mimicked by an unrestricted grammar.

Given a Turing machine $M = (Q, \sum, \tau, \delta, q_0, \square, F)$ and want to produce a grammar G such that $L(G) = L(M)$. The idea behind the construction is relatively simple, but its implementation becomes notationally cumbersome.

Since the computation of the Turing machine can be described by the sequence of instantaneous descriptions

$$q_0w \vdash^{*} xq_fy, \qquad (4.3)$$

try to arrange it so that the corresponding grammar has the property that

$$q_0w =>^{*} xq_fy, \qquad (4.4)$$

if and only if (4.3) holds. This is not hard to do; what is more difficult to see is how to make the connection between (4.4) and what is really required, namely,

$$S =>^{*} w$$

for all w satisfying (4.3). To achieve this, construct a grammar which, in broad outline, has the following properties:

1. S can derive $q_o w$ for all w € $\sum^+$.
2. (4.4) is possible if and only if (4.3) holds.
3. When a string $xq_f y$ with q_f € F is generated, the grammar transforms this string into the original w.

The complete sequence of derivations is then

$$S =>^* \quad q_0 w =>^* \; x q_f y =>^* w. \qquad (4.5)$$

The third step in the above derivation is the troublesome one. How can the grammar remember w if it is modified during the second step? Solve this by encoding strings so that the coded version originally has two copies of w. The first is saved, while the second is used in the steps in (4.4). When a final configuration is entered, the grammar erases everything except the saved w.

To produce two copies of w and to handle the state symbol of M (which eventually has to be removed by the grammar), introduce variables V_{ab} and V_{aib} for all $a \in \sum U \ \{\square\}$, $b \ \in \tau$, and all i such that $q_i \ \in Q$. The variable V_{ab} encodes the two symbols a and b, while V_{aib} encodes a and b as well as the state q_i.

The first step in (4.5) can be achieved (in the encoded form) by

$$S \rightarrow V_{\square\square} S \,|\, S V_{\square\square} \,|\; T, \qquad (4.6)$$

$$T \rightarrow T V_{aa} \,|\, V_{a0a}, \qquad (4.7)$$

for all $a \ \in \sum$. These productions allow the grammar to generate an encoded version of any string q_0'W with an arbitrary number of leading and trailing blanks.

For the second step, for each transition

$$\delta = (q_i, \ c) = (q_j, d, R)$$

of M, put into the grammar productions

$$V_{aic}V_{pq} \rightarrow V_{ad}V_{pjq} \qquad (4.8)$$

for all a, p € ∑ U {□}, q € T. For each

$$\delta = (q_i, c) = (q_j, d, L)$$

of M, include in G

$$V_{pq}V_{aic} \longrightarrow V_{pjq}V_{ad},$$

(4.9)

for all a, p ∈ ∑ U {□}, q ∈ τ.

If in the second step, M enters a final state, the grammar must then get rid of everything except which is saved in the first indices of the $V's$. Therefore, for every $q_{\}}$ G F, include productions

$$V_{ajb} \rightarrow a, \qquad (4.10)$$

for all a ∈ ∑ U {□}, b € τ. This creates the first terminal in the string, which then causes a rewriting in the rest by

$$cV_{ab} \rightarrow ca, \qquad (4.11)$$

$$V_{abc} \rightarrow ac, \qquad (4.12)$$

for all a, c ∈ ∑ U {□}, b ∈ τ. One more special production is needed

$$\square \rightarrow \lambda. \qquad (4.13)$$

This last production takes care of the case when M moves outside that part of the tape occupied by the input w. To make things work in this case, first use (4.6) and (4.7) to generate

$$\square \ldots \square q_0 w \square \ldots \square,$$

representing the entire tape region used. The extraneous blanks are removed at the end by (4.13).

The following example illustrates this complicated construction. Carefully check each step in the example to see what the various productions do and why they are needed.

Let $M = (Q, \sum, \tau, \delta, q_0, \square, F)$ be a Turing machine with

$$Q = \{q_0,q_1\},$$

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

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

$$F = \{q_1\},$$

and

$$\delta(q_0, a) = (q_0, a, R),$$

$$\delta(q_0, \square) = \{q_1, \square, L).$$

This machine accepts $L\ (aa^*)$.

Consider now the computation

$$q_0aa \vdash aq_0a \vdash aaq_0\square \vdash aq_1a\square \qquad (4.14)$$

which accepts the string aa. To derive this string with G, use rules of the form (4.6) and (4.7) to get the appropriate starting string,

$$S \rightarrow SV\square\square \rightarrow TV\square\square \rightarrow TVaa\square\square \rightarrow Va_0a\ Vaa\ V\square\square.$$

The last sentential form is the starting point for the part of the derivation that mimics the computation of the Turing machine. It contains the original input aa□ in the sequence of first indices and the initial instantaneous description $q_0aa\square$ in the remaining indices. Next, apply

$$V_{a0a}V_{aa} \rightarrow V_{aa}\ Va_0a,$$

and

$$V_{a0a}V\square\square \rightarrow V_{aa}V\square_0\square,$$

which are specific instances of (4.8), and

$$V_{aa}V\square_0\square \rightarrow Va1aV\square\square$$

coming from (4.9). Then the next steps in the derivation are

$$Va0aV_{aa}V\square\square \rightarrow V_{aa}V_{a0a}V\square\square => V_{aa}V_{aa}V\square0\square => V_{aa}V_{a1a}V\square\square$$

The sequence of first indices remains the same, always remembering the initial input. The sequence of the other indices is

$$0aa\square,\ a0a\square,\ aa0\square,\ a1a\square,$$

this is equivalent to the sequence of instantaneous descriptions in (4.14). Finally, (4.10) to (4.13) are used in the last steps

$$V_{aa}V_{a1a}V_{\square\square} \rightarrow V_{aa}aV_{\square\square} \rightarrow V_{aa}a\ \square \rightarrow aa\square \rightarrow aa.$$

The construction described in (4.6) to (4.13) is the basis of the proof of the following result.

Theorem 4.7

For every recursively enumerable language L, there exists an unrestricted grammar G, such that $L = L$ (G).

Proof: The construction described guarantees that if

$$x \vdash y,$$

then

$$e\ (x) \longrightarrow e\ (y),$$

Where e (x) denotes the encoding of a string according to the given convention. By an induction on the number of steps, it can then be shown that

$$e\ (q_0w) \xrightarrow{*} e(y),$$

if and only if

$$q_0w \vdash^{*} y.$$

It must be shown that one can generate every possible starting configuration and that w is properly reconstructed if and only if M enters a final configuration.

These two theorems establish what is set out to do. They show that the family of languages associated with unrestricted grammars is identical with the family of recursively enumerable languages.

4.3 Context-Sensitive Grammars and Languages

Between the restricted, context-free Grammars and the general, unrestricted Grammars, a great variety of "somewhat restricted" Grammars can be defined. Not all cases yield interesting results; among the ones that do, the context-sensitive Grammars have received considerable attention. These Grammars generate languages associated with a restricted class of Turing machines, linear bounded automata.

Definition: 4.3.1

A grammar $G = (V, T, S, P)$ is said to be context-sensitive if all the productions are of the form

$$x \rightarrow y,$$

where x, y is in $(V \cup T)^+$ and

$$|\mathbf{x}| \leq |\mathbf{y}|$$

4.4 Relation between Recursive and Context-Sensitive Languages

Every Context-Sensitive Language is accepted by some Turing machine and is therefore recursively enumerable.

Theorem 4.8

Every Context-Sensitive Language L is recursive.

Theorem 4.9

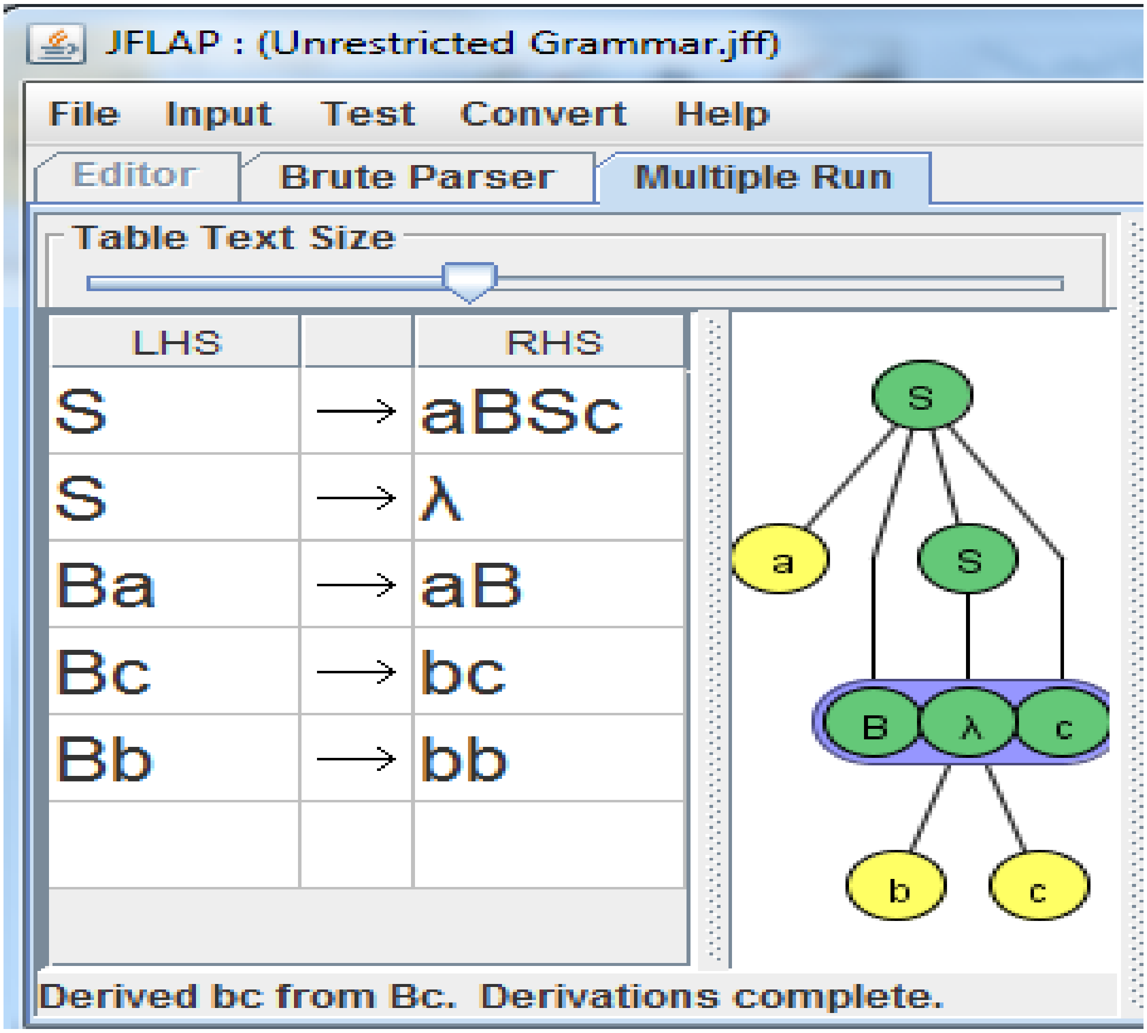
There exists a recursive language that is not context-sensitive.

These theorems 4.8 and 4.9 has been taken from "An Introduction to formal languages and automata", third edition authored by Peter Linz. The result in theorem 4.9 indicates the linear bounded automata are indeed less powerful than Turing machines, since they accept only a proper subset of the recursive languages. It follows from the same results that linear bounded automata are more powerful than pushdown automata. Context-free languages, being generated by Context-free grammars, are a subset of the context sensitive languages. They are a proper subset. Because of the essential equivalence of linear bounded automata and context-free languages on the other, it can be seen that any languages accepted by a pushdown automata is also accepted by some linear bounded automata, but that there are languages accepted by some linear bounded automata for which there are no pushdown automata.

CHAPTER 5

Implementation & Result

The implementation of a Turing Machine for the JFLAP platform is described. JFLAP is most successful and widely used tool for visualizing and simulating automata such as finite state machines, pushdown automata, and Turing Machines. By executing our Turing Machine in JFLAP, everyone get a direct and interactive experience of how this Turing Machine is capable of emulating other Turing Machines.

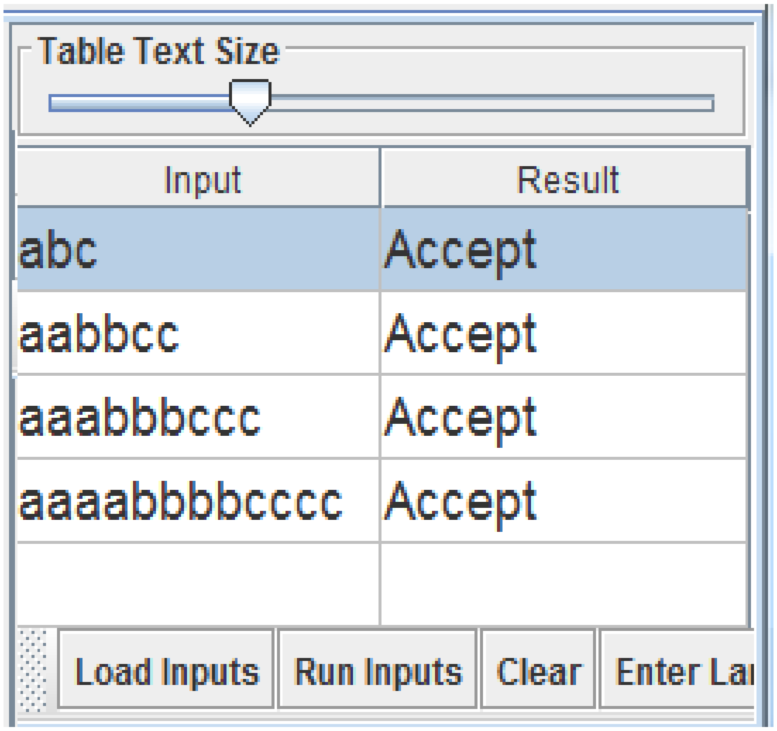


LHS		RHS
S	→	aBSc
S	→	λ
Ba	→	aB
Bc	→	bc
Bb	→	bb

Derived bc from Bc. Derivations complete.

Figure: 20 Unrestricted Grammar (with parse tree abc)

Any language generated by an unrestricted grammar is recursively enumerable. For every recursively enumerable language L, there exists an unrestricted grammar G, such that L = L (G). Language L is said to be recursively enumerable if there exists a Turing Machine that accepts it. By taking unrestricted grammar, this is shown in figure 20. The various strings are applied to this unrestricted grammar (which is also shown in figure 21).



Input	Result
abc	Accept
aabbcc	Accept
aaabbbccc	Accept
aaaabbbbcccc	Accept

Figure :21 Accepted String by Unrestricted Grammar

This grammar drives the language L= $a^nb^nc^n$, n > 0. The language $a^nb^nc^n$ is a recursively enumerable language which cannot be implemented using a FA as well as a PDA. The standard Turing machine T_M for the language $a^nb^nc^n$ is given in figure 22. The various strings are applied to the Turing Machine with multiple run. A few results are shown in figure 23.

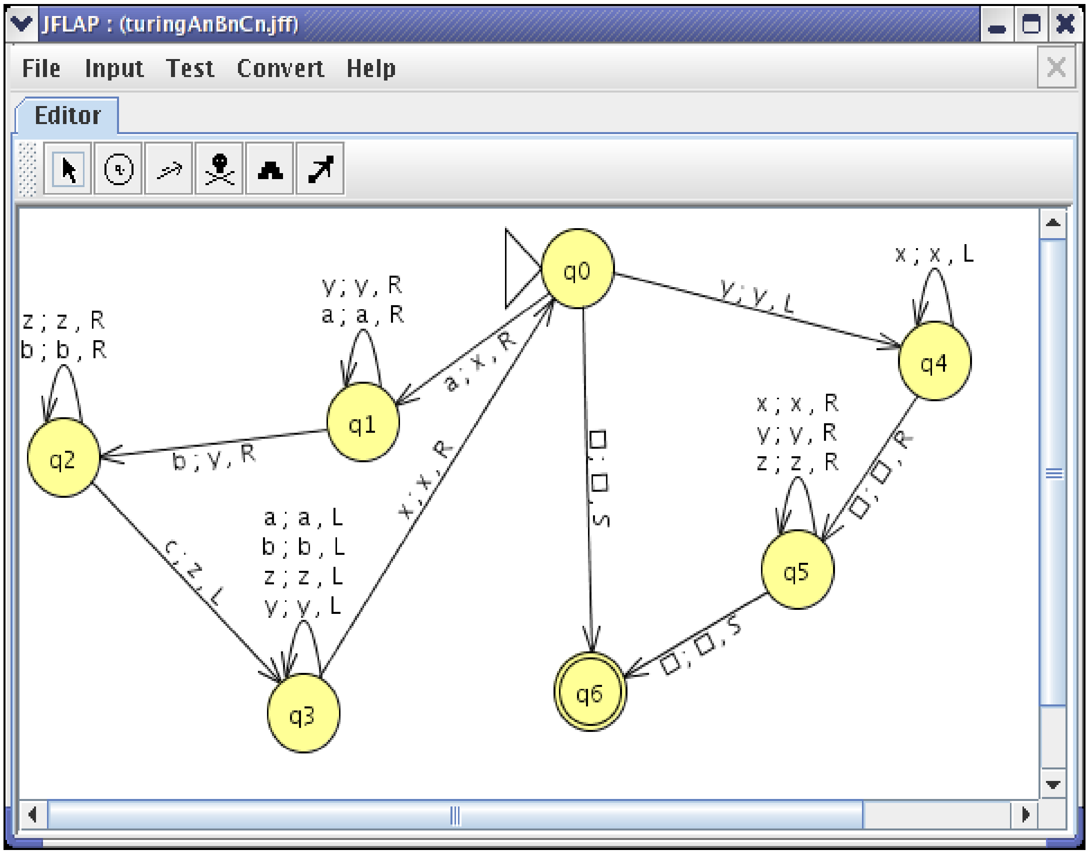
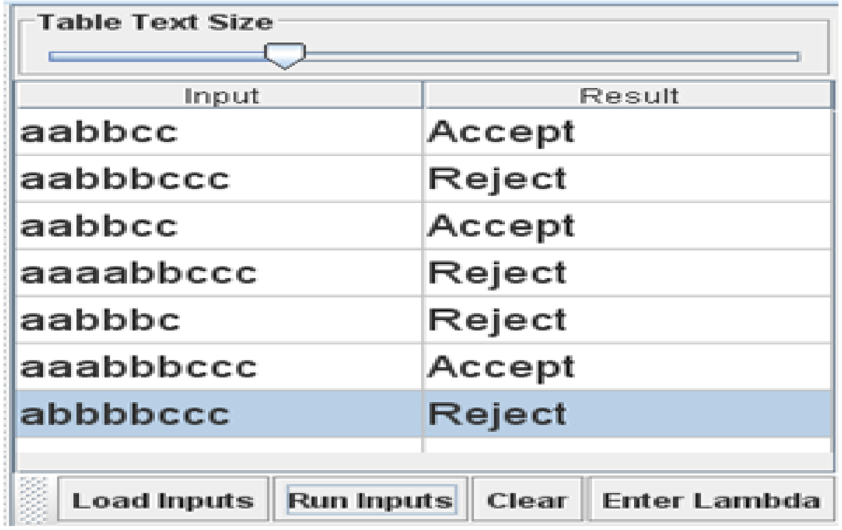


Figure :22 Turing Machine for $a^nb^nc^n$



Input	Result
aabbcc	Accept
aabbbccc	Reject
aabbcc	Accept
aaaabbccc	Reject
aabbbc	Reject
aaabbbccc	Accept
abbbbccc	Reject

Figure: 23 Multiple Run by Turing Machine

Any Turing machine can be converted REL into an unrestricted grammar. JFLAP defines an unrestricted grammar as a grammar that is similar to a context-free grammar (CFG), except that the left side of a production may contain any nonempty string of terminals and variables, rather than just a single variable. In an unrestricted grammar, the left side of a production is matched, which may be multiple symbols, and replaced by the corresponding right hand side.

There are 3 major steps that the algorithm follows in order to convert the Turing machine to an unrestricted grammar.

1. The first step is applying the basic rules from the start variable. It allows the grammar to generate an encoded version of any string q_0w with an arbitrary number of leading and trailing blanks.

2. The second step is applying generating the production for each transition of Turing machine.

3. The last step is applying additional rules to our productions if one of final states of TM is entered, so terminals can be derived.

CHAPTER 6

Conclusion & Future Work

Turing Machines are the most powerful computational machines. The Turing Machines provide an abstract model to all the problems. This work describes the working of a Turing Machine for Recursively Enumerable Languages and Unrestricted Grammar for JFLAP platform.

Regular languages form a proper subset of Context Free Languages. So PDAs are more powerful than finite automata. But CFLs are limited in scope because many of the simple languages like $a^nb^nc^n$ are not context free. So to incorporate the set of all languages that are not accepted by PDAs and hence that are not context free, more powerful language families has been formed. This creates the class of Recursively Enumerable Languages (REL). The finite automaton has no mechanism for storage. The PDA is more powerful than FAs as they have the stack as the temporary storage. The new class of languages REL is accepted by a new type of automaton that has the most powerful storage as well as the computation mechanisms. This created the Turing Machine (TM). A TM's storage has an infinite tape, extendable in both directions. The tape is divided into cells, each cell capable of holding one symbol. The information can be read as well as changed in any order. The TM has the capability of holding unlimited amount of information. Turing Machines work for regular languages, CFLs as well as RELs.

The Turing Machines differ from all other automata as it can work with Recursively Enumerable Languages and Unrestricted Grammar. Any language generated by an unrestricted grammar is recursively enumerable. The language $a^nb^nc^n$ is a recursively enumerable language which cannot be implemented using a Finite Automata or a PDA but can done using a Turing Machine. This requires more storage than for Context Free Languages and hence the Turing Machine with the infinite tapes, extendable in both directions is used for this.

Future work

For a deterministic Turing machine with m symbols in the alphabet such that $| \sum | = m$ and total number of states n, m X n transitions are possible. A Universal Turing Machine (UTM) with n states, $| \sum | = m$ and p possible directions branches into m X n X p states for execution. A problem that can be solved with a multitape Turing machine with m tapes in O (n) moves can be done with a UTM in O (n^m) moves. It is proposed system as:

i. Developing a Universal Turing Machine for recursively enumerable languages

ii. Implement the concept of universality by including more symbols in the input alphabet as well as the tape alphabet

iii. Developing a Universal Turing Machine for Indian Languages