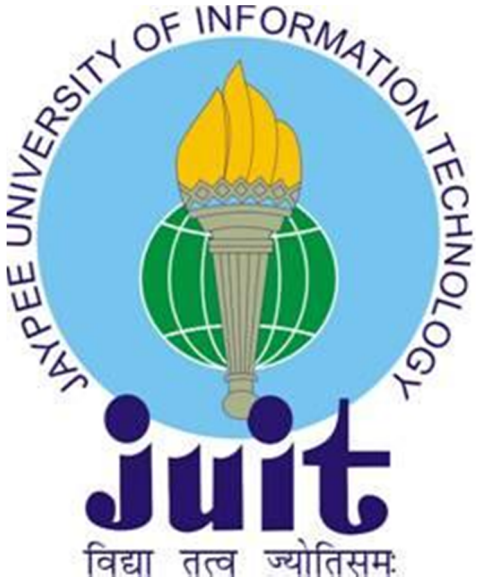
IMPLEMENTATION AND SIMULATION OF UNIVERSAL TURING MACHINE

ADITYA PRAKASH SRIVASTAVA (101213)

SOURABH DHANOTIA (101299)

MOHAMMAD SHARIQUE KHAN (101311)

UNDER SUPERVISION OF **DR. NITIN**



Submitted in partial fulfilment of the Degree of Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT, SOLAN

Certificate

This is to certify that the work titled **"IMPLEMENTATION AND SIMULATION OF UNIVERSAL TURING MACHINE"** submitted by **Aditya Prakash Srivastava, Sourabh Dhanotia** and **Mohammad Sharique Khan** in partial fulfilment for the award of degree of Bachelor of Technology of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor	……..
Date	……..
Name of Supervisor	**Dr. Nitin**
Designation	Associate Professor Department of CSE and ICT, Jaypee University of Information Technology , Waknaghat, Solan-173234, H.P.

Acknowledgement

On the very outset of this report, we would like to extend our sincere and heartfelt obligation towards all the personages who have helped us in this endeavour. Without their active guidance, help, cooperation & encouragement, we would not have made headway in the project.

We take this opportunity to express our profound gratitude and deep regards to our project guide **Dr. Nitin** for his exemplary guidance, monitoring and constant encouragement throughout the course of this project.

We also extend our gratitude to Jaypee University of Information Technology for giving us this opportunity.

……	……	……...
Aditya Prakash Srivastava	**Sourabh Dhanotia**	**Mohammad Sharique Khan**
[101213]	**[101299]**	**[101311]**

Table of contents

TOPIC **PAGE**

List of figures

1 Introduction

In computer science, a universal Turing machine (UTM) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape.

Alan Turing introduced this machine in 1936–1937. This model is considered to be the origin of the stored program computer—used by John von Neumann (1946) for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture. It is also known as universal computing machine, universal machine (UM).In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates.

Every Turing machine computes a certain fixed partial computable function from the input strings over its alphabet. In that sense it behaves like a computer with a fixed program. However, we can encode the action table of any Turing machine in a string. Thus we can construct a Turing machine that expects on its tape a string describing an action table followed by a string describing the input tape, and computes the tape that the encoded Turing machine would have computed.

Turing described such a construction in complete detail in his 1936 paper: "It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M."

2 Problem Statement and Motivation

2.1 Motivation

A major problem in computer science education is that many students obtain only a superficial understanding of theory, even though theoretical concepts provide the fundamental basis for most areas of computer science. In particular, a thorough understanding of the theory of computation (TOC) is crucial in designing programming languages and compilers. However, the traditional TOC course is taught in a non-visual and pencil-paper problem solving manner with no programming component. Students find this approach frustrating as they have no visualization to relate to, they do not receive immediate feedback on problems, and furthermore it is tedious and uninteresting to them to determine if their solutions are correct.

Many students turn in homework with errors as they do not bother to verify their solutions by hand. This contrasts starkly with the hands-on nature in most of their other computer science courses which contain programming assignments. In addition, many students leave the TOC course without understanding the importance of this material. They are told it has applications in other areas, but they don't experience the applications.

The material in the traditional FLA (formal languages and automata) or TOC course is important to the computer science major, as mentioned in the Computer Science Volume of the ACM IEEE Computing Curricula 2001 document in several places.

We are trying to develop a tool that allows students of TOC to experiment with the theory in this course, receiving immediate feedback, and making the course more interesting to students. With this tool one will be able to interactively build automata, Moore and Mealy machines and pushdown automata and run test input on them. With this tool one will be able to visualise how a Turing machine works. With this tool, one will be able to compute any computational problem with the help of universal Turing machine.

2.2 Problem statement

To develop a tool to help in visualizing and interacting with theoretical models and concepts in computer science. This tool will provide visualisation of FSM, Mealey machines, Moore machines, pushdown automata, Turing machine and universal Turing machine.

To develop a means of experimentation of construction-type proofs in formal languages. After learning the algorithms like Thomson construction, students can match their answer with the automata generated by this tool.

To develop means for experimenting with formal languages in a computational manner that can only be done with the use of a computer. Universal Turing machine will be able to compute any computational problem.

Hence enhance the understanding of the 'Theory of computation' course of the user.

3 Literature review

3.1 Finite state machine

A finite-state machine (FSM) or finite-state automaton (plural: automata), or simply a state machine, is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

The behaviour of state machines can be observed in many devices in modern society which perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines which dispense products when the proper combination of coins is deposited, elevators which drop riders off at upper floors before going down, traffic lights which change sequence when cars are waiting, and combination locks which require the input of combination numbers in the proper order.

In accordance with the general classification, the following formal definitions are found:

1. A *deterministic finite state machine* or acceptor deterministic finite state machine is a quintuple $(\sum, S, s_0, \delta, F)$, where:
    - $\sum$ is the input alphabet (a finite, non-empty set of symbols).
    - S is a finite, non-empty set of states.
    - s_0 is an initial state, an element of 'S'.
    - δ is the state-transition function: $\delta: S \times \sum \rightarrow S$ (in a nondeterministic finite automaton it would be $\delta : S \times \sum \rightarrow P(S)$, i.e., would return a set of states).
    - F is the set of final states, a (possibly empty) subset of S.

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. $\delta(q,x)$ does not have to be defined for every combination of $q \in S$ and $x \in \sum$. If an FSM M is in a state q, the next symbol is x and $\delta(q,x)$ is not defined, then M can announce an error (i.e. reject the input). This is useful in definitions of general state machines, but less useful when transforming the machine.

A finite-state machine is a restricted Turing machine where the head can only perform "read" operations, and always moves from left to right.

2. A *finite state transducer* is a sextuple $(\sum, \Gamma, S, s_0, \delta, \omega)$, where:
    - $\sum$ is the input alphabet (a finite non empty set of symbols).
    - Γ is the output alphabet (a finite, non-empty set of symbols).
    - S is a finite, non-empty set of states.
    - s_0 is the initial state, an element of . In a nondeterministic finite automaton, is a set of initial states.
    - δ is the state-transition function: .
    - ω is the output function.

If the output function is a function of a state and input alphabet ($\omega: S \times \sum \rightarrow \Gamma$) that definition corresponds to the Mealy model, and can be modelled as a Mealy machine. If the output function depends only on a state ($\omega: S \rightarrow \Gamma$) that definition corresponds to the Moore model, and can be modelled as a Moore machine. A finite-state machine with no output function at all is known as a semi automaton or transition system.

3.1.1 Mealy machine

Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs. A Mealy machine is a deterministic finite state transducer: for each state and input, at most one transition is possible.

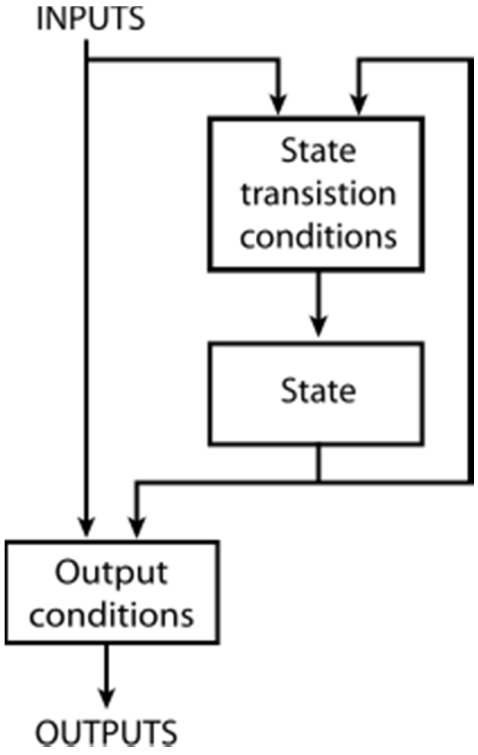


Fig 1: Mealy machine

Source: Wikipedia

Formal definition:

A Mealy machine is a 6-tuple, $(S, S_0, \Sigma, \Lambda, T, G)$, consisting of the following:

- a finite set of states (S)
- a start state (also called initial state) S_0 which is an element of (S)
- a finite set called the input alphabet (Σ)
- a finite set called the output alphabet (Λ)
- a transition function ($T : S \times \Sigma \rightarrow S$) mapping pairs of a state and an input symbol to the corresponding next state.
- an output function ($G : S \times \Sigma \rightarrow \Lambda$) mapping pairs of a state and an input symbol to the corresponding output symbol.

In some formulations, the transition and output functions are coalesced into a single function ($T : S \times \Sigma \rightarrow S \times \Lambda$).

3.1.2 Moore machine

Moore machine is a finite-state machine whose output values are determined solely by its current state. This is in contrast to a Mealy machine, whose output values are determined both by its current state and by the values of its inputs.

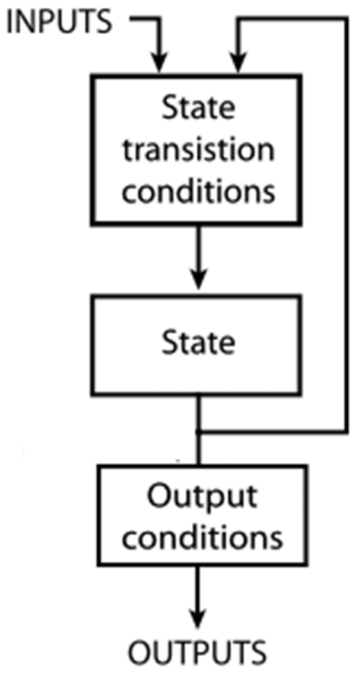


Fig 2:

Moore machine

Formal definition:

A Moore machine can be defined as a 6-tuple ($S, S_0, \Sigma, \Lambda, T, G$) consisting of the following:

- a finite set of states (S)
- a start state (also called initial state) S_0 which is an element of (S)
- a finite set called the input alphabet (Σ)
- a finite set called the output alphabet (Λ)
- a transition function ($T : S \times \Sigma \rightarrow S$) mapping a state and the input alphabet to the next state
- an output function ($G : S \rightarrow \Lambda$) mapping each state to the output alphabet

A Moore machine can be regarded as a restricted type of finite state transducer.

3.2 Pushdown automata

A pushdown automaton (PDA) is a type of automaton that employs a stack. It is more capable than a finite-state machine but less capable than a Turing machine. Because its input can be described with a formal grammar, it can be used in parser design. The deterministic pushdown automaton can handle all deterministic context-free languages while the nondeterministic version can handle all context-free languages.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A stack automaton, by contrast, does allow access to and operations on deeper elements. Stack automata can recognize a strictly larger set of languages than deterministic pushdown automata.

Formal definition:

We use standard formal language notation: Γ^* denotes the set of strings over alphabet Γ and ε denotes the empty string.

A PDA is formally defined as a 7-tuple: $M = (Q, \sum, \Gamma, \delta, q0, Z, F)$ where

- Q is a finite set of states
- $\sum$ is a finite set which is called the input alphabet
- Γ is a finite set which is called the stack alphabet
- δ is a finite subset of , the transition relation.

- $q_0 \in Q$ is the start state
- $Z \in \Gamma$ is the initial stack symbol
- $F \subseteq Q$ is the set of accepting states

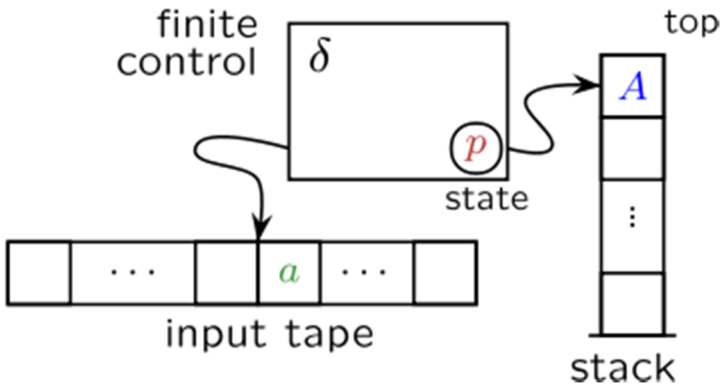


Fig 3:

Working PDA

Source: Wikipedia

An element $(p, a, A, q, \alpha) \in \delta$ is a transition of M. It has the intended meaning that M, in state $p \in Q$, with $a \in \sum U \{\varepsilon\}$ on the input and with $A \in \Gamma$ as topmost stack symbol, may read a, change the state to q, pop A, replacing it by pushing $\alpha \in \Gamma^*$. The $(\sum U \{\varepsilon\})$ component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

Here $\delta(p, a, A)$ contains all possible actions in state p with A on the stack, while reading a on the input. One writes $(q, \alpha) \in \delta(p, a, A)$ for the function precisely when $\delta(p, a, A, q, \alpha) \in \delta$ for the relation. Note that finite in this definition is essential.

Pushdown automata differ from finite state machines in two ways:

- They can use the top of the stack to decide which transition to take.
- They can manipulate the stack as part of performing a transition.

Pushdown automata choose a transition by indexing a table by input signal, current state, and the symbol at the top of the stack. This means that those three parameters completely determine the transition path that is chosen. Finite state machines just look at the input signal and the current state: they have no stack to work with. Pushdown automata add the stack as a parameter for choice.

Pushdown automata can also manipulate the stack, as part of performing a transition. Finite state machines choose a new state, the result of

following the transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack, and leave it as it is. The choice of manipulation (or no manipulation) is determined by the transition table.

Put together: Given an input signal, current state, and stack symbol, the automaton can follow a transition to another state, and optionally manipulate (push or pop) the stack.

In general, pushdown automata may have several computations on a given input string, some of which may be halting in accepting configurations. If only one computation exists for all accepted strings, the result is a deterministic pushdown automaton (DPDA) and the language of these strings is a deterministic context-free language. Not all context-free languages are deterministic. As a consequence of the above the DPDA is a strictly weaker variant of the PDA and there exists no algorithm for converting a PDA to an equivalent DPDA, if such a DPDA exists.

If we allow a finite automaton access to two stacks instead of just one, we obtain a more powerful device, equivalent in power to a Turing machine. A linear bounded automaton is a device which is more powerful than a pushdown automaton but less so than a Turing machine.

3.3 Context free grammar

In formal language theory, a context-free grammar (CFG) is a formal grammar in which every production rule is of the form $V \rightarrow w$ where V is a single nonterminal symbol, and w is a string of terminals and/or non terminals (w can be empty). A formal grammar is considered "context free" when its production rules can be applied regardless of the context of a nonterminal. It does not matter which symbols the nonterminal is surrounded by, the single nonterminal on the left hand side can always be replaced by the right hand side.

Languages generated by context-free grammars are known as context-free languages (CFL). Different Context Free grammars can generate the same context free language. It is important to distinguish properties of the language (intrinsic properties) from properties of a particular grammar (extrinsic properties). Given two context free grammars, the language equality question (do they generate the same language?) is undecidable.

Context-free grammars are important in linguistics for describing the structure of sentences and words in natural language, and in computer science

for describing the structure of programming languages and other formal languages.

Formal definition:

A context-free grammar G is defined by the 4-tuple:

$G = (V, \sum, R, S)$ where

- V is a finite set; each element $v \in V$ is called a non-terminal character or a variable. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
- $\sum$ is a finite set of terminals, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
- R is a finite relation from V to $(V \cup \sum)^*$, where the asterisk represents the Kleene star operation. The members of R are called the (rewrite) rules or productions of the grammar. (also commonly symbolized by a P)
- S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

A regular grammar

Every regular grammar is context-free, but not all context-free grammars are regular. The following context-free grammar, however, is also regular.

$S \rightarrow a$

$S \rightarrow aS$

$S \rightarrow bS$

The terminals here are a and b, while the only non-terminal is S. The language described is all nonempty strings of a_s and b_s that end in a.This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.Using pipe symbols, the grammar above can be described more tersely as follows:

$S \rightarrow a \mid aS \mid bS$

3.4 Regular expression

In computing, a regular expression is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

Each character in a regular expression is either understood to be a meta character with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The pattern sequence itself is an expression that is a statement in a language designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text processing of general text files, specific textual forms, or of random input strings.

A regular expression processor processes a regular expression statement expressed in terms of a grammar in a given formal language, and with that examines the target text string, parsing it to identify substrings that are members of its language, the regular expressions.

Regular expressions are so useful in computing that the various systems to specify regular expressions have evolved to provide both a basic and extended standard for the grammar and syntax; modern regular expressions heavily augment the standard. Regular expression processors are found in several search engines, search and replace dialogs of several word processors and text editors, and in the command lines of text processing utilities.

Formal definition:

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (Empty set) ∅ denoting the set ∅.
- (Empty string) ε denoting the set containing only the "empty" string, which has no characters at all.
- (Literal character) a in Σ denoting the set containing only the character a.
- Given regular expressions R and S, the following operations over them are defined to produce regular expressions:
- (concatenation) RS denoting the set { αβ | α in set described by expression R and β in set described by S }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.

- (alternation) R | S denoting the set union of sets described by R and S. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression R | S describes {"ab", "c", "d", "ef"}.
- (Kleene star) R* denoting the smallest superset of set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by R. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.
-

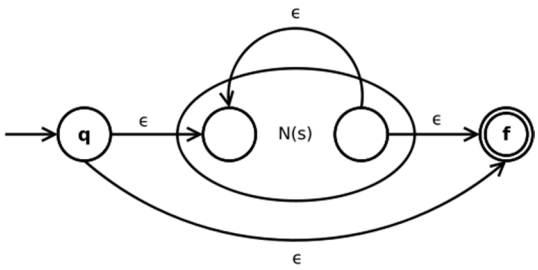


Fig 4: Kleene star

Source: Wikipedia

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, (ab)c can be written as abc, and a|(b(c*)) can be written as a|bc*. Many textbooks use the symbols ∪, +, or ∨ for alternation instead of the vertical bar.

Examples:

- a|b* denotes {ε, "a", "b", "bb", "bbb", ...}
- (a|b)* denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}
- ab*(c|ε) denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", ...}

3.5 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 mathematically models a machine that mechanically operates on a tape. On this tape are symbols, which the machine can read and write, one at a time, using a tape head. Operation is fully

determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc.

More precisely, a Turing machine consists of:

1. A **tape** divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.
2. A **head** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
3. A **state register** that stores the state of the Turing machine, one of finitely many. Among these is the special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.
4. A finite **table** (occasionally called an **action table** or **transition function**) of instructions (usually quintuples [5-tuples] : $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes quadruples [4-tuples]) that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under the head) tells the machine to do the following in sequence (for the 5-tuple models):
   - Either erase or write a symbol (replacing a_j with a_{j1}), *and then*
   - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
   - Assume the same or a *new state* as prescribed (go to state q_{i1}).

In the 4-tuple models, erasing or writing a symbol (a_{j1}) and moving the head left or right (d_k) are specified as separate instructions. Specifically, the table tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

Note that every part of the machine (i.e. its state and symbol-collections) and its actions (such as printing, erasing and tape motion) is *finite*, *discrete* and *distinguishable*; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

4 Designing

4.1 Project timeline

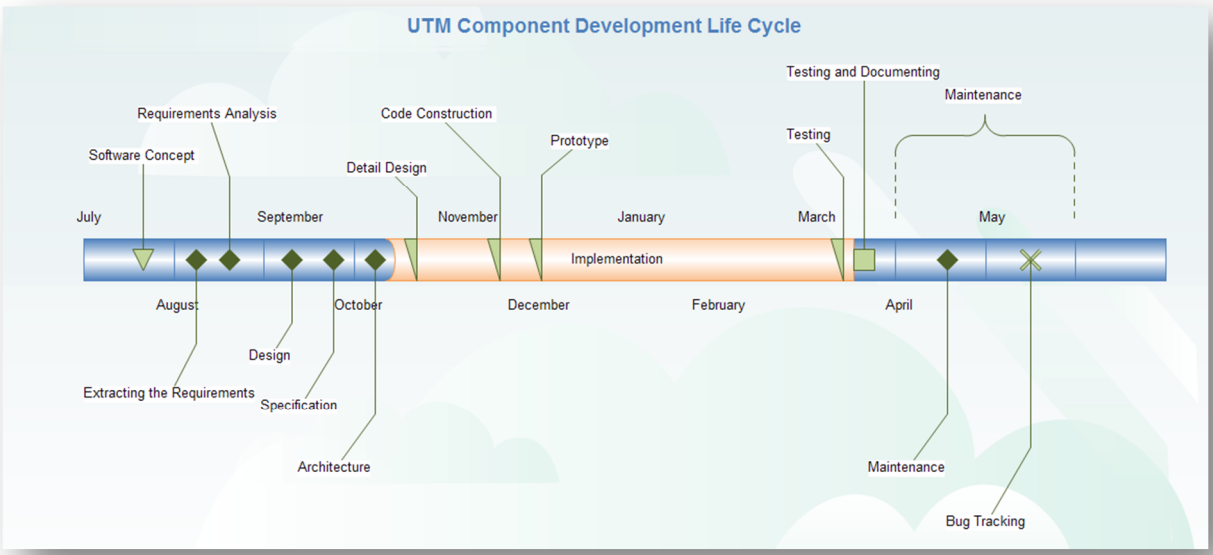


Fig 5: Project timeline

4.2 Event timeline

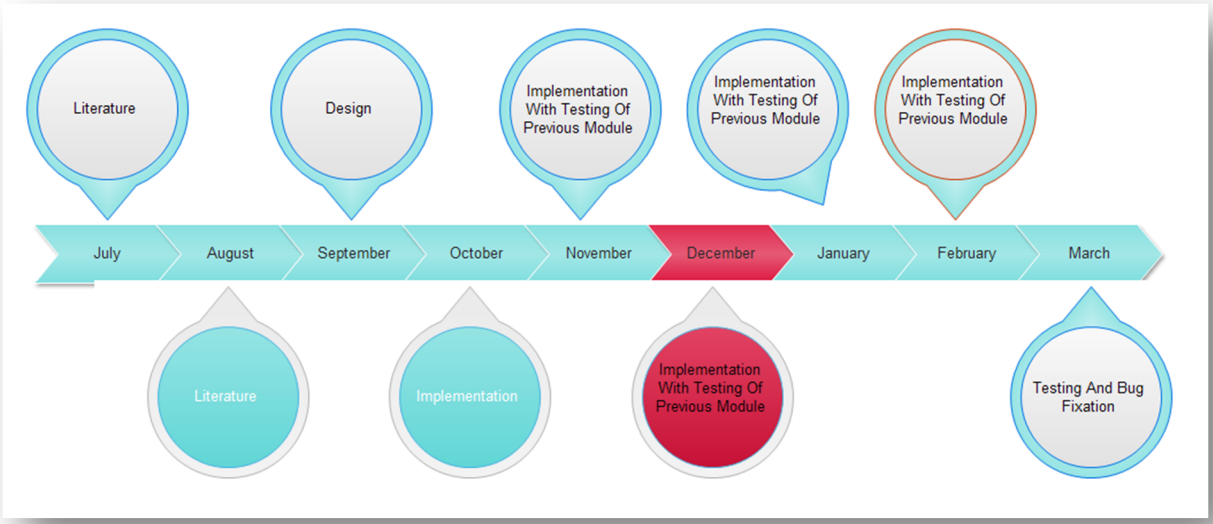


Fig 6: Event timeline

4.3 Use case diagram

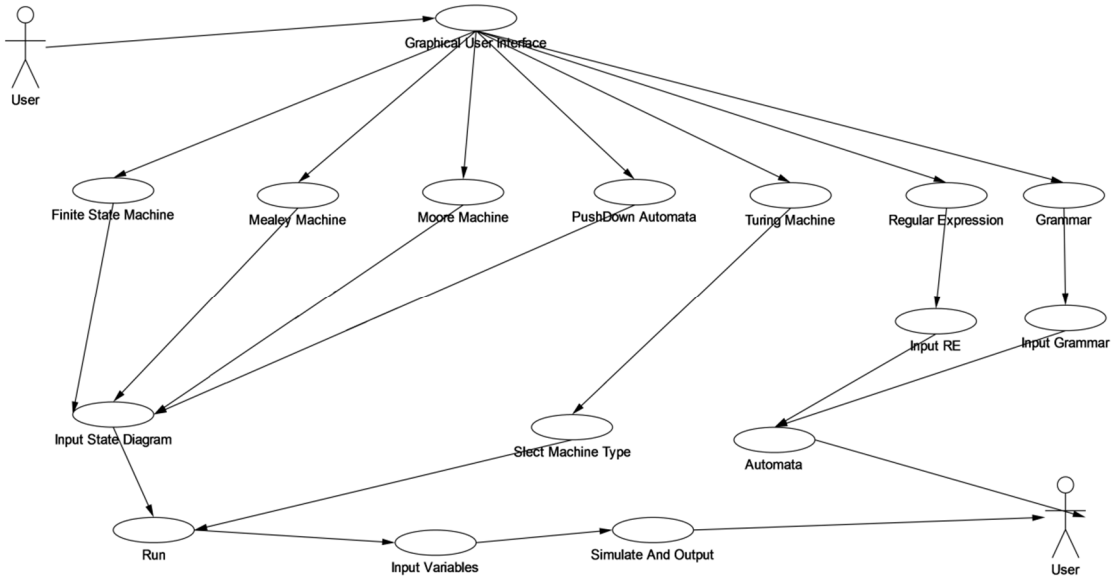


Fig 7: Use case diagram

4.4 Data flow diagram

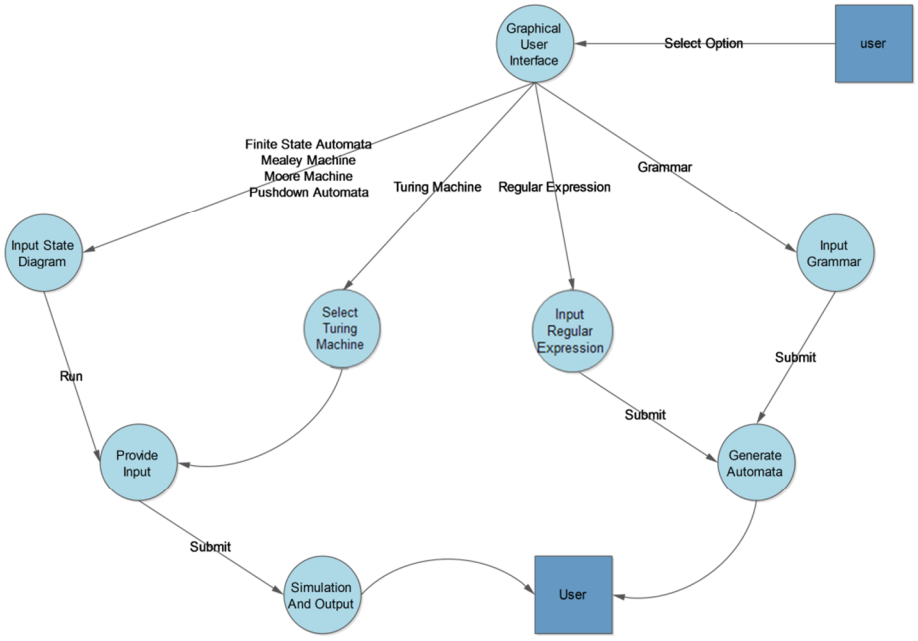


Fig 8: Data flow diagram

5 Work done

5.1 Work done till 7th semester

A brief description of work done is mentioned under the following headings:

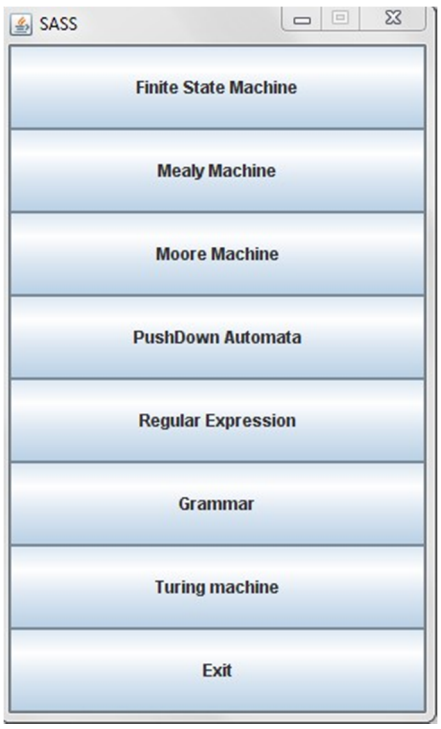


Fig 9: Interface

5.1.1 Finite State Machine (FSM)

- This module simulates a general FSM provided by the user.
- Designed a Frame for simulating Finite State Machine consisting of a Toolkit and a drawing area
- The toolkit contains tools in the form of buttons for designing various parts and attributes of the Finite State Machine like states, transitions etc.
- Users can select a tool from the toolkit and draw it on the drawing area accordingly e.g. When a user selects state, he can create a state on the drawing area wherever he clicks.
- For drawing transitions, user clicks on the start and the end state of the transition and provide the necessary inputs through the dialog boxes.
- After the machine is drawn, user can provide the input through “run” button and the simulator will traverse through the FSM according to the transitions, display FSM at each step and the final result (Accept/ Reject).

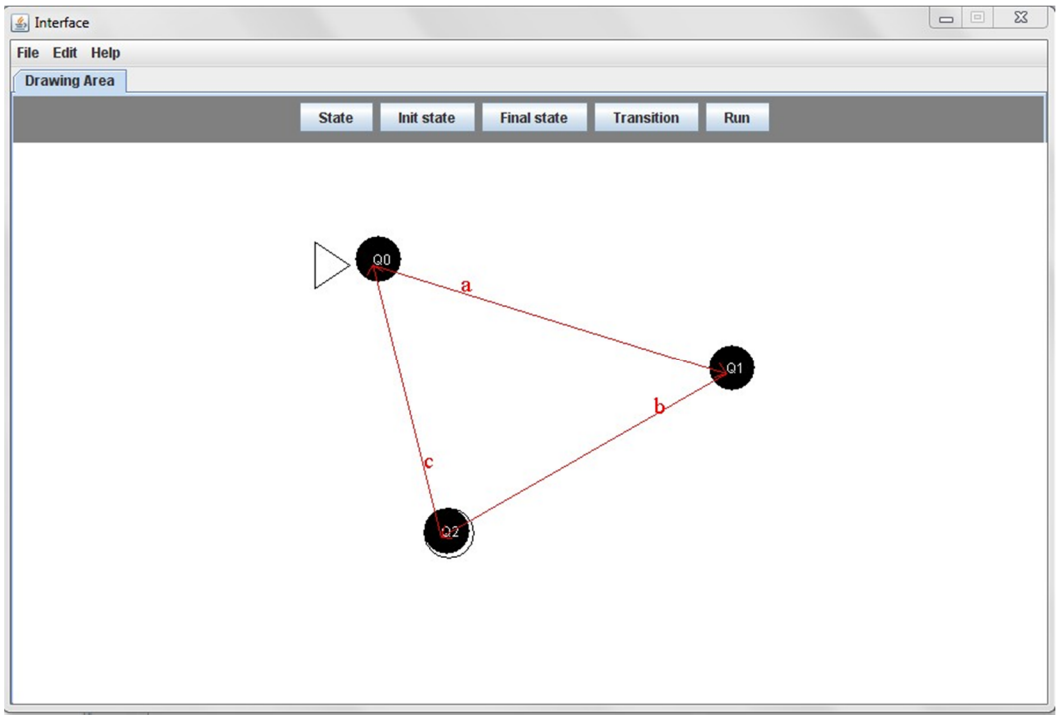


Fig 10: Finite state machine

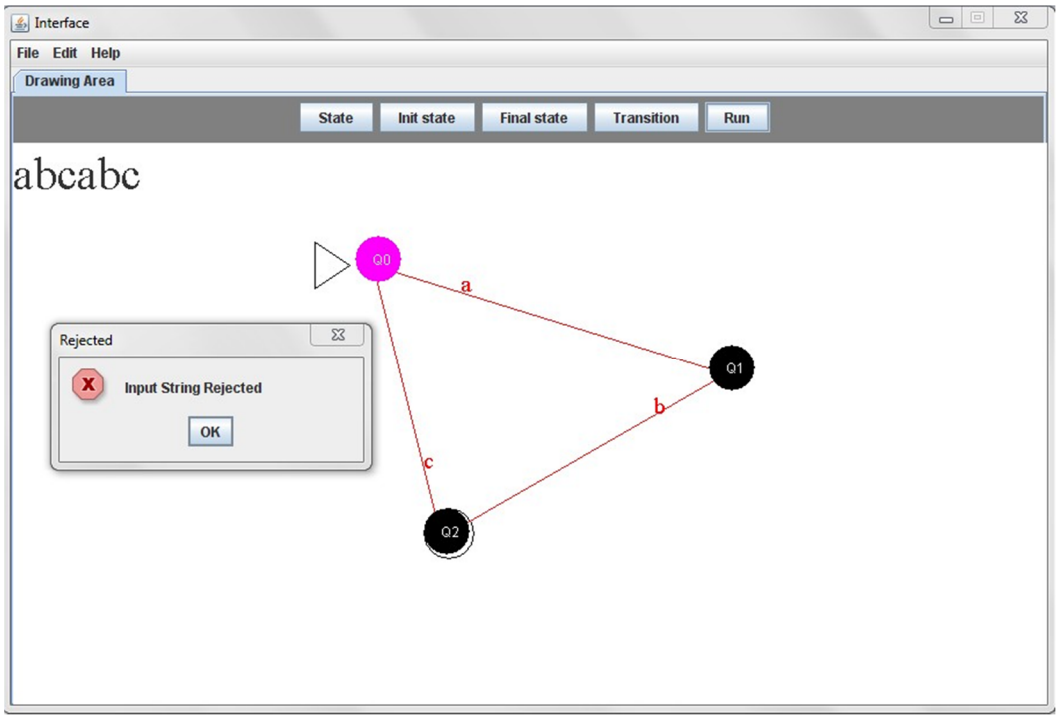


Fig 11: Finite state machine showing simulation

Algorithm:

1. Fetch the initial state from the state array and set it as current state and the first symbol of the input string and set it as current input.
2. Search for the corresponding transition in the transition array for this combination of current state and symbol.

3. Read the transition and modify the current state accordingly and read the next input symbol.
4. Repeat step (2) and (3) till the end of input string.
5. At the end, if the current state is final state, Accept the string else reject it.

5.1.2 Mealy machine

- Mealy Machine, a special type of FSM, is simulated in this module.
- Designed a Frame for simulating Mealy Machine consisting of a Toolkit and a drawing area
- The toolkit contains tools in the form of buttons for designing various parts and attributes of the Mealy Machine like states, transitions etc.
- Users can select a tool from the toolkit and draw it on the drawing area accordingly e.g. When a user selects state, he can create a state on the drawing area wherever he clicks.
- For drawing transitions, user clicks on the start and the end state of the transition and provide the necessary inputs through the dialog boxes.
- After the machine is drawn, user can provide the input through "run" button and the simulator will traverse through the mealy machine according to the transitions, display Machine at each step and generate the output.

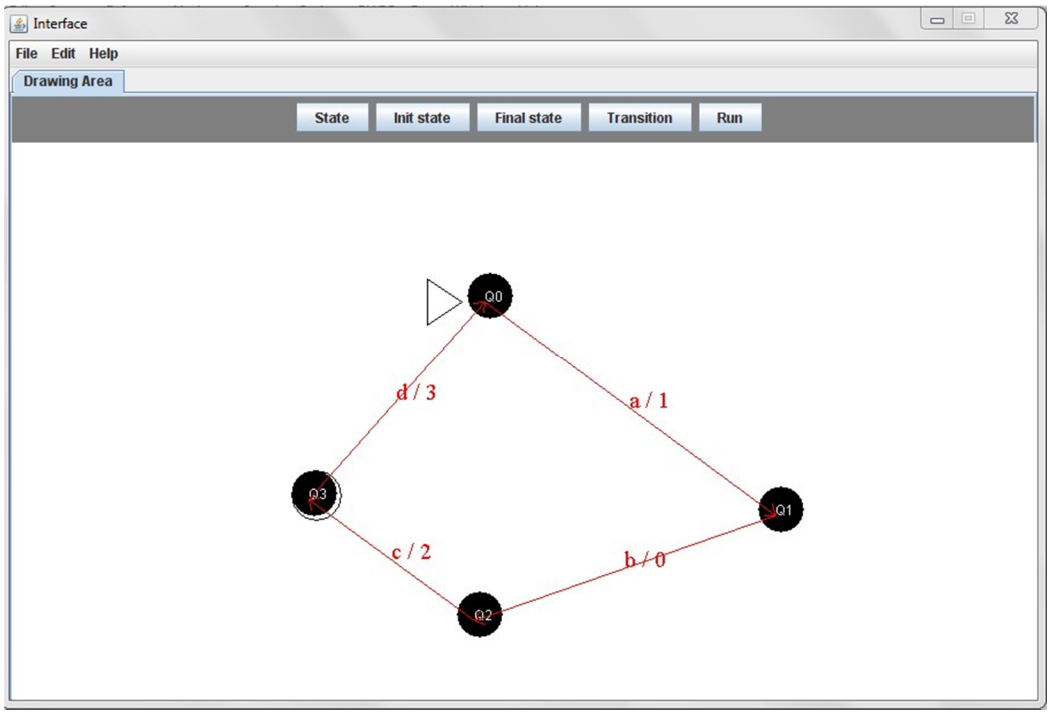


Fig 12: Mealy machine construction

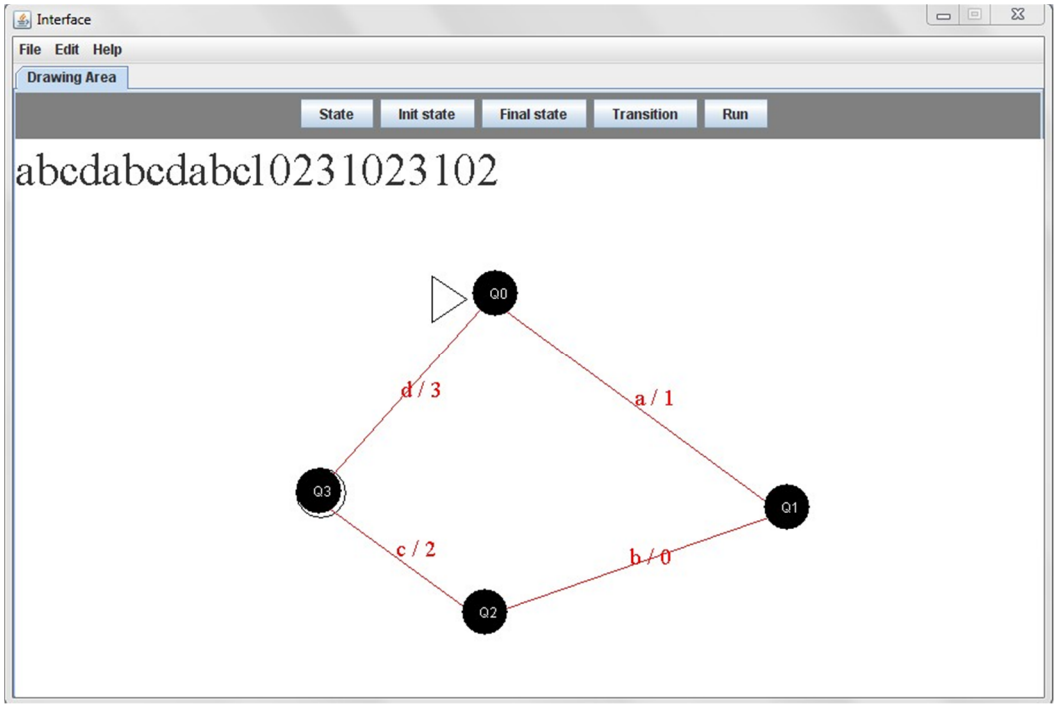


Fig 13: Mealy machine simulation

Algorithm:

1. Fetch the initial state from the state array and set it as current state and the first symbol of the input string and set it as current input.
2. Search for the corresponding transition in the transition array for this combination of initial state and input symbol.
3. Read the transition and modify the current state accordingly and concatenate the output symbol of the transition to the output string and read the next input symbol.
4. Repeat step (2) and (3) till the end of input string.
5. At the end, Display the output.

5.1.3 Moore machine

- Moore Machine, a special type of FSM, is simulated in this module.
- Designed a Frame for simulating Moore Machine consisting of a Toolkit and a drawing area
- The toolkit contains tools in the form of buttons for designing various parts and attributes of the Moore Machine like states, transitions etc.
- Users can select a tool from the toolkit and draw it on the drawing area accordingly e.g. When a user selects state, he can create a state on the drawing area wherever he clicks and provide the output of the state.

- For drawing transitions, user clicks on the start and the end state of the transition and provide the necessary inputs through the dialog boxes.
- After the machine is drawn, user can provide the input through “run” button and the simulator will traverse through the Moore machine according to the transitions, display Machine at each step and generate the output after each step.

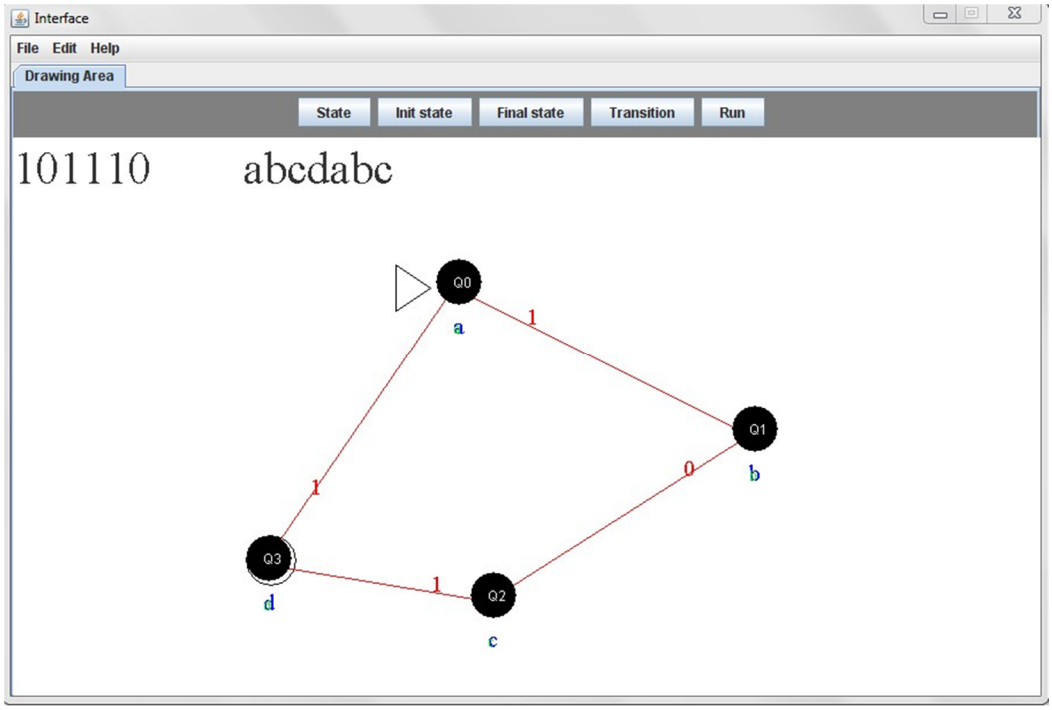


Fig 14: Moore machine construction

Algorithm:

1. Fetch the initial state from the state array and set it as current state and set the first symbol of the input string as the current input.
2. Concatenate the output symbol of this state in the output string.
3. Search for the corresponding transition in the transition array for this combination of current state and input symbol.
4. Read the transition and modify the current state accordingly and read the next input symbol.
5. Repeat step (2), (3) and (4) till the end of input string.
6. At the end, Display the output.

5.1.4 Pushdown automaton (PDA)

- The simulator for PDA contains a Toolkit and a drawing area which contains a stack with the marker symbol ‘$’.

- The toolkit contains tools in the form of buttons for designing various parts and attributes of the state diagram of the PDA like states, transitions etc.
- Users can select a tool from the toolkit and draw it on the drawing area accordingly e.g. When a user selects state, he can create a state on the drawing area wherever he clicks.
- For drawing transitions, user clicks on the start and the end state of the transition and provide the necessary inputs through the dialog boxes i.e. in case of PDA input symbol, top of stack and the modified stack (push/pop).
- After the machine is drawn, user can provide the input through "run" button and the simulator will traverse through the state diagram according to the transitions, display Machine with its stack at each step and generate the final output (Accept/Reject).

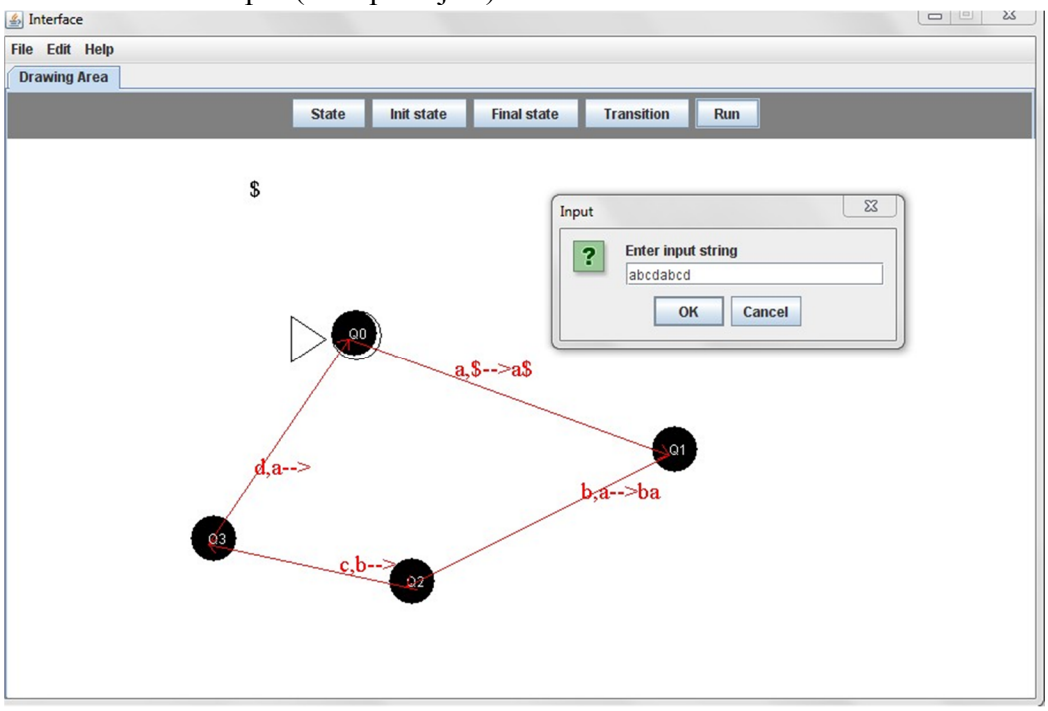


Fig 15: PDA construction

Algorithm:

1. Fetch the initial state from the state array and set it as current state , the first symbol of the input string and set it as current input and the top of the stack.
2. Search for the corresponding transition in the transition array for this combination of initial state, input symbol and the top of stack symbol.
3. Read the transition and modify the current state accordingly and perform the operation on stack (Push/pop) as mentioned in the transition and read the next input symbol.
4. Repeat step (2) and (3) till the end of input string.

At the end, if the stack is empty and the current state is final, Accept the input string else reject it.

5.1.5 Regular expression (to finite state automata)

- This module is a converter which converts a Regular expression provided by the user to its corresponding Finite State Automata.
- The simulator for Regular expression to FSA prompts the user for a Regular expression.
- The Regular expression can consist of the following 3 basic operations
    1. '*' Denoting the Kleene closure operation
    2. '.' Denoting the Concatenation operation
    3. '+' Denoting the Union operation
- After the user submits the expression, a window opens which contains the corresponding FSA.

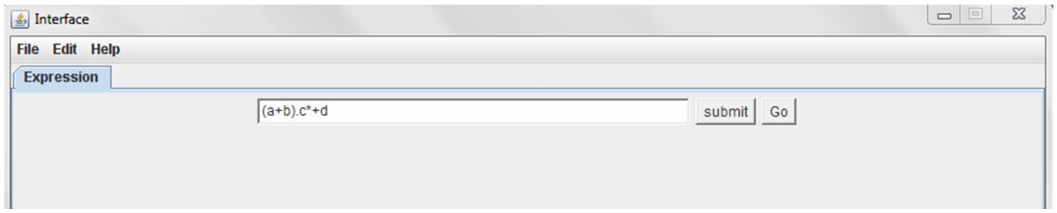


Fig 16: Input regular expression for NFA construction

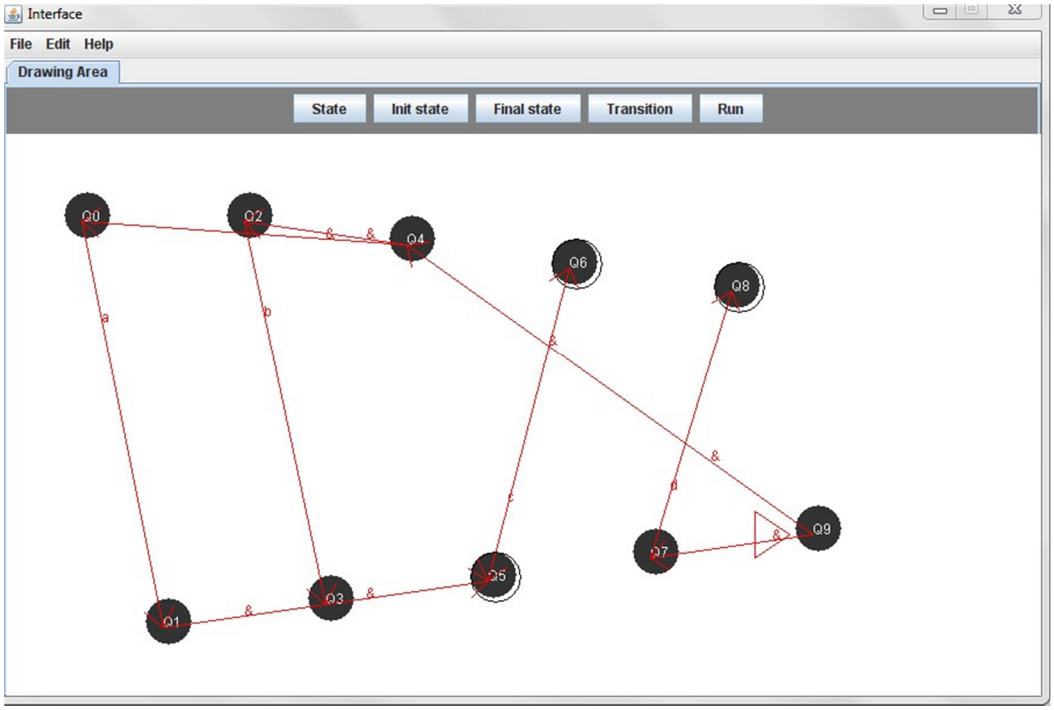


Fig 17: Output: NFA

Algorithm:

Thompson's Construction Algorithm (see Appendices)

5.1.6 Right linear grammar (to finite state automata)

- This module is a converter which converts a Right Linear Grammar provided by the user to its corresponding Finite State Automata.
- The simulator for Right Linear Grammar to FSA provides the user with an interface for inputting the productions of a grammar.
- The Productions consist of the following format
- L.H.S. → R.H.S.
- After the user submits the Grammar, a window opens which contains the corresponding FSA.

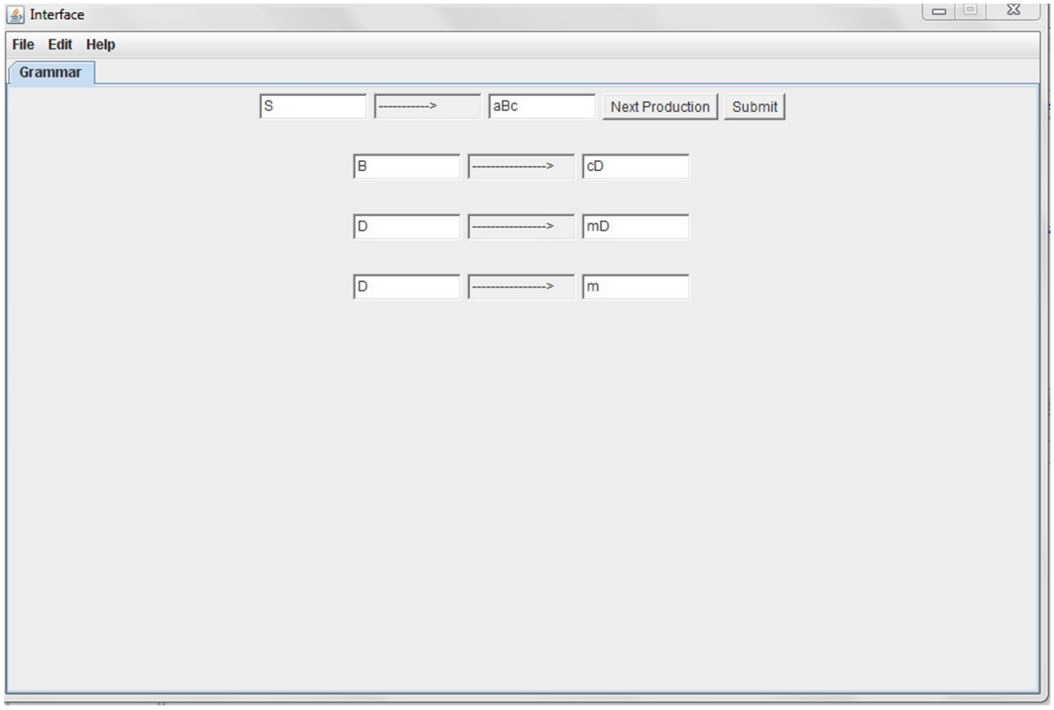


Fig 18: Regular grammar to finite automata

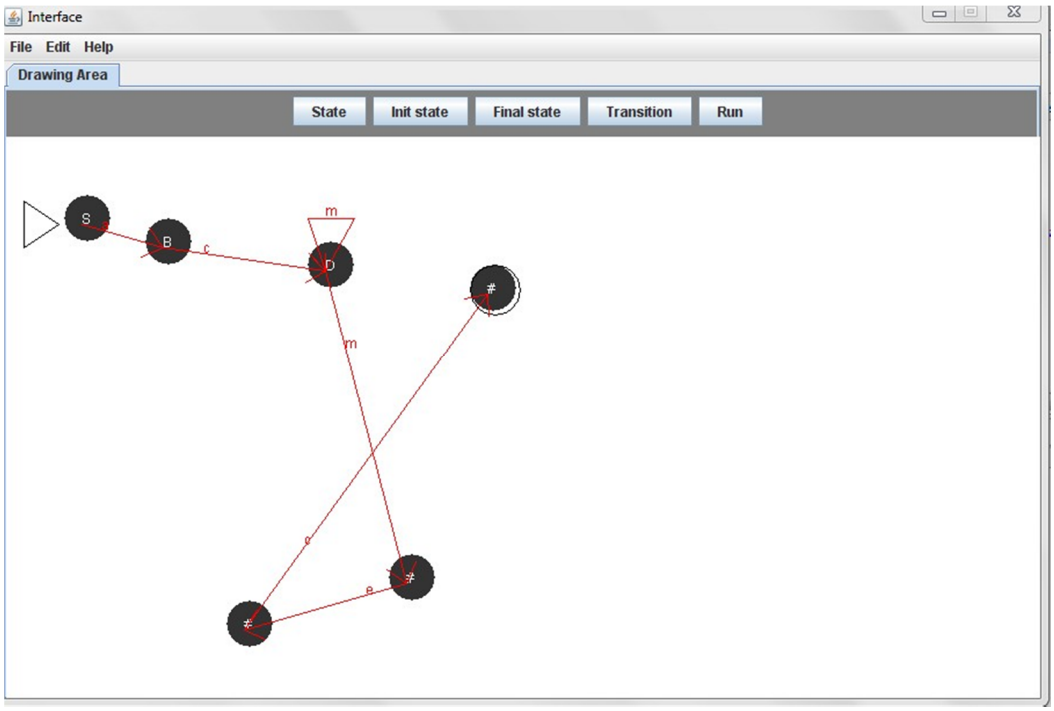


Fig 19: Output: finite automata

Algorithm:

1. Read the first production stored in the production array.
2. Create a new state and set its label as LHS of the production.
3. Traverse the RHS until a Variable is encountered, store this string of terminals and set the current variable as this variable.
4. If the state for this variable already exists, make a transition from the last state to this state with the temporary string as the transition symbol.
5. If the state does not exist, create a new state; make a transition from the last state to this state with the temporary string as the transition symbol. Traverse through production array; compare LHS with the current variable. Repeat this procedure until the end of the RHS.
6. Add the last terminal string of RHS to a final state.
7. Repeat this process recursively for all the productions of the start symbol.

5.1.7 Turing machine

- This module is used to simulate some very common Turing machines which are listed below
    1. Binary Adder
    2. Binary subtracter
    3. String copier
    4. Power 2 length String Accepter
    5. Palindrome
    6. Symbol delete
    7. Eraser
    8. Binary Increment
    9. Binary Decrement
    10. String with equal no of 0's and 1's

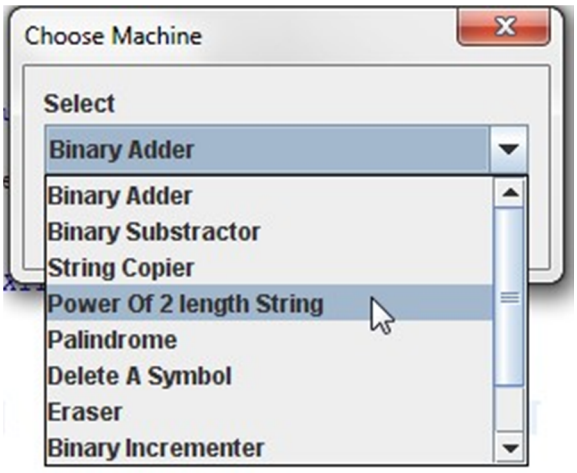


Fig 20: Machine chooser

- When user prompts for a Turing Machine, an option pane is opened which asks the user to select the input.
- After the user selects the Turing machine, he is directed to the Turing Area where appropriate input is asked.
- Then the step by step functioning of Turing machine over the provided input is displayed.

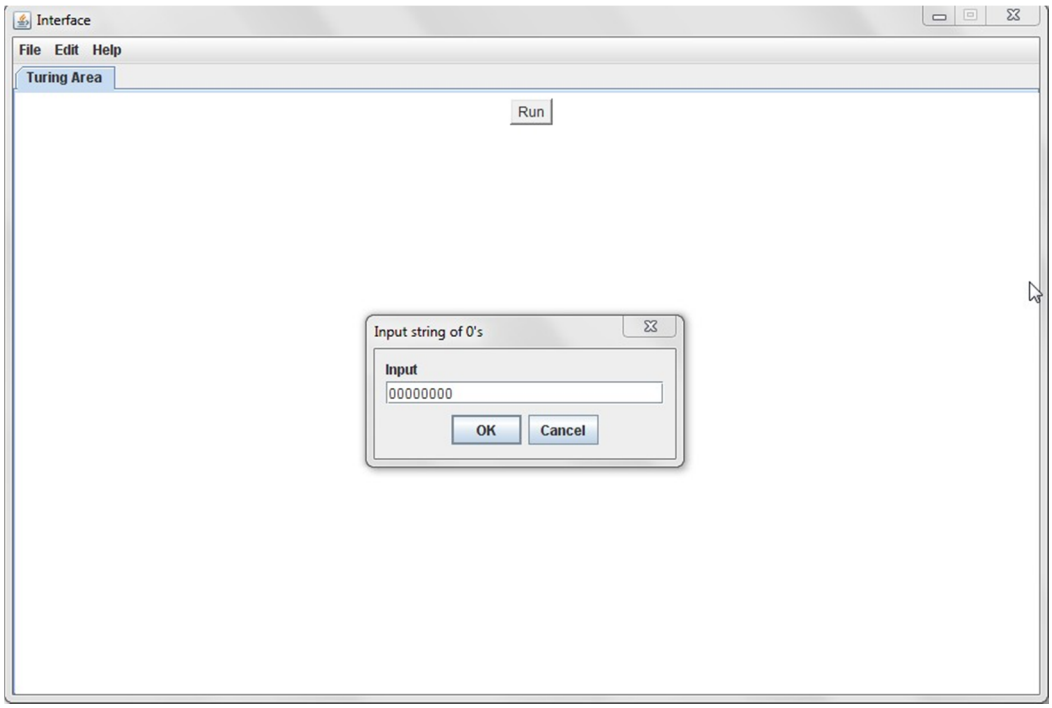


Fig 21: Input for simulation of 2^i Turing machine

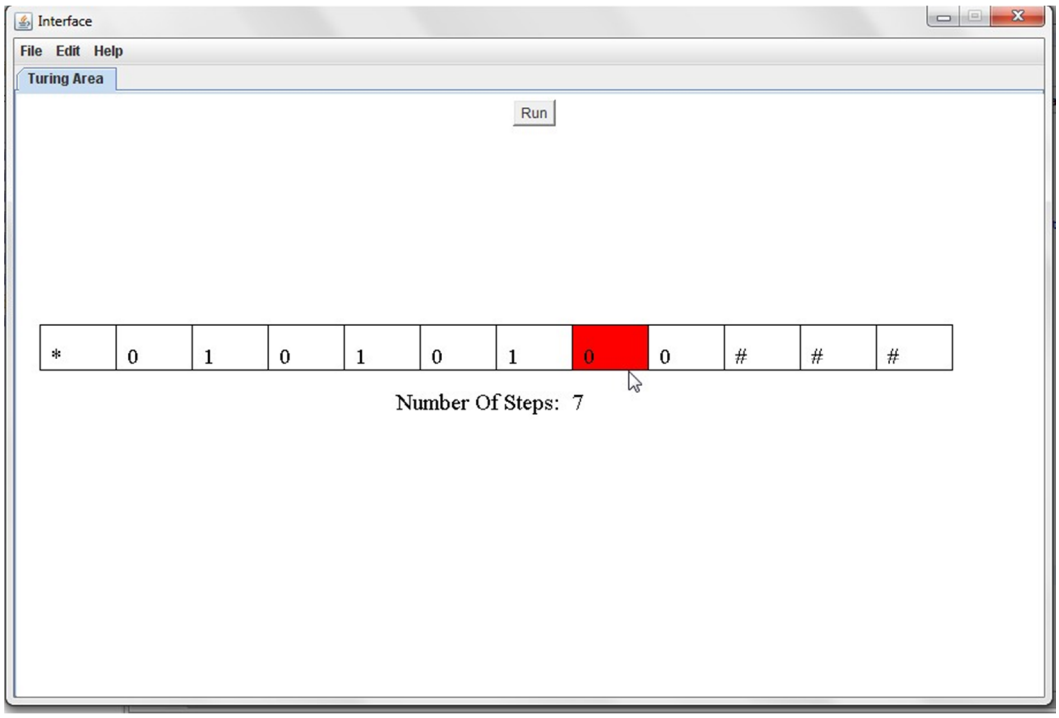


Fig 22: Simulation of 2^i Turing machine

Fig 23: Final output for 2^i Turing machine

Algorithm:

1. Write the Input string on the Tape of the Turing machine.
2. Fetch the initial state from the state array and set it as current state, the symbol at the tape head as the current symbol.
3. Search for the corresponding transition in the transition array for this combination of initial state and tape symbol
4. Read the transition and modify the current state accordingly, modify the tape block according to the transition, read the motion as described in the transition, move the tape head accordingly and read the next tape symbol.
5. Repeat step (3) and (4) till the end of input string.
6. At the end, if the current state is final, Accept the input string else reject it or show the output as per the Turing machine specification.

5.2 Work for 8th semester

- Implementation of Multi Tape Turing Machine

  A Multi-tape Turing machine is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank.

- Implementation of Pumping Lemma
  - For Regular Languages

    In the theory of formal languages, the pumping lemma for regular languages describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be *pumped* — that is, have a middle section of the word repeated an arbitrary number of times — to produce a new word that also lies within the same language.

    Specifically, the pumping lemma says that for any regular language L there exists a constant p such that any word w in L with length at least p can be split into three substrings, $w = xyz$, where the middle portion y must not be empty, such that the words xz, xyz, $xyyz$, $xyyyz$, … constructed by repeating y an arbitrary number of times (including zero times) are still in L. This process of repetition is known as "pumping". Moreover, the pumping lemma guarantees that the length of xy will be at most p, imposing a limit on the ways in which w may be split. Finite languages trivially satisfy the pumping lemma by having $p$equal to the maximum string length in L plus one.

  - For Context Free Languages

    The pumping lemma for context-free languages, also known as the Bar-Hillel lemma, is a lemma that gives a property shared by all context-free languages.

    If a language L is context-free, then there exists some integer $p \geq 1$ such that every string s in L with $|s| \geq p$ (where p is a "pumping length"[1]) can be written as

    $s = uvxyz$

    with substrings u, v, x, y and z, such that

    1. $|vxy| \leq p$,
    2. $|vy| \geq 1$, and
    3. $uv^n xy^n z$ is in L for all $n \geq 0$.

- Conversion of NFA to DFA

  Convert to an equivalent *deterministic* finite state machine (DFA) from a *non-deterministic* finite state machine (NFA)

- Minimization of DFA

  For each regular language that can be accepted by a DFA, there exists a minimal automaton, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names.)

  There are two classes of states that can be removed/merged from the original DFA without affecting the language it accepts to minimize it.

  Unreachable states are those states that are not reachable from the initial state of the DFA, for any input string.

  Non-distinguishable states are those that cannot be distinguished from one another for any input string.

- To implement a module that converts a finite state automata to Push Down Automata and Turing Machine which accepts the same set of languages

- Implementation of Universal Turing Machine Module

  Algorithm for UTM implementation:

  1. Step 1: Copy current state into buffer.
  2. Step 2: Copy scanned symbol into buffer.
  3. Step 3: Set X and Y markers.
  4. Step 4: Search for quintuple.
  5. Step 5: Erase buffer.
  6. Step 6: Write new symbol in tape description.
  7. Step 7: Shift tape marker.
  8. Step 8: Locate next state.

6 Appendices

6.1 Simulation of FSM

```
repaint();
inp1=ToolBox.inp;
        int inplength=inp1.length();
        int current=initialstate,next,j;
        next=-1;
        rejectit=false;

        while(inplength>0)
        {
                state[current].drawCurrent(inp1,inplength,"");

            for(j=0;j<transcount;j++)
            {
            if(trans[j].init==current &&
(trans[j].symbol).charAt(0)==inp1.charAt(inp1.length()-inplength))
                {
                        next=trans[j].fin;
                        break;
                }
            }

           if(j==transcount){rejectit=true;break;}
           nostaterepaint=true;
           repaint();
           nostaterepaint=false;
           try{Thread.sleep(1000);}catch(Exception e){}

            state[current].removecurrent(output);

            current=next;
            inplength--;
        }
        state[current].drawCurrent(inp1,inplength,"");
        repaint();
        if(state[current].isfinal && rejectit==false)
        {
                JOptionPane.showMessageDialog(new JFrame(),"Input String
Accepted","Accepted",JOptionPane.PLAIN_MESSAGE);
        //      repaintit();

        }
        else
        {
                JOptionPane.showMessageDialog(new JFrame(),"Input String
Rejected","Rejected",JOptionPane.ERROR_MESSAGE);
```

```
                //repaintit();
            }
            state[current].removecurrent(output);
            inp1="";
            output="";
            nostaterepaint=false;
            ToolBox.inp="";
            repaint();
```

6.2 Simulation of PDA

```
            repaint();
            st=new Stack();

            st.push(new Character('$'));
            stackarray=st.toString();
            repaint();
            inp1=ToolBox.inp;
            //inp1=inp1+'$';
            int inplength=inp1.length();
            int current=initialstate,next,j;
            next=-1;
            rejectit=false;

            while(inplength>0)
            {
                    state[current].drawCurrent(inp1,inplength,stackarray);
                System.out.println("My Transcount:"+transcount+" --"+st.peek()+ "
inplength="+inplength);
                for(j=0;j<transcount;j++)
                {
                System.out.println(trans[j].init  +  "   "+ current +  "   "+
(trans[j].symbol).charAt(0)+ "   "+ inp1.charAt(inp1.length()-inplength) +  "   "+ trans[j].out
+ "   "+  st.peek() );

if(trans[j].init==current && (trans[j].symbol).charAt(0)==inp1.charAt(inp1.length()-
inplength) && trans[j].out.equals((st.peek().toString())) )
                        {
                                next=trans[j].fin;
                                System.out.println("i am in");

                                st.pop();
                                stackarray=st.toString();
                                repaint();
                                    if(trans[j].top.length()>1)
                                            {
                                            st.push(new
Character((trans[j].top).charAt(1)));
                                            stackarray=st.toString();
                                            repaint();
```

```
                                        }
                                    if(trans[j].top.length()>0)
                                        {
                                        st.push(new
Character((trans[j].top).charAt(0)));
                                        stackarray=st.toString();
                                        repaint();
                                        }
                                break;
                            }
                        }

                        stackarray=st.toString();
                       System.out.println("stack content:"+stackarray.toString());
                        nostaterepaint=true;
                        System.out.println("yeah rapainting : "+stackarray);



                        try{Thread.sleep(1000);}catch(Exception e){}
                        System.out.println("repaint ke upar"+Interface.machine+ToolBox.choice);


                        if(j==transcount){rejectit=true;break;}
                        nostaterepaint=false;


                        state[current].removecurrent(output);

                        current=next;
                        inplength--;
                    }


                    repaint();
                    if(state[current].isfinal && rejectit==false &&
(st.peek().toString()).equals("$"))
                    {
                            JOptionPane.showMessageDialog(new JFrame(),"Input String
Accepted","Accepted",JOptionPane.PLAIN_MESSAGE);
                    //      repaintit();

                    }
                    else
                    {
                            JOptionPane.showMessageDialog(new JFrame(),"Input String
Rejected","Rejected",JOptionPane.ERROR_MESSAGE);

                            //repaintit();
                    }
```

```
state[current].removecurrent(output);
inp1="";
output="";
nostaterepaint=false;
//state[current].drawCurrent(inp1,inplength);
repaint();
```

6.3 Simulation of Turing machine

```
int i;
int currentstate=0;
int head=0,step=0;
boolean flag=true;
currentstate=initialstate;

Graphics g=this.getGraphics();
int width=(int)800/tape.length;
drawtape(g,width,tape,head,tape.length,step);
while(flag)
{
for(i=0;i<transcount;i++)
{

        if(trans[i].init==currentstate &&
trans[i].symbol.charAt(0)==tape[head])
        {
                currentstate=trans[i].fin;
                tape[head]=trans[i].out.charAt(0);
                if(trans[i].dir=='L'){head--;step++;}
                if(trans[i].dir=='R'){head++;step++;}
                try{Thread.sleep(200);}catch(Exception e){}
                drawtape(g,width,tape,head,tape.length,step);
        }


        if(state[currentstate].isaccepting==true ||
state[currentstate].isrejecting==true)flag=false;

}
}

Font f1;
f1=g.getFont();

g.setFont(new Font("Times New Roman",Font.PLAIN,25));
Interface.machine=19;

    if(state[currentstate].isaccepting==true){g.setColor(Color.green);g.drawString("String
Accepted",330,330);}
```

```
else
if(state[currentstate].isrejecting==true){g.setColor(Color.red);g.drawString("String
Rejected",330,330);}
g.setColor(Color.black);
```

6.4 Automata generator

```
int i,j,k;
int finstate=0;
boolean stflag;
String temp="";
int curstate=0;
int endst=0;


state[statecount++]=new State(x,y,'#',this,0,"");
state[statecount-1].isfinal=false;
//state[statecount-1].drawfinal();
endst=statecount-1;
if(statecount%2==0){y2-=20;y=y2;}
else
{y1+=20;y=y1;}

x+=70;


for(j=0;j<=nop;j++)
{
        curstate=sstate;
        if(prod[j][0].charAt(0)==symbol)
        {

                for(k=0;k<prod[j][1].length();k++)
                {
                        if(prod[j][1].charAt(k)>=65 &&
                        prod[j][1].charAt(k)<91)
                        {

                        stflag=true;
                        int k1;
                                for(k1=0;k1<statecount;k1++)
                                {if(state[k1].tag==prod[j][1].charAt(k))
                                {stflag=false;finstate=k1;break;}}

                                if(stflag==false)
                                        {
```

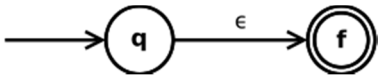
```
                                        trans[transcount++]= new
Transition(curstate,finstate,temp,"","");
                                        curstate=finstate;

                                        }
                                else
                                {
                                        state[statecount++]=new
State(x,y,prod[j][1].charAt(k),this,0,"");
                                        state[statecount-1].isfinal=false;
                                        if(statecount%2==0){y2-=20;y=y2;}
                                        else
                                        {y1+=20;y=y1;}

                                        x+=70;
                                        finstate=statecount-1;

                                        trans[transcount++]= new
Transition(curstate,finstate,temp,"","");
                                        curstate=genauto(finstate,state[k1].tag);
                                        temp="";
                                }

                        }
                        else
                        {
                                temp+=prod[j][1].charAt(k);
                        }
                }

                if(true)
                {

                        trans[transcount++]= new
Transition(curstate,endst,temp,"","");
                        curstate=endst;
                        temp="";
                }
        }
}


return endst;
```

6.5 Thompson's Construction Algorithm

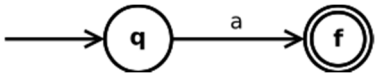
Thompson's Construction Algorithm (TCA) derives a nondeterministic finite automaton (NFA) from any regular expression by splitting it into its constituent subexpressions, from which the NFA will be constructed using a set of rules.

Rules:

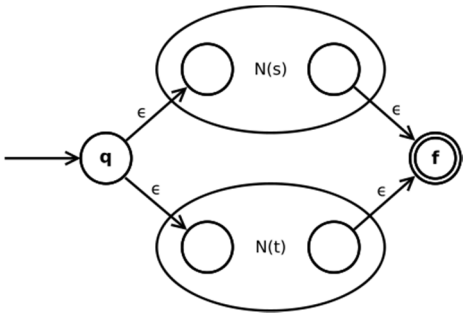
The **expression** ε is converted to



A **symbol** a **of the input alphabet** is converted to

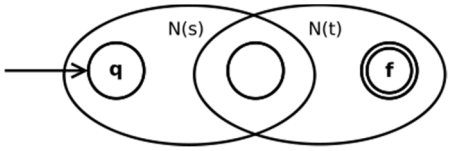


The **union expression** $s|t$ is converted to



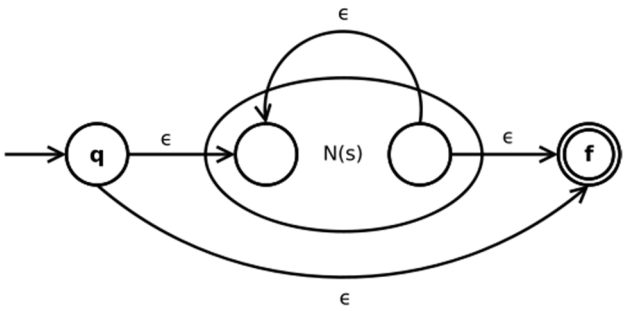
State q goes via ε either to the initial state of $N(s)$ or $N(t)$. Their final states become intermediate states of the whole NFA and merge via two ε-transitions into the final state of the NFA.

The **concatenation expression** st is converted to



The initial state of $N(s)$ is the initial state of the whole NFA. The final state of $N(s)$ becomes the initial state of $N(t)$. The final state of $N(t)$ is the final state of the whole NFA.

The **Kleene star expression** s^* is converted to



An ε-transition connects initial and final state of the NFA with the sub-NFA $N(s)$ in between. Another ε-transition from the inner final to the inner initial state of $N(s)$ allows for repetition of expression s according to the star operator.

The **parenthesized expression** (s) is converted to $N(s)$ itself.

7 References

- "**A taxonomy of finite automata construction algorithms"** by Bruce W. Watson , prof. Dr. J.C.M. Baeten, prof. Dr. M. Rem ; Department of Mathematics and Computing Science , Eindhoven University of Technology ; Netherlands , January 24, 1995.

- **"The Equivalent Conversion between Regular Grammar and Finite Automata"** by Jielan Zhang1, Zhongsheng Qian2 . 1Department of Information Technology, Yingtan Vocational and Technical College, Yingtan, China; 2School of Information Tech- nology, Jiangxi University of Finance and Economics, Nanchang, China.

- "**Introduction to Languages and The Theory of Computation**" by John C Martin

- **"Theory of Computer Science – Automata, Languages and Computation"** by K.L.P. Mishra and N. Chandrasekaran