Automata Theory And Computability

Subject Code: 15CS54 **I.A. Marks : 20**
Hours/Week : 04 **Exam Hours: 03**
Total Hours : 50 **Exam Marks: 80**

Module I 10 Hours

Why study the Theory of Computation, Languages and Strings:
Strings, Languages. A Language Hierarchy, Computation, Finite State Machines (FSM): Deterministic FSM, Regular languages, Designing FSM, Nondeterministic FSMs, From FSMs to Operational Systems, Simulators for FSMs, Minimizing FSMs, Canonical form of Regular languages, Finite State Transducers, Bidirectional Transducers.

MODULE – 2 10Hours

Regular Expressions (RE):
what is a RE?, Kleene's theorem, Applications of REs, Manipulating and Simplifying REs. Regular Grammars: Definition, Regular Grammars and Regular languages. Regular Languages (RL) and Nonregular Languages: How many RLs, To show that a language is regular, Closure properties of RLs, to show some languages are not RLs

MODULE – 3 10 Hours

Context-Free Grammars(CFG):
Introduction to Rewrite Systems and Grammars, CFGs and languages, designing CFGs, simplifying CFGs, proving that a Grammar is correct, Derivation and Parse trees, Ambiguity, Normal Forms. Pushdown Automata (PDA): Definition of non-deterministic PDA, Deterministic and Non-deterministic PDAs, Non-determinism and Halting, alternative equivalent definitions of a PDA, alternatives that are not equivalent to PDA.

MODULE – 4 10 Hours

Context-Free and Non-Context-Free Languages:
Where do the Context-Free Languages(CFL) fit, Showing a language is context-free, Pumping theorem for CFL, Important closure properties of CFLs, Deterministic CFLs. Algorithms and Decision Procedures for CFLs: Decidable questions, Un-decidable questions. Turing Machine: Turing machine model, Representation, Language acceptability by TM, design of TM, Techniques for TM construction

MODULE – 5 10 Hours

Variants of Turing Machines (TM), The model of Linear Bounded automata:
Decidability: Definition of an algorithm, decidability, decidable languages, 10 Hours Undecidable languages, halting problem of TM, Post correspondence problem. Complexity: Growth rate of functions, the classes of P and NP, Quantum Computation: quantum computers, Church-Turing thesis.

Text Books:

1. Elaine Rich: Automata, Computability and Complexity, 1st Edition, Pearson Education, 2012/13.

2. K.L.P. Mishra: Theory of Computer Science, Automata, Languages, and Computation, 3rd
Edition, PHI, 2012.

Reference Books:

1. John E Hopcroft, Rajeev Motwani, Jeffery D Ullman, Introduction to AutomataTheory, Languages, and Computation, 3rd Edition, Pearson Education, 2013.

2. Michael Sipser : Introduction to the Theory of Computation, 3rd edition, Cengage learning,2013

3. John C Martin, Introduction to Languages and The Theory of Computation, 3rd Edition, Tata McGraw –Hill Publishing Company Limited, 2013 .

4. Peter Linz, "An Introduction to Formal Languages and Automata", 3rd Edition, Narosa Publishers, 1998.

5. Basavaraj S. Anami, Karibasappa K G, Formal Languages and Automata theory, Wiley India, 2012 6. C K Nagpal, Formal Languages and Automata Theory, Oxford University press, 2012.

Table Of Contents

AUTOMATA THEORY AND COMPUTABILITY

Module I

Why study the Theory of Computation, Languages and Strings:

1.1: Introduction to finite Automata

1.2 : Central concepts of automata theory

1.3: Deterministic finite state machine

1.4:Non deterministic finite state machine

1.1:Introduction to finite automata

In this chapter we are going to study a class of machines called finite automata. Finite automata are computing devices that accept/recognize regular languages and are used to model operations of many systems we find in practice. Their operations can be simulated by a very simple computer program. A kind of systems finite automnata can model and a computer program to simulate their operations are discussed.

Formal definition

Automaton

An **automaton** is represented formally by a 5-tuple **(Q,Σ,δ,q_0,F)**, where:

- Q is a finite set of *states*.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ is the **transition function**, that is, $\delta: Q \times \Sigma \rightarrow Q$.
- q_0 is the *start state*, that is, the state of the automaton before any input has been processed, where $q_0 \in Q$.
- F is a set of states of Q (i.e. $F \subseteq Q$) called **accept states**.

Input word

An automaton reads a finite string of symbols $a_1,a_2,\ldots, a_n$, where $a_i \in \Sigma$, which is called an *input word*. The set of all words is denoted by Σ^*.

Run

A *run* of the automaton on an input word $w = a_1,a_2,\ldots, a_n \in \Sigma^*$, is a sequence of states $q_0,q_1,q_2,\ldots, q_n$, where $q_i \in Q$ such that q_0 is the start state and $q_i = \delta(q_{i-1},a_i)$ for $0 < i \leq n$. In words, at first the automaton is at the start state q_0, and then the automaton reads symbols of the input word in sequence. When the automaton reads symbol a_i it jumps to state $q_i = \delta(q_{i-1},a_i)$. q_n is said to be the *final state* of the run.

Accepting word

A word $w \in \Sigma^*$ is accepted by the automaton if $q_n \in F$.

Recognized language

An automaton can recognize a formal language. The language $L \subseteq \Sigma^*$ recognized by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages

The recognizable languages are the set of languages that are recognized by some automaton. For the above definition of automata the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

1.2:concepts of automata theory

Automata theory is a subject matter that studies properties of various types of automata. For example, the following questions are studied about a given type of automata.

- Which class of formal languages is recognizable by some type of automata? (Recognizable languages)
- Are certain automata *closed* under union, intersection, or complementation of formal languages? (Closure properties)
- How much is a type of automata expressive in terms of recognizing class of formal languages? And, their relative expressive power? (Language Hierarchy)

Automata theory also studies if there exist any effective algorithm or not to solve problems similar to the following list.

- Does an automaton accept any input word? (emptiness checking)
- Is it possible to transform a given non-deterministic automaton into deterministic automaton without changing the recognizable language? (Determinization)
- For a given formal language, what is the smallest automaton that recognizes it? (Minimization).

Classes of automata

The following is an incomplete list of types of automata.

Automata	Recognizable language
Deterministic finite state machine(DFSM)	regular languages
Nondeterministic finite state machine(NDFSM)	regular languages
Nondeterministic finite state machine with ε-transitions (FND-ε or ε-NDFSM)	regular languages
Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	recursively enumerable languages
Timed automata	
Deterministic Büchi automata	ω-limit languages
Nondeterministic Büchi automata	ω-regular languages
Nondeterministic/Deterministic Rabin automata	ω-regular languages
Nondeterministic/Deterministic Streett automata	ω-regular languages
Nondeterministic/Deterministic parity automata	ω-regular languages
Nondeterministic/Deterministic Muller automata	ω-regular languages

.1.3:Deterministic finite state machine

. **Definition**: A DFSM is 5-tuple or quintuple $M = (Q, \sum, \delta, q_0, A)$ where

Q is non-empty, finite set of states.

$\sum$ is non-empty, finite set of input alphabets.

δ is transition function, which is a mapping from $Q \times \sum$ to Q.

q_0 □ Q is the start state.

A □ Q is set of accepting or final states.

Note: For each input symbol *a*, from a given state there is exactly one transition (there can be no transitions from a state also) and we are sure (or can determine) to which state the machine enters. So, the machine is called **Deterministic** machine. Since it has finite number of states the machine is called Deterministic finite machine or Deterministic Finite Automaton or Finite State Machine (FSM).

The language accepted by DFSM is

L(M) = { w | w □ $\sum$* and $\delta^*(q_0, w)$ □ A }

The non-acceptance of the string *w* by an FA or DFSM can be defined in formal notation as:

L(M) = { w | w □ $\sum$* and $\delta^*(q_0, w)$ □ A }

Obtain a DFSM to accept strings of a's and b's starting with the string ab

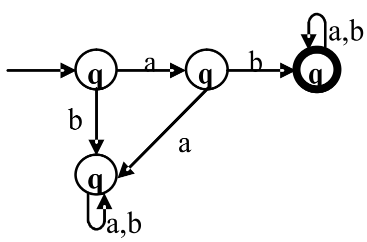
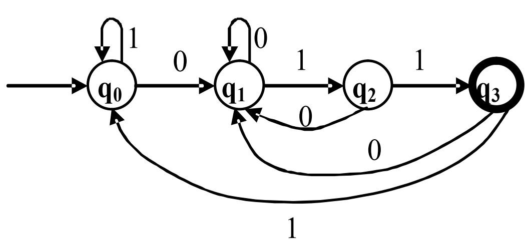


Fig.1.1 Transition diagram to accept string ab(a+b)*

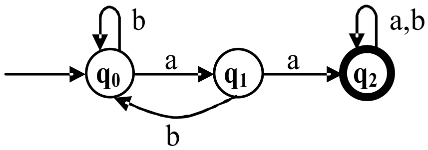
So, the DFSM which accepts strings of a's and b's starting with the string *ab* is given by
M = (Q, $\sum$, δ, q_0, A) where

Q = {q_0, q_1, q_2, q_3}
$\sum$ = {a, b}
q_0 is the start state
A = {q_2}.
δ is shown the transition table 2.4.

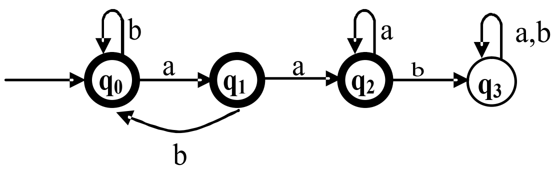
Draw a DFSM to accept string of 0's and 1's ending with the string 011.



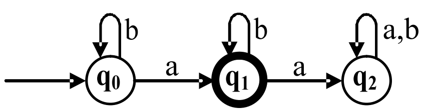
Obtain a DFSM to accept strings of a's and b's having a sub string aa

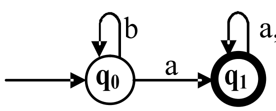


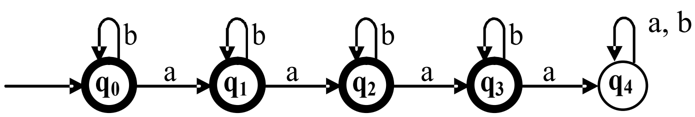
Obtain a DFSM to accept strings of a's and b's except those containing the substring aab.



Obtain DFSMs to accept strings of a's and b's having exactly one a,







Obtain a DFSM to accept strings of a's and b's having even number of a's and b's

The machine to accept even number of a's and b's is shown in fig.2.22.

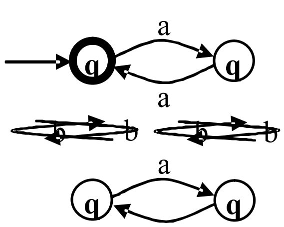
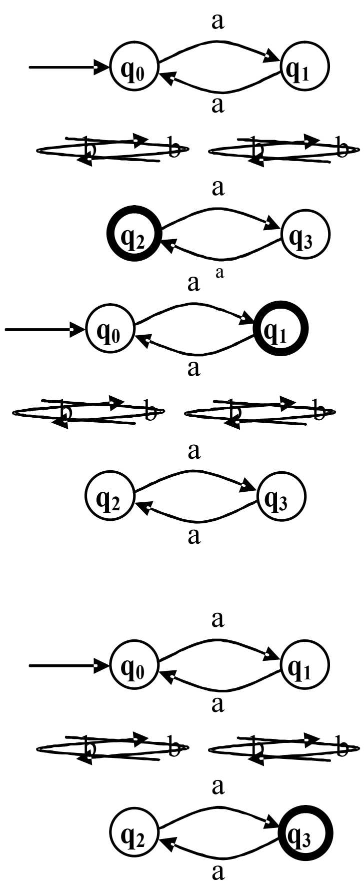


Fig.2.22 DFSM to accept even no. of a's and b's



Regular language

Definition: Let M = (Q, $\sum$, δ, q_0, A) be a DFSM. The language L is regular if there exists a
machine M such that L = L(M).

*** Applications of Finite Automata ***

String matching/processing

Compiler Construction

The various compilers such as C/C++, Pascal, Fortran or any other compiler is designed using the finite automata. The DFSMs are extensively used in the building the various phases
of compiler such as

- Lexical analysis (To identify the tokens, identifiers, to strip of the comments etc.)
- Syntax analysis (To check the syntax of each statement or control statement used in the program)
- Code optimization (To remove the un wanted code)
- Code generation (To generate the machine code)

Other applications- The concept of finite automata is used in wide applications. It is not possible to list all the applications as there are infinite number of applications. This section lists some applications:

1. Large natural vocabularies can be described using finite automaton which includes the applications such as spelling checkers and advisers, multi-language dictionaries, to indent the documents, in calculators to evaluate complex expressions based on the priority of an operator etc. to name a few. Any editor that we use uses finite automaton for implementation.
2. Finite automaton is very useful in recognizing difficult problems i.e., sometimes it is very essential to solve an un-decidable problem. Even though there is no general solution exists for the specified problem, using theory of computation, we can find the approximate solutions.
3. Finite automaton is very useful in hardware design such as circuit verification, in design of the hardware board (mother board or any other hardware unit), automatic traffic signals, radio controlled toys, elevators, automatic sensors, remote sensing or controller etc.

In game theory and games wherein we use some control characters to fight against a monster, economics, computer graphics, linguistics etc., finite automaton plays a very important role

1.4 : Non deterministic finite state machine(NDFSM)

Definition: An NDFSM is a 5-tuple or quintuple $M = (Q, \sum, \delta, q_0, A)$ where

Q is non empty, finite set of states.

$\sum$ is non empty, finite set of input alphabets.

δ is transition function which is a mapping from

$Q \times \{\sum \cup \varepsilon\}$ to subsets of 2^Q. This function shows

the change of state from one state to a set of states based on the input symbol.

q_0 □ Q is the start state.

A □ Q is set of final states.

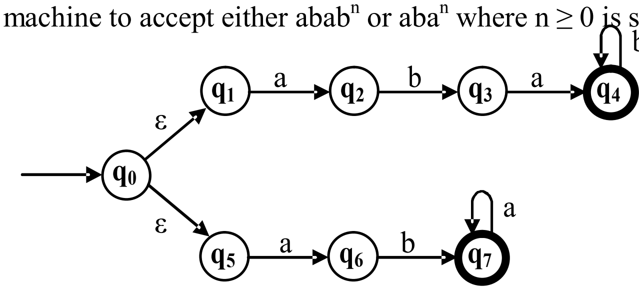
Acceptance of language

Definition: Let M = (Q, ∑, δ, q_0, A) be a DFSM where Q is set of finite states, ∑ is set of input
alphabets (from which a string can be formed), δ is transition function from Q x {∑Uε} to 2^Q, q_0 is the start state and A is the final or accepting state. The string (also called language) *w* accepted by an NDFSM can be defined in formal notation as:

L(M) = { w | w □ ∑*and δ*(q_0, w) = Q with atleast one
Component of Q in A}

Obtain an NDFSM to accept the following language L = {w | w □ $abab^n$ or aba^n where $n \geq 0$}

The machine to accept either $abab^n$ or aba^n where $n \geq 0$ is shown below:



Conversion from NDFSM to DFSM

Let $M_N = (Q_N, \sum_N, \delta_N, q_0, A_N)$ be an NDFSM and accepts the language $L(M_N)$. There should be
an equivalent DFSM $M_D = (Q_D, \sum_D, \delta_D, q_0, A_D)$ such that $L(M_D) = L(M_N)$. The procedure to convert an NDFSM to its equivalent DFSM is shown below:

Step1:

The start state of NDFSM M_N is the start state of DFSM M_D. So, add q_0(which is the start
state of NDFSM) to Q_D and find the transitions from this state. The way to obtain
different transitions is shown in step2.

Step2:

For each state $[q_i, q_j, \ldots q_k]$ in Q_D, the transitions for each input symbol in ∑ can be obtained as shown below:

1. $\delta_D([q_i, q_j, \ldots q_k], a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \ldots \delta_N(q_k, a)$
   $= [q_l, q_m, \ldots q_n]$ say.
2. Add the state $[q_l, q_m, \ldots q_n]$ to Q_D, if it is not already in Q_D.
3. Add the transition from $[q_i, q_j, \ldots q_k]$ to $[q_l, q_m, \ldots q_n]$ on the input symbol *a* iff the state $[q_l, q_m, \ldots q_n]$ is added to Q_D in the previous step.

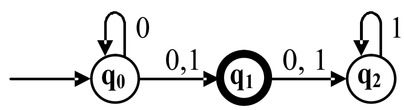
Step3:

The state $[q_a, q_b, \ldots q_c] \in Q_D$ is the final state, if at least one of the state in $q_a, q_b, \ldots.. q_c \in A_N$ i.e., at least one of the component in $[q_a, q_b, \ldots q_c]$ should be the final state of NDFSM.

Step4:

If epsilon (ϵ) is accepted by NDFSM, then start state q_0 of DFSM is made the final state.

Convert the following NDFSM into an equivalent DFSM.



Step1: q_0 is the start of DFSM (see step1 in the conversion procedure).

So, $Q_D = \{[q_0]\}$ (2.7)

Step2: Find the new states from each state in Q_D and obtain the corresponding transitions.

Consider the state $[q_0]$:

When $a = 0$

$\delta_D([q_0], 0) = \delta_N([q_0], 0)$
$= [q_0, q_1]$
(2.8)

When $a = 1$

$\delta_D([q_0], 1) = \delta_N([q_0], 1)$
$= [q_1]$
(2.9)

Since the states obtained in (2.8) and (2.9) are not in Q_D(2.7), add these two states to Q_D so that

$Q_D = \{[q_0], [q_0, q_1], [q_1] \}$ (2.10)

The corresponding transitions on $a = 0$ and $a = 1$ are shown below.

Σ

δ	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_0, q_1]$		
$[q_1]$		

Q

Consider the state $[q_0, q_1]$:

When $a = 0$

$\delta_D([q_0, q_1], = \delta_N([q_0, q_1], 0)$

0) $= \delta_N(q_0, 0) \cup \delta_N(q_1, 0)$
$= \{q_0, q_1\} \cup \{q_2\}$
$= [q_0, q_1, q_2]$
(2.11)

When $a = 1$

$\delta_D([q_0, q_1], 1) = \delta_N([q_0, q_1], 1)$
$= \delta_N(q_0, 1) \cup \delta_N(q_1, 1)$
$= \{q_1\} \cup \{q_2\}$
$= [q_1, q_2]$
(2.12)

Since the states obtained in (2.11) and (2.12) are the not defined in Q_D(see 2.10), add these two states to Q_D so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2]\} \qquad (2.13)$$

and add the transitions on $a = 0$ and $a = 1$ as shown below:

Σ

δ	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$[q_1]$		
$[q_0, q_1, q_2]$		

Q

Consider the state $[q_1]$:

When $a = 0$

$\delta_D([q_1], 0) = \delta_N([q_1], 0)$
$= [q_2]$
(2.14)

When $a = 1$

$\delta_D([q_1], 1) = \delta_N([q_1], 1) = [q_2] \qquad (2.15)$

Since the states obtained in (2.14) and (2.15) are same and the state q_2 is not in Q_D(see 2.13), add the state q_2 to Q_D so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2], [q_2]\} \qquad (2.16)$$

and add the transitions on $a = 0$ and $a = 1$ as shown below:

Σ

$[q_0]$	0	1
$[q_0, q_1]$	$[q_0, q_1]$	$[q_1]$
$[q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_2]$	$[q_2]$
$[q_1, q_2]$		

Consider the state $[q_0, q_1, q_2]$:

Q

When $a = 0$

$$\begin{aligned}\delta_D([q_0,q_1,q_2], 0) &= \delta_N([q_0,q_1,q_2], 0)\\ &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0)\\ &= \{q_0,q_1\} \cup \{q_2\} \cup \{\varphi\}\\ &= [q_0,q_1,q_2] \quad (2.17)\end{aligned}$$

When $a = 1$

$$\begin{aligned}\delta_D([q_0,q_1,q_2], 1) &= \delta_N([q_0,q_1,q_2], 1)\\ &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1)\\ &= \{q_1\} \cup \{q_2\} \cup \{q_2\}\\ &= [q_1, q_2] \quad (2.18)\end{aligned}$$

Since the states obtained in (2.17) and (2.18) are not new states (are already in Q_D, see 2.16), do not add these two states to Q_D. But, the transitions on $a = 0$ and $a = 1$ should be added to the transitional table as shown below:

$\longleftarrow \Sigma \longrightarrow$

δ	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$[q_1]$	$[q_2]$	$[q_2]$
$[q_0, q_1, q_2]$	$[q_0,q_1,q_2]$	$[q_1, q_2]$
$[q_1, q_2]$		

Q

Consider the state $[q_1,q_2]$:

When $a = 0$

$$\begin{aligned}\delta_D([q_1,q_2], 0) &= \delta_N([q_1,q_2], 0)\\ &= \delta_N(q_1, 0) \cup \delta_N(q_2, 0)\\ &= \{q_2\} \cup \{\varphi\}\\ &= [q_2] \quad (2.19)\end{aligned}$$

When $a = 1$

$$\begin{aligned}\delta_D([q_1,q_2], 1) &= \delta_N([q_1,q_2], 1)\\ &= \delta_N(q_1, 1) \cup \delta_N(q_2, 1)\\ &= \{q_2\} \cup \{q_2\}\\ &= [q_2] \quad (2.20)\end{aligned}$$

Since the states obtained in (2.19) and (2.20) are not new states (are already in Q_D see 2.16), do not add these two states to Q_D. But, the transitions on a = 0 and a = 1 should be added to the transitional table as shown below:

Σ

δ	0	1
[q_0]	[q_0, q_1]	[q_1]
[q_0, q_1]	[q_0, q_1, q_2]	[q_1, q_2]
[q_1]	[q_2]	[q_2]
[q_0, q_1, q_2]	[q_0,q_1,q_2]	[q_1, q_2]
[q_1, q_2]	[q_2]	[q_2]

Consider the state [q_2]:

Q

When a = 0

δ_D([q_2], 0) = δ_N([q_2], 0)
= {φ} (2.21)

When a = 1

δ_D([q_2], 1) = δ_N([q_2], 1)
= [q_2] (2.22)

Since the states obtained in (2.21) and (2.22) are not new states (are already in Q_D, see 2.16), do not add these two states to Q_D. But, the transitions on a = 0 and a = 1 should be added to the transitional table. The final transitional table is shown in table 2.14. and final DFSM is shown in figure 2.35.

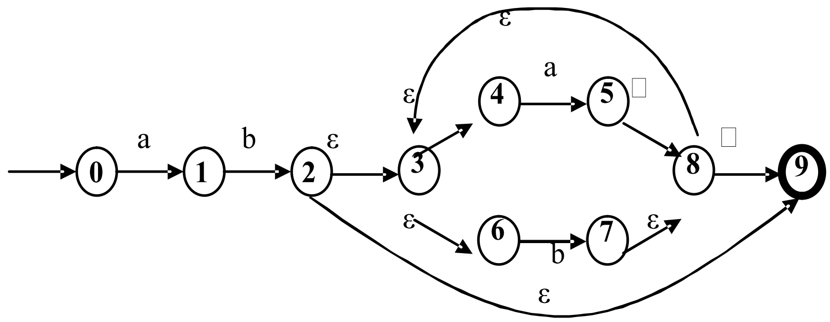
δ	0	1
[q_0]	[q_0, q_1]	[q_1]
	[q_0, q_1, q_2]	[q_1, q_2]
	[q_2]	[q_2]
	[q_0,q_1,q_2]	[q_1, q_2]
	[q_2]	[q_2]
[q_2]	φ	[q_2]

0

0 1

0

Convert the following NDFSM to its equivalent DFSM.



Let $Q_D = \{0\}$ (A)

Consider the state [A]:

When input is *a*:

$\delta(A, a) = \delta_N(0, a)$
$= \{1\}$
(B)

When input is *b*:

$\delta(A, b) = \delta_N(0, b)$
$= \{\varphi\}$

Consider the state [B]:

When input is *a*:

$\delta(B, a) = \delta_N(1, a)$
$= \{\varphi\}$

When input is *b*:

δ(B, b) = δ_N(1, b)
= {2}
= {2,3,4,6,9} (C)

This is because, in state 2, due to ε-transitions (or without giving any input) there can be transition to states 3,4,6,9 also. So, all these states are reachable from state 2. Therefore,

δ(B, b) = {2,3,4,6,9} = C

Consider the state [C]:

When input is *a*:

δ(C, a) = δ_N({2,3,4,6,9}, a)
= {5}
= {5, 8, 9, 3, 4, 6}
= {3, 4, 5, 6, 8, 9} (ascending order) (D)

This is because, in state 5 due to ε-transitions, the states reachable are {8, 9, 3, 4, 6}. Therefore,

δ(C, a) = {3, 4, 5, 6, 8, 9} = D

When input is *b*:

δ(C, b) = δ_N({2, 3, 4, 6, 9}, b)
= {7}
= {7, 8, 9, 3, 4, 6}
= {3, 4, 6, 7, 8, 9}(ascending order) (E)

This is because, from state 7 the states that are reachable without any input (i.e., ε-transition) are {8, 9, 3, 4, 6}. Therefore,

δ(C, b) = {3, 4, 6, 7, 8, 9} = E

Consider the state [D]:

When input is *a*:

δ(D, a) = δ_N({3,4,5,6,8,9}, a)
= {5}
= {5, 8, 9, 3, 4, 6}
= {3, 4, 5, 6, 8, 9} (ascending order) (D)

When input is *b*:

δ(D, b) = δ_N({3,4,5,6,8,9}, b)
= {7}
= {7, 8, 9, 3, 4, 6}
= {3, 4, 6, 7, 8, 9} (ascending

order) (E)

Consider the state [E]:

When input is *a*:

$\delta(E, a) = \delta_N(\{3,4,6,7,8,9\}, a)$
$= \{5\}$
$= \{5, 8, 9, 3, 4, 6\}$
$= \{3, 4, 5, 6, 8, 9\}$(ascending order)
(D)

When input is *b*:

$\delta(E, b) = \delta_N(\{3,4,6,7,8,9\}, b)$
$= \{7\}$
$= \{7, 8, 9, 3, 4, 6\}$
$= \{3, 4, 6, 7, 8, 9\}$(ascending order)
(E)

Since there are no new states, we can stop at this point and the transition table for the DFSM is
shown in table 2.15.

Σ

δ	a	b
A	B	-
B	-	C
	D	E
	D	E
	D	E

Table 2.15 Transitional table

The states C, D and E are final states, since 9 (final state of NDFSM) is present in
C, D and E. The final transition diagram of DFSM shown in figure 2.36
is

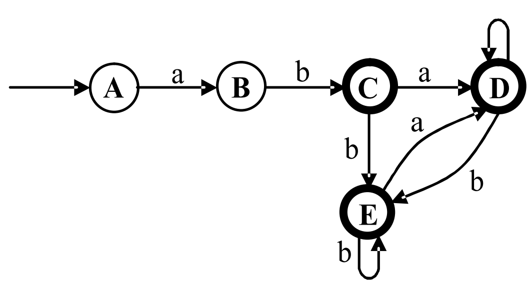


Fig. 2.36 The DFSM

MODULE-2:

REGULAR EXPRESSIONS

2.1 An application of finite automata

2.2 Finite automata with Epsilon transitions

2.3 Regular expressions

2.4 Finite automata and regular expressions

2.5 Applications of Regular expressions

2.6 Regular languages

2.7 proving languages not to be regular languages

2.8 closure properties of regular languages

2.9 decision properties of regular languages

2.10 equivalence and minimization of automata

2.1 An application of finite automata

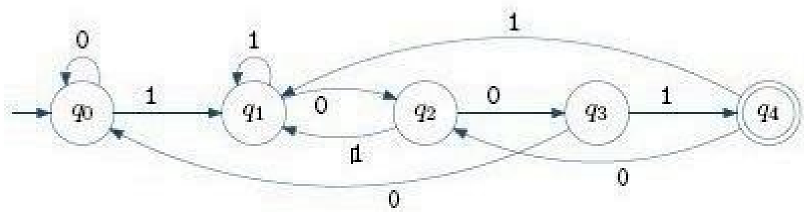
Applications of finite automata includes String matching algorithms, network protocols and lexical analyzers

String Processing

Consider finding all occurrences of a short string (*pattern string*) within a Long string (*text string*).This can be done by processing the text through a DFSM: the DFSM for all strings that *end* with the pattern string. Each time the accept state is reached, the current position in the text is output

Example: Finding 1001

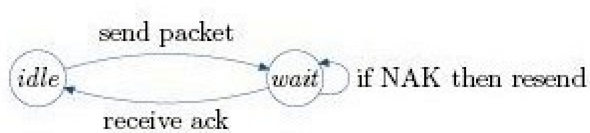
To find all occurrences of pattern 1001, construct the DFSM for all strings ending in **1001.**



Finite-State Machines

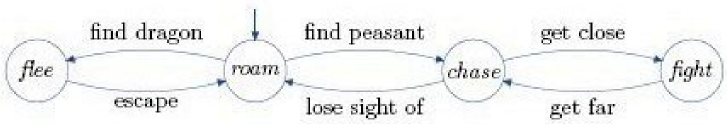
A *finite-state machine* is an FA together with actions on the arcs.

A trivial example for a communication link:



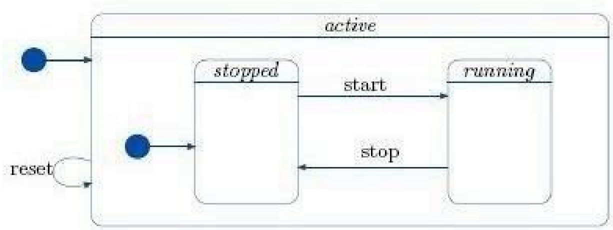
Example FSM: Bot Behavior

A *bot* is a computer-generated character in a video game.



State charts

State charts model tasks as a set of states and actions. They extend FA diagrams. Here is a simplified state chart for a stopwatch



Lexical Analysis

In compiling a program, the first step is *lexi-cal analysis*. This isolates keywords,identifiersetc., while eliminating irrelevant symbols.A *token* is a category, for example "identifier","relation operator" or specific keyword.

For example,

token RE

keyword then then

variable name [a-zA-Z][a-zA-Z0-9]* where latter RE says it is any string of alphanumeric **characters starting with a letter.**

A lexical analyzer takes source code as a string,and outputs sequence of *tokens*.

For example,

for i = 1 to max do

x[i] = 0;

might have token sequence

for id = num to id do id [id] = num sep

As a token is identified, there may be an action.

For example, when a number is identified, itsvalue is calculated

2.2 Finite automata with Epsilon transitions

We can extend an NDFSM by introducing a "feature" that allows us to make a transition on , the empty string. All the transition lets us do is spontaneously make a transition, without receiving an input symbol. This is another mechanism that allows our NDFSM to be in multiple states at once. Whenever we take an edge, we must fork off a new "thread" for the NDFSM starting in the destination state.
Just as nondeterminism made NDFSM's more convenient to represent some problems than DFSM's but were not more powerful, the same applies to εNDFSM's. While more expressive, anything we can represent with an εNDFSM we can represent with a DFSM that has no ε transitions.

Epsilon Closure

Epsilon Closure of a state is simply the set of all states we can reach by following the transition function from the given state that are labeled . Generally speaking, a collection of objects is closed under some operation if applying that operation to members of the collection returns an object still in the collection.
In the above example:
ε (q) = { q }
ε (r) = { r, s}
let us define the extended transition function for an εNDFSM. For a regular, NDFSM we said for the induction step:
Let
δ^(q,w) = $\{p_1, p_2, ... p_k\}$
$\delta(p_i,a) = S_i$ for i=1,2,...k
Then ^(q, wa) = $S_1,S_2... S_k$
For an -NDFSM, we change for ^(q, wa):
Union[δ (Each state in $S_1, S_2, ... S_k$)]
This includes the original set $S_1,S_2... S_k$ as well as any states we can reach via .
When coupled with the basis that ^(q,) = δ (q) lets us inductively define an extended transition function for a εNDFSM.

Eliminating εTransitions

εTransitions are a convenience in some cases, but do not increase the power of the NDFSM. To eliminate them we can convert a εNDFSM into an equivalent DFSM, which is quite similar to the steps we took for converting a normal NDFSM to a DFSM, except we must now follow all εTransitions and add those to our set of states.

1. Compute ε for the current state, resulting in a set of states S.
2. δ(S,a) is computed for all a in ∑ by
    a. Let S = $\{p_1, p_2, ... p_k\}$
    b. Compute $I=1 \rightarrow k$ (p_i,a) and call this set $\{r_1, r_2, r_3... r_m\}$. This set is achieved by following input a,
    not by following any ε transitions
    c. Add the ε transitions in by computing (S,a)= $I=1 \rightarrow m$ ε(r_1)
3. Make a state an accepting state if it includes any final states in the -NDFSM.

Note : The ε (epsilon) transition refers to a transition from one state to another without the reading of an input symbol (ie without the tape containing the input string moving). Epsilon transitions can be inserted between any states. There is also a conversion algorithm from a NDFSM with epsilon transitions to a NDFSM without epsilon transitions.

δ	a	b	C	ε
q0	{q0}	φ	φ	{q1}
q1	φ	{q2}	φ	{q2}
q2	φ	φ	{q2}	φ

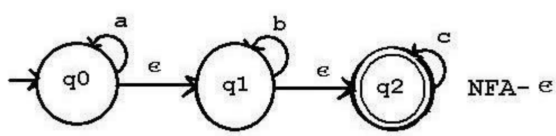
Consider the NDFSM-epsilon move machine M = { Q, ∑, δ, q0, F}

Q = { q0, q1, q2 }

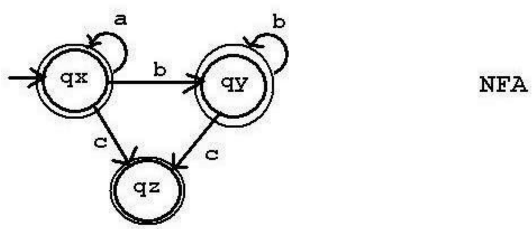
∑= { a, b, c } and ε moves

q0 = q0

F = { q2 }



regular expression $a^* b^* c^*$



Note: add an arc from qz to qz labeled "c" to figure above.

The language accepted by the above NDFSM with epsilon move the set of strings over {a,b,c} including the null string and all strings with any number of a's followed by any number of b's followed by any number of c's.

Now convert the NDFSM with epsilon moves to a NDFSM M = (Q', ∑, δ', q0', F')

First determine the states of the new machine, Q' = the epsilon closure of the states in the NDFSM with epsilon moves. There will be the same number of states but the names can be constructed by writing the state name as the set of states in the epsilon closure. The epsilon closure is the initial state and all states that can be reached by one or more epsilon moves.

Thus q0 in the NDFSM-epsilon becomes {q0,q1,q2} because the machine can move from q0 to q1 by an epsilon move, then check q1 and find that it can move from q1 to q2 by an epsilon move.

q1 in the NDFSM-epsilon becomes {q1,q2} because the machine can move from q1 to q2 by an epsilon move.

q2 in the NDFSM-epsilon becomes {q2} just to keep the notation the same. q2 can go nowhere except q2, that is what phi means, on an epsilon move.
We do not show the epsilon transition of a state to itself here, but, beware, we will take into account the state to itself epsilon transition when converting NDFSM's to regular expressions.

The initial state of our new machine is {q0,q1,q2} the epsilon closure of q0

The final state(s) of our new machine is the new state(s) that contain a state symbol that was a final state in the original machine.

The new machine accepts the same language as the old machine, thus same sigma.

So far we have for out new NDFSM
Q' = { {q0,q1,q2}, {q1,q2}, {q2} } or renamed { qx, qy, qz }
∑= { a, b, c }
F' = { {q0,q1,q2}, {q1,q2}, {q2} } or renamed { qx, qy, qz }
q0 = {q0,q1,q2} or renamed qx

inputs

δ′	a	b	c
qx or{q0,q1,q2}			
qy or{q1,q2}			
qz or{q2}			

Now we fill in the transitions. Remember that a NDFSM has transition entries that are sets.
Further, the names in the transition entry sets must be only the state names from Q'.
Very carefully consider each old machine transitions in the first row.
You can ignore any φ entries and ignore the ε column.
In the old machine δ(q0,a)=q0 thus in the new machine
δ'({q0,q1,q2},a)={q0,q1,q2} this is just because the new machine accepts the same language as the old machine and must at least have the the same transitions for the new state names.

inputs

δ′	a	b	c
qx or{q0,q1,q2}	{qx} or{{q0,q1,q2}}		
qy or{q1,q2}			
qz or{q2}			

No more entries go under input a in the first row because old δ(q1,a)=φ, δ(q2,a)=φ

Now consider the input b in the first row, δ(q0,b)=φ, δ(q1,b)={q2} and δ(q2,b)=φ. The reason we considered q0, q1 and q2 in the old machine was because out new state has symbols q0, q1 and q2 in the new state name from the epsilon closure. Since q1 is in {q0,q1,q2} and δ(q1,b)=q1 then δ'({q0,q1,q2},b)={q1,q2}. WHY {q1,q2} ?, because {q1,q2} is the new machines name for the old machines name q1. Just compare the zeroth column of δ to δ'. So we have

inputs

δ′	a	b	c
qx or{q0,q1,q2}	{qx} or{{q0,q1,q2}}	{qy} or{{q1,q2}}	
qy or{q1,q2}			
qz or{q2}			

Now, because our new qx state has a symbol q2 in its name and δ(q2,c)=q2 is in the old machine, the new name for the old q2, which is qz or {q2} is put into the input c transition in row 1.

Inputs

δ′	a	b	c
qx or{q0,q1,q2}	{qx} or{{q0,q1,q2}}	{qy} or{{q1,q2}}	{qz} or{{q2}}
qy or{q1,q2}			
qz or{q2}			

Now, tediously, move on to row two, ...
You are considering all transitions in the old machine, delta, for all old machine state symbols in the name of the new machines states. Fine the old machine state that results from an input and translate the old machine state to the corresponding new machine state name and put the new machine state name in the set in delta'. Below are the "long new state names" and the renamed state names in delta'.

Inputs

δ′	a	b	c
qx or{q0,q1,q2}	{qx} or{{q0,q1,q2}}	{qy} or{{q1,q2}}	{qz} or{{q2}}
qy or{q1,q2}	φ	{qy} or{{q1,q2}}	{qz} or{{q2}}
qz or{q2}	φ	φ	{qz} or{{q2}}

The figure above labeled NDFSM shows this state transition table.

It seems rather trivial to add the column for epsilon transitions,
but we will make good use of this in converting regular expressions
to machines. regular-expression -> NDFSM-epsilon -> NDFSM -> DFSM.

2.3 :Regular expression

Definition: A regular expression is recursively defined as follows.

1. φ is a regular expression denoting an empty language.
2. ε-(epsilon) is a regular expression indicates the language containing an empty string.
3. *a* is a regular expression which indicates the language containing only {a}
4. If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S, then
    a. R+S is a regular expression corresponding to the language $L_R U L_S$.
    b. R.S is a regular expression corresponding to the language $L_R.L_S$.
    c. R* is a regular expression corresponding to the language L_R^*.
5. The expressions obtained by applying any of the rules from 1-4 are regular expressions.

The table 3.1 shows some examples of regular expressions and the language corresponding to these regular expressions.

Regular expressions	Meaning
(a+b)*	Set of strings of a's and b's of any length including the NULL string.
(a+b)*abb	Set of strings of a's and b's ending with the string abb
ab(a+b)*	Set of strings of a's and b's starting with the string ab.
(a+b)*aa(a+b)*	Set of strings of a's and b's having a sub string aa.
a*b*c*	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include

	empty string).
$a^+b^+c^+$	Set of string consisting of at least one ‘a’ followed by string consisting of at least one ‘b’ followed by string consisting of at least one ‘c’.
aa*bb*cc*	Set of string consisting of at least one ‘a’ followed by string consisting of at least one ‘b’ followed by string consisting of at least one ‘c’.
(a+b)* (a + bb)	Set of strings of a’s and b’s ending with either *a* or *bb*
(aa)*(bb)*b	Set of strings consisting of even number of a’s followed by odd number of b’s
(0+1)*000	Set of strings of 0’s and 1’s ending with three consecutive zeros(or ending with 000)
(11)*	Set consisting of even number of 1’s

Table 3.1 Meaning of regular expressions

Obtain a regular expression to accept a language consisting of strings of a’s and b’s of even length.

String of a’s and b’s of even length can be obtained by the combination of the strings aa, ab, ba and bb. The language may even consist of an empty string denoted by ε. So, the regular expression can be of the form

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

The * closure includes the empty string.
Note: This regular expression can also be represented using set notation as

$$L(R) = \{(aa + ab + ba + bb)^n \mid n \geq 0\}$$

Obtain a regular expression to accept a language consisting of strings of a’s and b’s of odd length.

String of a’s and b’s of odd length can be obtained by the combination of the strings aa, ab, ba and bb followed by either *a* or *b*. So, the regular expression can be of the form

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

String of a’s and b’s of odd length can also be obtained by the combination of the strings aa, ab, ba and bb preceded by either *a* or *b*. So, the regular expression can also be represented as

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

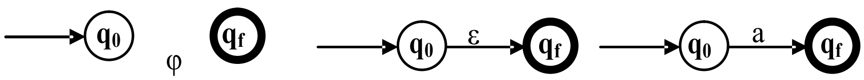
Note: Even though these two expression are seems to be different, the language corresponding to those two expression is same. So, a variety of regular expressions can be obtained for a language and all are equivalent.

2.4 :finite automata and regular expressions

Obtain NDFSM from the regular expression

Theorem: Let R be a regular expression. Then there exists a finite automaton $M = (Q, \sum, \delta, q_0, A)$ which accepts L(R).

Proof: By definition, φ, ε and *a* are regular expressions. So, the corresponding machines to recognize these expressions are shown in figure 3.1.a, 3.1.b and 3.1.c respectively.



(a) (b) (c)

Fig 3.1 NDFSMs to accept φ, ε and a

The schematic representation of a regular expression R to accept the language L(R) is shown in figure 3.2. where q is the start state and f is the final state of machine M.

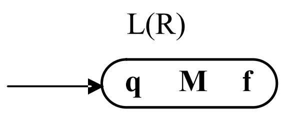


Fig 3.2 Schematic representation of FA accepting L(R)

In the definition of a regular expression it is clear that if R and S are regular expression, then R+S and R.S and R* are regular expressions which clearly uses three operators '+', '-' and '.'. Let us take each case separately and construct equivalent machine. Let $M_1 = (Q_1, \sum_1, \delta_1, q_1, f_1)$ be a machine which accepts the language $L(R_1)$ corresponding to the regular expression R_1. Let $M_2 = (Q_2, \sum_2, \delta_2, q_2, f_2)$ be a machine which accepts the language $L(R_2)$ corresponding to the regular expression R_2.

Case 1: $R = R_1 + R_2$. We can construct an NDFSM which accepts either $L(R_1)$ or $L(R_2)$ which can be represented as $L(R_1 + R_2)$ as shown in figure 3.3.

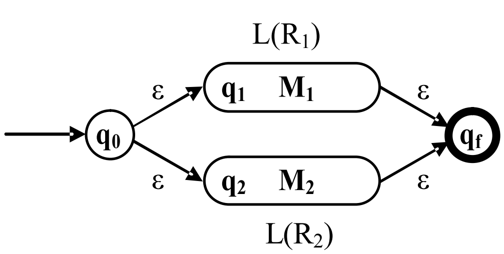


Fig. 3.3 To accept the language L(R1 + R2)

It is clear from figure 3.3 that the machine can either accept $L(R_1)$ or $L(R_2)$. Here, q_0 is the start state of the combined machine and q_f is the final state of combined machine M.

Case 2: $R = R_1 \,.\, R_2$. We can construct an NDFSM which accepts $L(R_1)$ followed by $L(R_2)$ which
can be represented as $L(R_1 \,.\, R_2)$ as shown in figure 3.4.

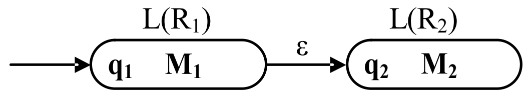
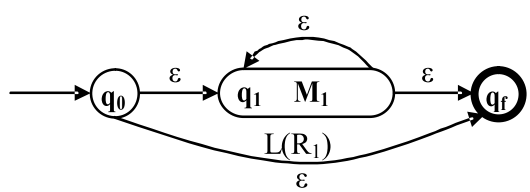


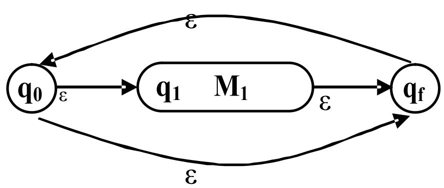
Fig. 3.4To accept the language L(R1 . R2)

It is clear from figure 3.4 that the machine after accepting $L(R_1)$ moves from state q_1 to f_1. Since there is a ε-transition, without any input there will be a transition from state f_1 to state q_2. In state q_2, upon accepting $L(R_2)$, the machine moves to f_2 which is the final state. Thus, q_1 which is the start state of machine M_1 becomes the start state of the combined machine M and f_2 which is the final state of machine M_2, becomes the final state of machine M and accepts the language $L(R_1.R_2)$.

Case 3: $R = (R_1)^*$. We can construct an NDFSM which accepts either $L(R_1)^*$) as shown in figure
3.5.a. It can also be represented as shown in figure 3.5.b.



(a)



(b)

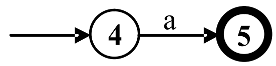
Fig. 3.5 To accept the language $L(R1)^*$

It is clear from figure 3.5 that the machine can either accept ε or any number of $L(R_1)$s thus accepting the language $L(R_1)^*$. Here, q_0 is the start state q_f is the final state.

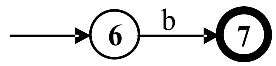
Obtain an NDFSM which accepts strings of a's and b's starting with the string ab.

The regular expression corresponding to this language is ab(a+b)*.

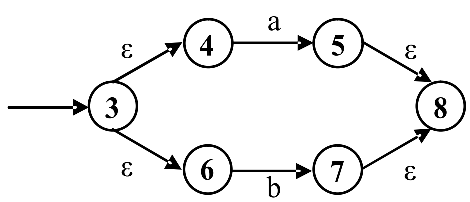
Step 1: The machine to accept 'a' is shown below.



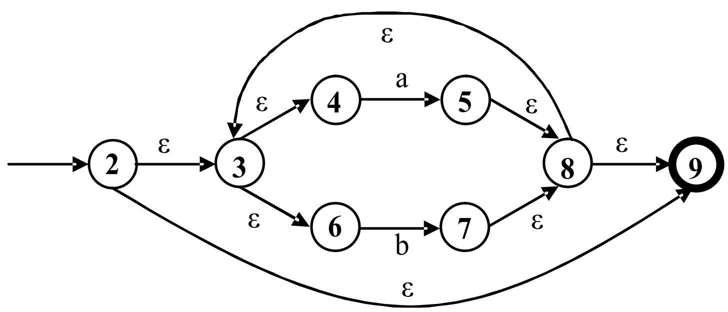
Step 2: The machine to accept 'b' is shown below.



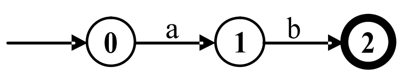
Step 3: The machine to accept (a + b) is shown below.



Step 4: The machine to accept (a+b)* is shown below.



Step 5: The machine to accept ab is shown below.



Step 6: The machine to accept ab(a+b)* is shown below.

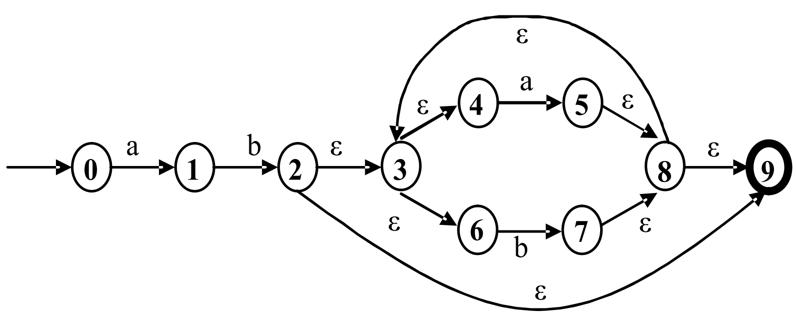


Fig. 3.6 To accept the language L(ab(a+b)*)

Obtain the regular expression from FA

Theorem: Let M = (Q, ∑, δ, q_0, A) be an FA recognizing the language L. Then there exists an equivalent regular expression R for the regular language L such that L = L(R).

The general procedure to obtain a regular expression from FA is shown below. Consider the generalized graph

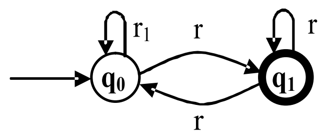


Fig. 3.9 Generalized transition graph

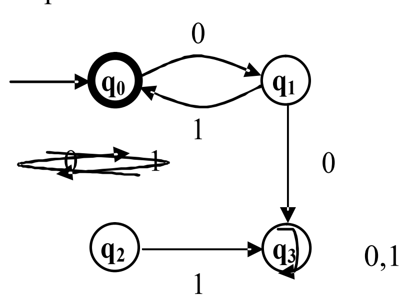
where r_1, r_2, r_3 and r_4 are the regular expressions and correspond to the labels for the edges. The regular expression for this can take the form:

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{3.1}$$

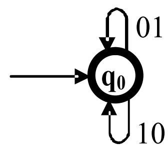
Note:

1. Any graph can be reduced to the graph shown in figure 3.9. Then substitute the regular expressions appropriately in the equation 3.1 and obtain the final regular expression.
2. If r_3 is not there in figure 3.9, the regular expression can be of the form
   $$r = r_1^* r_2 r_4^* \tag{3.2}$$
3. If q_0 and q_1 are the final states then the regular expression can be of the form
   $r = r_1^* + r_1^* r_2 r_4^*$ (3.3

Obtain a regular expression for the FA shown below:



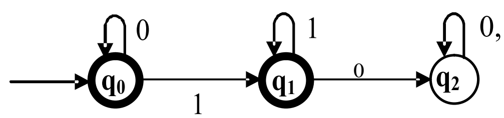
The figure can be reduced as shown below:



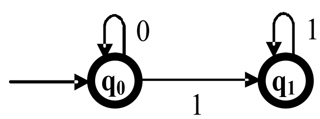
It is clear from this figure that the machine accepts strings of 01's and 10's of any length and the regular expression can be of the form

$$(01 + 10)^*$$

What is the language accepted by the following FA



Since, state q_2 is the dead state, it can be removed and the following FA is obtained.



The state q_0 is the final state and at this point it can accept any number of 0's which can be represented using notation as

$$0^*$$

q_1 is also the final state. So, to reach q_1 one can input any number of 0's followed by 1 and followed by any number of 1's and can be represented as

$$0^*11^*$$

So, the final regular expression is obtained by adding 0^* and 0^*11^*. So, the regular expression is

$$\begin{aligned} R.E &= 0^* + 0^*11^* \\ &= 0^*(\square + 11^*) \\ &= 0^*(\square + 1^+) \\ &= 0^*(1^*) = 0^*1^* \end{aligned}$$

It is clear from the regular expression that language consists of any number of 0's (possibly ε) followed by any number of 1's(possibly ε).

2.5:Applications of Regular Expressions

Pattern Matching refers to a set of objects with some common properties. We can match an identifier or a decimal number or we can search for a string in the text.

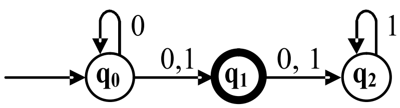
An application of regular expression in UNIX editor ed.
In UNIX operating system, we can use the editor *ed* to search for a specific pattern in the text. For example, if the command specified is

/acb*c/

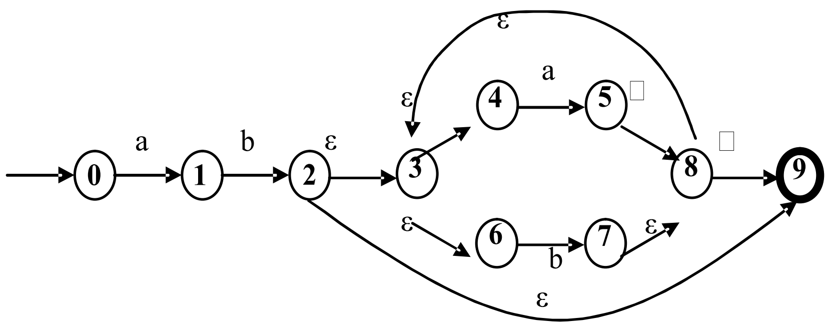
then the editor searches for a string which starts with *ac* followed by zero or more b's and followed by the symbol *c*. Note that the editor ed accepts the regular expression and searches for that particular pattern in the text. As the input can vary dynamically, it is challenging to write programs for string patters of these kinds.

Questions:

1. Obtain an NDFSM to accept the following language L = {w | w □ abab^n or aba^n where n $\geq$ 0}
2. Convert the following NDFSM into an equivalent DFSM.



3. Convert the following NDFSM to its equivalent DFSM.



4. P.T. Let R be a regular expression. Then there exists a finite automaton M = (Q, ∑, δ, q_0, A) which accepts L(R).

5. Obtain an NDFSM which accepts strings of a's and b's starting with the string ab.

6. Define grammar? Explain Chomsky Hierarchy? Give an example
7. (a) Obtain grammar to generate string consisting of any number of a's and b's with at least one b.
   - Obtain a grammar to generate the following language: L ={WW^R where W□{a, b}*}
8. (a) Obtain a grammar to generate the following language: L = { 0^m $1^m 2^n$ | m>= 1 and n>=0}
   - Obtain a grammar to generate the set of all strings with no more than three a's when Σ = {a, b}
9. Obtain a grammar to generate the following language:
   (i) L = { w | $n_a(w) > n_b(w)$ }
   (ii) L = { $a^n b^m c^k$ | n+2m = k for n>=0, m>=0}
10. Define derivation , types of derivation , Derivation tree & ambiguous grammar. Give example for each.
11. Is the following grammar ambiguous?
    S → aB | bA
    A → aS | bAA |a
    B → bS | aBB | b
12. Define PDA. Obtain PDA to accept the language L = {$a^n b^n$ | n>=1} by a final state.
13. write a short note on application of context free grammar.

2.6:Regular languages

In theoretical computer science and formal language theory, a **regular language** is a formal language that can be expressed using a regular expression. Note that the "regular expression" features provided with many programming languages are augmented with features that make them capable of recognizing languages that can not be expressed by the formal regular expressions (*as formally defined below*).

In the Chomsky hierarchy, regular languages are defined to be the languages that are generated by Type-3 grammars (regular grammars). Regular languages are very useful in input parsing and programming language design.

Formal definition

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- The empty language Ø is a regular language.
- For each $a \in \Sigma$ (a belongs to Σ), the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \cdot B$ (concatenation), and A^* (Kleene star) are regular languages.
- No other languages over Σ are regular.

See regular expression for its syntax and semantics. Note that the above cases are in effect the defining rules of regular expression

Examples

All finite languages are regular; in particular the empty string language {ε} = Ø* is regular. Other typical examples include the language consisting of all strings over the alphabet $\{a, b\}$ which contain an even number of *a*s, or the language consisting of all strings of the form: several *a*s followed by several *b*s.

A simple example of a language that is not regular is the set of strings $\{a^n b^n \mid n \geq 0\}$. Intuitively, it cannot be recognized with a finite automaton, since a finite automaton has finite memory and it cannot remember the exact number of a's. Techniques to prove this fact rigorously are given below.

proving languages not to be regular languages

- Pumping Lemma
  Used to prove certain languages like $L = \{0^n 1^n \mid n \geq 1\}$ are not regular.

- Closure properties of regular languages
  Used to build recognizers for languages that are constructed from other languages by certain operations.
  Ex. Automata for intersection of two regular languages
- Decision properties of regular languages
  - Used to find whether two automata define the same language
  - Used to minimize the states of DFSM
    eg. Design of switching circuits.

Pumping Lemma for regular languages (Explanation)

Let $L = \{0^n 1^n \mid n \geq 1\}$

There is no regular expression to define L. 00*11* is not the regular expression defining L.
Let L= $\{0^2 1^2$

State 6 is a trap state, state 3 remembers that two 0's have come and from there state 5 remembers that two 1's are accepted.

This implies DFSM has no memory to remember arbitrary 'n'. In other words if we have to
remember n, which varies from 1 to ∞ we have to have infinite states, which is not possible with a finite state machine, which has finite number of states.

Pumping Lemma (PL) for Regular Languages

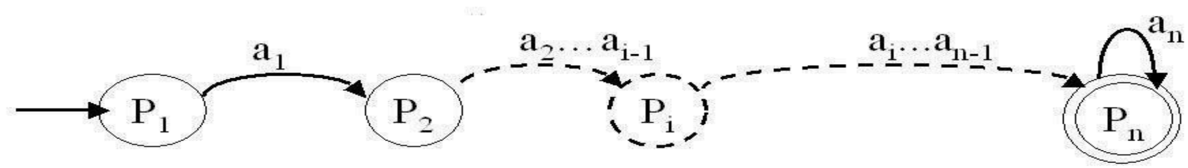
Theorem:

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

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

PROOF:

Let L be regular defined by an FA having 'n' states. Let w= a_1,a_2,a_3----a_n and is in L.
$|w| = n \geq n$. Let the start state be P_1. Let w = xyz where x= a_1,a_2,a_3 -----a_{n-1} , y=a_n and z = ε.



$\delta(P_1,a_1)=P_2$

$\delta(P_2,a_2)=P_3$

$\vdots$

$\delta(P_n,a_n)=P_{n+1}$

But there are only n states. => there must be a loop. Let there be a loop in P_n State.

Let $x=a_1,\ldots\ldots a_{n-1}$

$y=a_n$

$z=\varepsilon$

Therefore $xy^kz = a_1$ ------ a_{n-1} $(a_n)^k$ ε

k=0 a_1 ------ a_{n-1} is accepted

k=1 a_1 ------ a_n is accepted

k=2 a_1 ------ a_{n+1} is accepted

k=10 a_1 ------ a_{n+9} is accepted and so on.

Uses of Pumping Lemma: - This is to be used to show that, certain languages are not regular. It should never be used to show that some language is regular. If you want to show that language is regular, write separate expression, DFSM or NDFSM.

General Method of proof: -

(i) Select w such that $|w| \geq n$

(ii) Select y such that $|y| \geq 1$

(iii) Select x such that $|xy| \leq n$

(iv) Assign remaining string to z

(v) Select k suitably to show that, resulting string is not in L.

Example 1.

To prove that $L=\{w|w\ \varepsilon\ a^nb^n$, where $n \geq 1\}$ is not regular

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L. Let $w = a^nb^n$, such that $|w|=2n$. Since $2n > n$ and L is regular it must satisfy PL.

Consider $w = \underbrace{\overbrace{aa\text{-----}a}^{n}}_{xy}\ \underbrace{\overbrace{bb\text{-----}b}^{n}}_{z}$

xy contain only a's. (Because $|xy| \leq n$).
Let $|y|=l$, where $l > 0$ (Because $|y| > 0$).

Then, the break up of x. y and z can be as follows

$$w = \overbrace{a^{n-l}}^{x}\ \overbrace{a^{l}}^{y}\ \overbrace{b^{n}}^{z}$$

from the definition of PL , $w=xy^kz$, where k=0,1,2,------∞, should belong to L.

That is $a^{n-l}\,(a^l)^k\,b^n \in L$, for all k=0,1,2,------→∞

Put k=0. we get $a^{n-l}\,b^n \notin L$.

Contradiction. Hence the Language is not regular.

Example 2.

To prove that L={w|w is a palindrome on {a,b}*} is not regular. i.e., L={aabaa, aba, abbbba,…}

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L. Let w = a^nba^n, such that $|w|=2n+1$. Since $2n+1 > n$ and L is regular it must satisfy PL.

Consider $w = \underbrace{\overbrace{aa\text{-----}a}^{n}}_{xy}\ \underbrace{b\ \overbrace{aa\text{-----}a}^{n}}_{z}$

xy contain only a's. **(Because $|xy| \leq n$).**
Let $|y|=l$, where $l > 0$ **(Because $|y| > 0$).**

That is, the break up of x. y and z can be as follows

$$w = \overbrace{a^{n-l}}^{x}\ \overbrace{a^{l}}^{y}\ \overbrace{ba^{n}}^{z}$$

from **the definition** of PL $w=xy^kz$, where k=0,1,2,------∞, should belong to L.

That is $a^{n-l}\,(a^l)^k\,ba^n \in L$, for all k=0,1,2,------→∞.

Put k=0. we get $a^{n-l}\,b\,a^n \notin L$, because, it is not a palindrome. Contradiction, hence the language is not regular

.

Example 3.

To prove that L={ all strings of 1's whose length is prime} is not regular. i.e., L={1^2, 1^3 ,1^5 ,1^7 ,1^{11} ,----}

Proof: Let L be regular. Let w = 1^p where p is prime and | p| = n +2

Let y = m.

by PL xy^kz □L

$| xy^kz | = | xz | + | y^k |$ Let k = p-m

$= (p-m) + m (p-m)$

$= (p-m) (1+m)$ ----- this can not be prime if $p-m \geq 2$ or $1+m \geq 2$

1. $(1+m) \geq 2$ because $m \geq 1$
2. Limiting case p=n+2
   $(p-m) \geq 2$ since $m \leq n$

Example 4.

To prove that L={ 0^{i^2} | i is integer and i >0} is not regular. i.e., L={0^2, 0^4 ,0^9 ,0^{16} ,0^{25} ,----}

Proof: Let L be regular. Let w = 0^{n^2} where $|w| = n^2 \geq n$

by PL xy^kz □L, for all k = 0,1,---

Select k = 2

$| xy^2z | = | xyz | + | y |$

$= n^2 +$ Min 1 and Max n

Therefore $n^2 < | xy^2z | \leq n^2 + n$

$n^2 < | xy^2z | < n^2 + n + 1+n$ adding 1 + n (Note that less than or equal to is replaced by less than sign)
$n^2 < | xy^2z | < (n + 1)^2$

Say n = 5 this implies that string can have length > 25 and < 36 which is not of the form 0^{i^2}.

a) Show that following languages are not regular

2.8 :<u>**closure properties of regular languages**</u>

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.

4. The difference of two regular languages is regular.

5. The reversal of a regular language is regular.

6. The closure (star) of a regular language is regular.

7. The concatenation of regular languages is regular.

8. A homomorphism (substitution of strings for symbols) of a regular language is regular.

9. The inverse homomorphism of a regular language is regular

Closure under Union

<u>Theorem</u>: If L and M are regular languages, then so is L □M.

Ex1.

$L1=\{a,a^3,a^5,-----\}$

$L2=\{a^2,a^4,a^6,-----\}$

$L1$□$L2 = \{a,a^2,a^3,a^4,----\}$

$RE=a(a)^*$

Ex2.

$L1=\{ab, a^2 b^2, a^3b^3, a^4b^4,-----\}$

$L2=\{ab,a^3 b^3,a^5b^5,-----\}$

$L1$□$L2 = \{ab,a^2b^2, a^3b^3, a^4b^4, a^5b^5----\}$

$RE=ab(ab)^*$

Closure Under Complementation

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

Ex1.

$L1=\{a,a^3,a^5,-----\}$

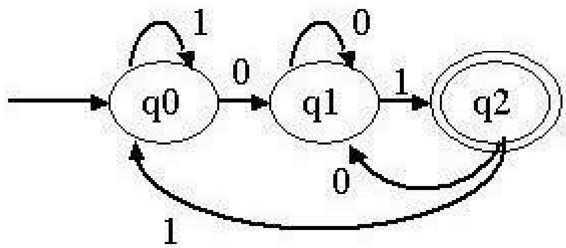
$\mathbf{\Sigma}^*$ $-L1=\{e,a^2,a^4,a^6,-----\}$

RE=(aa)*

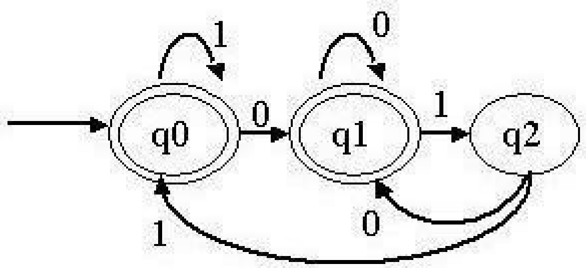
Ex2.

Consider a DFSM, A that accepts all and only the strings of 0's and 1's that end in 01. That is L(A) = $(0+1)^*01$. The complement of L(A) is therefore all string of 0's and 1's that do not end in 01

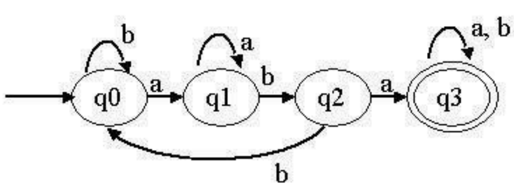
$L(A)=(0+1)^*01$



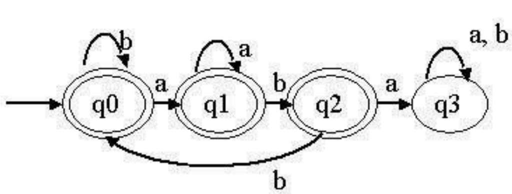
$\overline{L(A)}=\{0,1\}^* - L(A)$



$L(A)=(a+b)^*aba\,(a+b)^*$



$\overline{L(A)}=\{a,b,c\}^* - L(A)$



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

<u>Proof</u>: - Let L =L(A) for some DFSM. A=(Q, **Σ, δ,** q_0, F). Then $\overline{L}$ = L(B), where B is the DFSM (Q, **Σ, δ,** q_0, Q-F). That is, B is exactly like A, but the accepting states of A have become non-accepting states of B, and vice versa, then w is in L(B) if and only if $\hat{\delta}(q_0, w)$ is in Q-F, which occurs if and only if w is not in L(A).

Closure Under Intersection

Theorem : If L and M are regular languages, then so is L ∩ M.

Ex1.

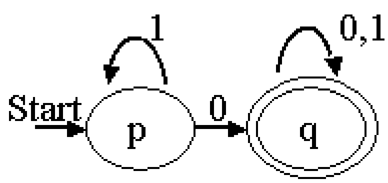
$L1=\{a,a^2,a^3,a^4,a^5,a^6,\text{-----}\}$
$L2=\{a^2,a^4,a^6,\text{-----}\}$
$L1L2 = \{a^2,a^4,a^6,\text{----}\}$
RE=aa(aa)*

Ex2

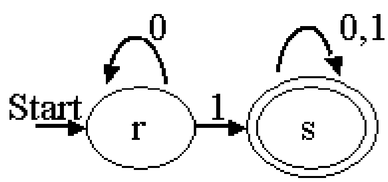
$L1=\{ab,a^3b^3,a^5b^5,a^7b^7\text{-----}\}$
$L2=\{a^2 b^2, a^4b^4, a^6b^6,\text{-----}\}$
$L1 \cap L2 = \varphi$
RE= φ

Ex3.

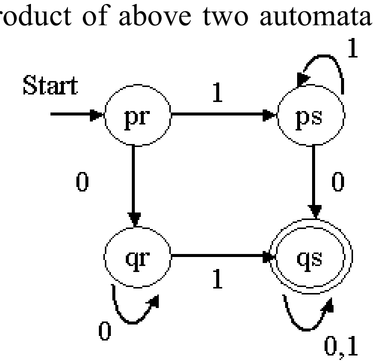
Consider a DFSM that accepts all those strings that have a 0.



Consider a DFSM that accepts all those strings that have a 1.



The product of above two automata is given below.

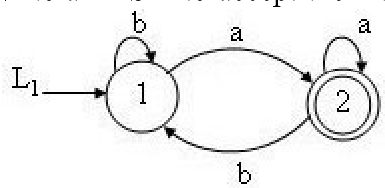


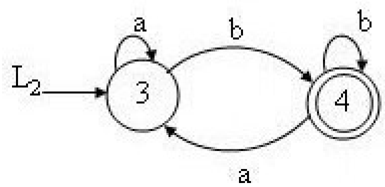
This automaton accepts the intersection of the first two languages: Those languages that have both a 0 and a 1. Then pr represents only the initial condition, in which we have seen neither

0 nor 1. Then state qr means that we have seen only once 0's, while state ps represents the condition that we have seen only 1's. The accepting state qs represents the condition where we have seen both 0's and 1's.

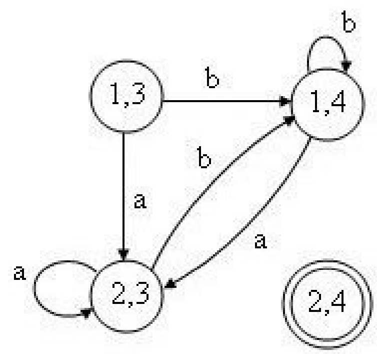
Ex 4 (on intersection)

Write a DFSM to accept the intersection of L1=(a+b)*a and L2=(a+b)*b that is for L1 ∩ L2.



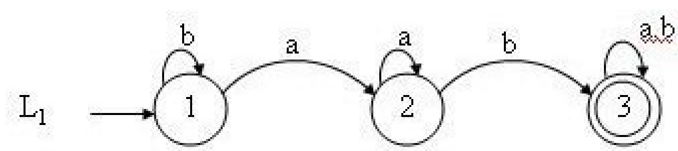


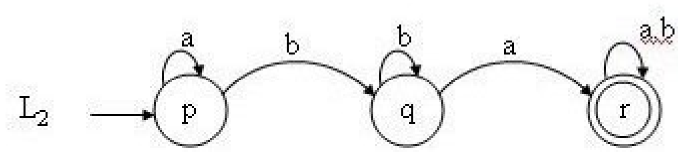
DFSM for L1 ∩ L2 = φ (as no string has reached to final state (2,4))



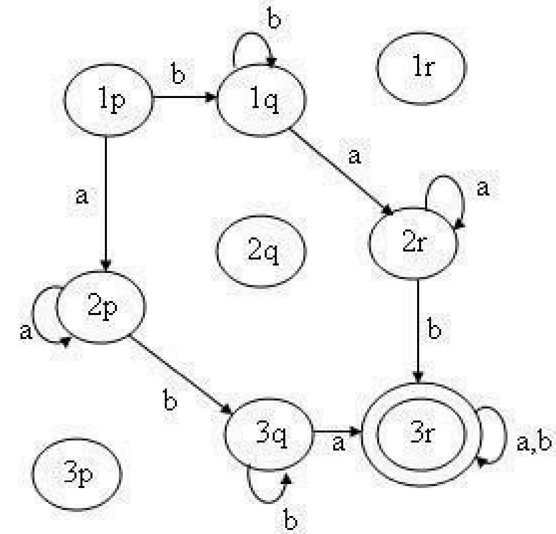
Ex5 (on intersection)

Find the DFSM to accept the intersection of L1=(a+b)*ab (a+b)* and L2=(a+b)*ba (a+b)* that is for L1 ∩ L2





DFSM for L1 ∩ L2



Closure Under Difference

Theorem : If L and M are regular languages, then so is L – M.

Ex.

$L1=\{a,a^3,a^5,a^7,-----\}$

$L2=\{a^2,a^4,a^6,-----\}$

$L1-L2 = \{a,a^3,a^5,a^7----\}$

$RE=a(a)^*$

Reversal

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

Ex.

L={001,10,111,01}

L^R={100,01,111,10}

To prove that regular languages are closed under reversal.

Let L = {001, 10, 111}, be a language over **Σ**={0,1}.
L^R is a language consisting of the reversals of the strings of L.
That is L^R = {100,01,111}.

If L is regular we can show that L^R is also regular.

Proof.

As L is regular it can be defined by an FA, M = (Q, **Σ** , **δ**, q_0, F), having only one final state.
If there are more than one final states, we can use □- transitions from the final states going to a common final state.

Let FA, $M^R = (Q^R, \Sigma^R, \delta^R, q_0^R, F^R)$ defines the language L^R,

Where $Q^R = Q$, $\Sigma^R = \Sigma$, $q_0^R=F$, $F^R=q_0$, and δ^R (p,a)-> q, iff δ (q,a) -> p

Since M^R is derivable from M, L^R is also regular.

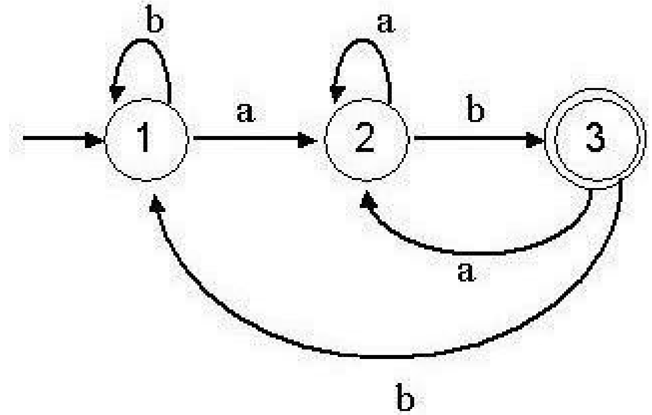
The proof implies the following method

1. Reverse all the transitions.
2. Swap initial and final states.
3. Create a new start state p0 with transition on □ to all the accepting states of original DFSM

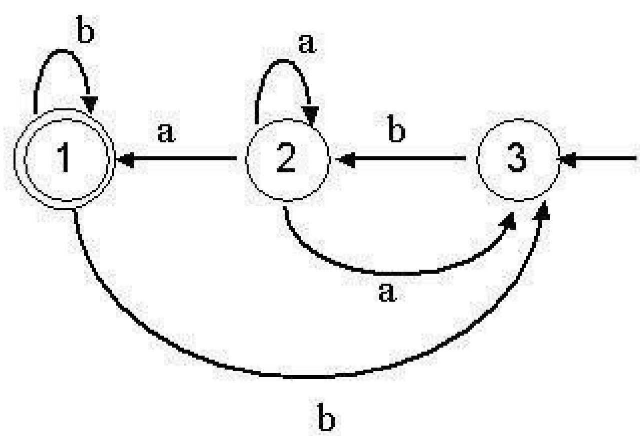
Example

Let r=(a+b)* ab define a language L. That is

L = {ab, aab, bab,aaab, -----}. The FA is as given below



The FA for L^R can be derived from FA for L by swapping initial and final states and changing the direction of each edge. It is shown in the following figure.



Homomorphism

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

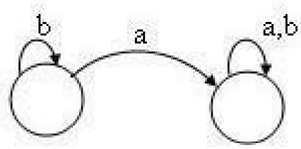
Theorem : If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then h (L) is also regular.

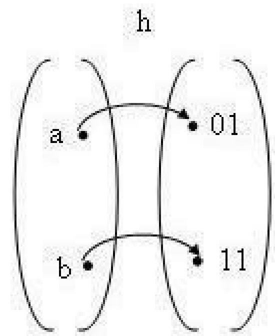
Ex.

The function h defined by h(0)=ab h(1)=c is a homomorphism.

h applied to the string 00110 is ababccab

L1= (a+b)* a (a+b)*





$h : \{a, b\} \longrightarrow \{0, 1\}^*$

$h : \{a, b\} \longrightarrow \{0, 1\}^*$

Ex.

$h_1(a) = 01$

$h_1(b) = 11$

Ex.

$h_2(a) = 101$

$h_2(b) = 010$

Ex.

$h_2(a) = 01$

$h_3(a) = 101$

Resulting :

h1(L) = (01 + 11)* 01 (01 + 11)*

h2(L) = (101 + 010)* 101 (101 + 010)*

h3(L) = (01 + 101)* 01 (01 + 101)*

Inverse Homomorphism

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

Ex.Let L be the language of regular expression $(00+1)^*$.
Let h be the homomorphism defined by h(a)=01 and h(b)=10. Then $h^{-1}(L)$ is the language of regular expression $(ba)^*$.

2.9: decision properties of regular languages

1. is the language described empty?

2. Is a particular string w in the described language?

3. Do two descriptions of a language actually describe the same language?

This question is often called "equivalence" of languages.

Converting Among Representations

Converting NDFSM's to DFSM's

Time taken for either an NDFSM or -NDFSM to DFSM can be exponential in the number of states of the NDFSM. Computing ε-Closure of n states takes $O(n^3)$ time. Computation of DFSM takes $O(n^3)$ time where number of states of DFSM can be 2^n. The running time of NDFSM to DFSM conversion including ε transition is $O(n^3\ 2^n)$. Therefore the bound on the running time is $O(n^3 s)$ where s is the number of states the DFSM actually has.

DFSM to NDFSM Conversion

Conversion takes O(n) time for an n state DFSM.

Automaton to Regular Expression Conversion

For DFSM where n is the number of states, conversion takes $O(n^3 4^n)$ by substitution method and by state elimination method conversion takes $O(n^3)$ time. If we convert an NDFSM to DFSM
and then convert the DFSM to a regular expression it takes the time $O(n^3 4^{n^3} 2^n)$

Regular Expression to Automaton Conversion

Regular expression to ε-NDFSM takes linear time – O(n) on a regular expression of length n. Conversion from ε-NDFSM to NDFSM takes $O(n^3)$ time.

Testing Emptiness of Regular Languages

Suppose R is regular expression, then

1. R = R1 + R2. Then L(R) is empty if and only if both L(R1) and L(R2) are empty.
2. R= R1R2. Then L(R) is empty if and only if either L(R1) or L(R2) is empty.
3. R=R1* Then L(R) is not empty. It always includes at least ε
4. R=(R1) Then L(R) is empty if and only if L(R1) is empty since they are the same language.

Testing Emptiness of Regular Languages

Suppose R is regular expression, then

1. R = R1 + R2. Then L(R) is empty if and only if both L(R1) and L(R2) are empty.
2. R= R1R2. Then L(R) is empty if and only if either L(R1) or L(R2) is empty.
3. R=(R1)* Then L(R) is not empty. It always includes at least ε

4. R=(R1) Then L(R) is empty if and only if L(R1) is empty since they are the same language.

Testing Membership in a Regular Language

Given a string w and a Regular Language L, is w in L.

If L is represented by a DFSM, simulate the DFSM processing the string of input symbol
w, beginning in start state. If DFSM ends in accepting state the answer is ‘Yes’ , else it is ‘no’.
This test takes O(n) time

If the representation is NDFSM, if w is of length n, NDFSM has s states, running time of this
algorithm is $O(ns^2)$

If the representation is ε - NDFSM, ε - closure has to be computed, then processing of each input symbol , a , has 2 stages, each of which requires $O(s^2)$ time.

If the representation of L is a Regular Expression of size s, we can convert to an ε - NDFSM with almost 2s states, in O(s) time. Simulation of the above takes $O(ns^2)$ time on an input w of length n

2.10 :Minimization of Automata (Method 1)

Let p and q are two states in DFSM. Our goal is to understand when p and q (p ≠ q) can
be replaced by a single state.

Two states p and q are said to be distinguishable, if there is at least one string, w, such that one of δ^ (p,w) and δ^ (q,w) is accepting and the other is not accepting.

Algorithm 1:

List all unordered pair of states (p,q) for which p ≠ q. Make a sequence of passes through these pairs. On first pass, mark each pair of which exactly one element is in F. On each subsequent pass, mark any pair (r,s) if there is an a� ∑ for which δ (r,a) = p, δ (s,a) = q, and (p,q) is already marked. After a pass in which no new pairs are marked, stop. The marked pair (p,q) are distinguishable.

Examples:

1. Let L = {□, a^2, a^4, a^6, ….} be a regular language over ∑ = {a,b}. The FA is shown in Fig 1.

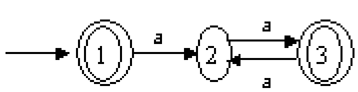


Fig 1.

Fig 2. gives the list of all unordered pairs of states (p,q) with p ≠ q.

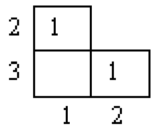


Fig 2.

The boxes (1,2) and (2,3) are marked in the first pass according to the algorithm 1.

In pass 2 no boxes are marked because, $\delta(1,a) \rightarrow \varphi$ and $\delta(3,a) \rightarrow 2$. That is $(1,3) \xrightarrow{a} (\varphi,2)$, where φ and 3 are non final states.

□(1,b) $\rightarrow \varphi$ and □ (3,b) $\rightarrow \varphi$. That is $(1,3) \xrightarrow{b} (\varphi,\varphi)$, where φ is a non-final state. This implies that (1,3) are equivalent and can replaced by a single state A.

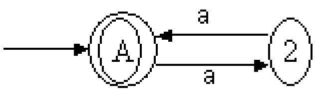
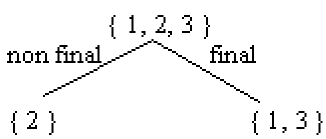


Fig 3. Minimal Automata corresponding to FA in Fig 1

Minimization of Automata (Method 2)



Consider set {1,3}. $(1,3) \xrightarrow{a} (2,2)$ and $(1,3) \xrightarrow{b} (\varphi,\varphi)$. This implies state 1 and 3 are equivalent and can not be divided further. This gives us two states 2,A. The resultant FA is shown is Fig 3.

Example 2. (Method1):

Let r= (0+1)*10, then L(r) = {10,010,00010,110, ---}. The FA is given below

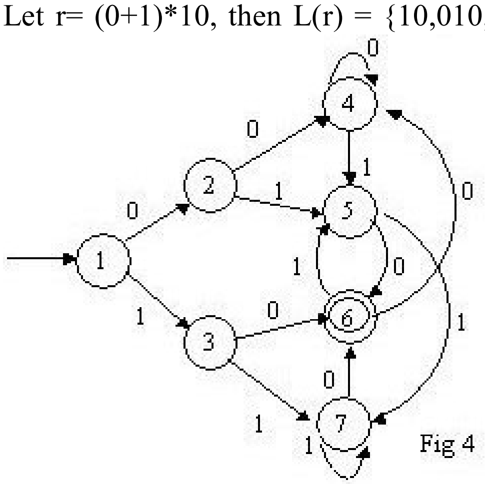


Fig 4

Following fig shows all unordered pairs (p,q) with $p \neq q$

	1	2	3	4	5	6
2						
3	2	2				
4			2			
5	2	2		2		
6	1	1	1	1	1	
7	2	2		2		1

In pass 2,

$(3,1) \xrightarrow{0} (6,2)$
$(3,2) \xrightarrow{0} (6,4)$
$(5,1) \xrightarrow{0} (6,2)$
$(5,2) \xrightarrow{0} (6,4)$
$(7,4) \xrightarrow{0} (6,4)$
$(7,2) \xrightarrow{0} (6,4)$
and so on.

The pairs marked 1 are those of which exactly one element is in F; They are marked on pass 1. The pairs marked 2 are those marked on the second pass. For example (5,2) is one of these, since (5,2) → (6,4), and the pair (6,4) was marked on pass 1.

From this we can make out that 1, 2, and 4 can be replaced by a single state 124 and states 3, 5, and 7 can be replaced by the single state 357. The resultant minimal FA is shown in Fig. 6

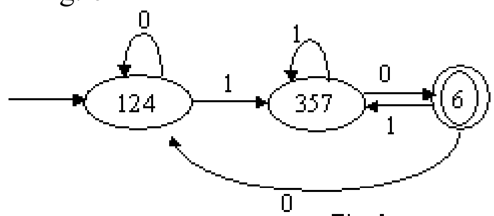


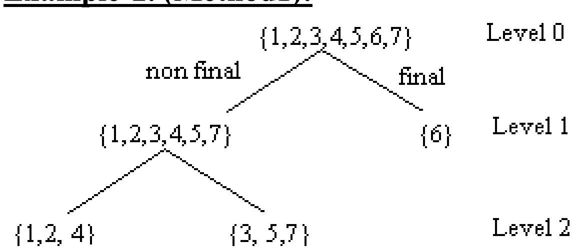
Fig 6

The transitions of fig 4 are mapped to fig 6 as shown below

<u>Original DFA</u>	<u>Minimal DFA</u>
$1 \xrightarrow{0} 2$	$124 \xrightarrow{0} 124$
$2 \xrightarrow{1} 5$	$124 \xrightarrow{1} 357$
$3 \xrightarrow{0} 6$	$357 \xrightarrow{0} 6$

and so on.

<u>Example 2. (Method1):</u>



$(2,3) \xrightarrow{0} (4,6)$ this implies that 2 and 3 belongs to different group hence they are split in level 2. similarly it can be easily shown for the pairs (4,5) (1,7) and (2,5) and so on.

MODULE 3:
Context Free Grammar and languages

3.1: Context free grammar

Context Free grammar or CGF, G is represented by four components that is G=(V,T,P,S), where V is the set of variables, T the terminals, P the set of productions and S the start symbol.

Example: The grammar G_{pal} for palindromes is represented by

G_{pal} = ({P},{0,1}, A, P)

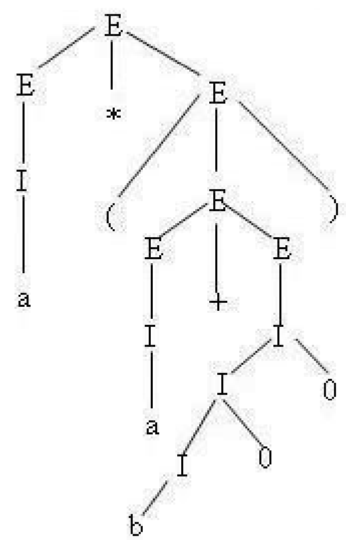
where A represents the set of five productions

1. P→□
2. P→0
3. P→1
4. P→0P0
5. P→1P1

Derivation using Grammar

Consider a context-free grammar for simple expressions

1. E→ I
2. E→ E + E
3. E→ E * E
4. E→ (E)
5. I→ a
6. I→ b
7. I→ Ia
8. I→ Ib
9. I→ I0
10. I→ I1



3.2: parse trees

Parse trees are trees labeled by symbols of a particular CFG.

Leaves: labeled by a terminal or ε.

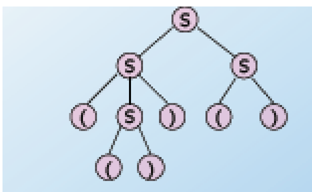
Interior nodes: labeled by a variable.

Children are labeled by the right side of a production for the parent.

Root: must be labeled by the start symbol.

Example: Parse Tree

S -> SS | (S) | ()

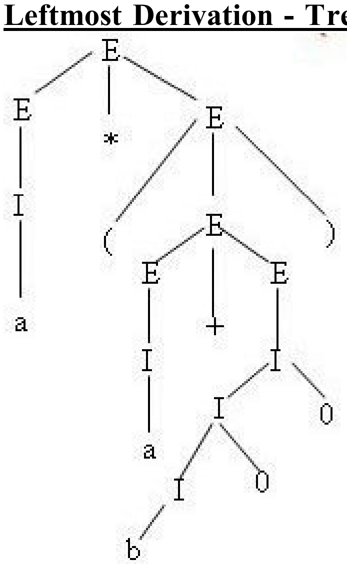


Example 1: Leftmost Derivation

The inference that a * (a+b00) is in the language of variable E can be reflected in a derivation of that string, starting with the string E. Here is one such derivation:

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow$$
$$a * (E) \Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow$$
$$a * (a + I) \Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$$

Leftmost Derivation - Tree



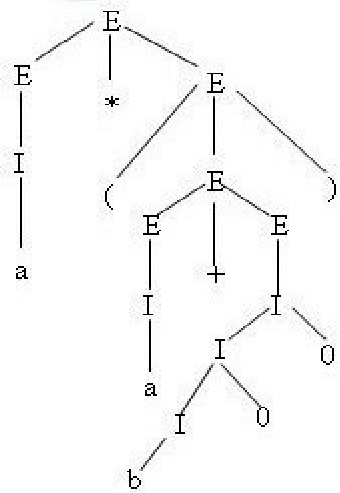
Example 2: Rightmost Derivations

The derivation of Example 1 was actually a leftmost derivation. Thus, we can describe the same derivation by:

E➔ E * E ➔ E *(E) ➔ E * (E + E) ➔
E * (E + I) ➔ E * (E +I0) ➔ E * (E + I00) ➔ E * (E + b00) ➔
E * (I + b00) ➔ E * (a +b00) ➔ I * (a + b00) ➔ a * (a + b00)

We can also summarize the leftmost derivation by saying
E ➔ a * (a + b00), or express several steps of the derivation by expressions such as E * E ➔ a * (E).

Rightmost Derivation - Tree



There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

E ➔ E * E ➔ E * (E) ➔ E * (E + E) ➔
E * (E + I) ➔ E * (E + I0) ➔ E * (E + I00) ➔ E * (E + b00) ➔
E * (I + b00) ➔ E * (a + b00) ➔ I * (a + b00) ➔ a * (a + b00)
This derivation allows us to conclude E ➔ a * (a + b00)

Consider the Grammar for string(a+b)*c

E➔E + T | T
T➔ T * F | F
F➔ (E) | a | b | c

Leftmost Derivation
E➔T➔T*F➔F*F➔(E)*F➔(E+T)*F➔(T+T)*F➔(F+T)*F ➔(a+T)*F ➔(a+F)*F ➔(a+b)*F➔(a+b)*c

Rightmost derivation
E➔T➔T*F➔T*c➔F*c➔(E)*c➔(E+T)*c➔(E+F)*c ➔(E+b)*c➔(T+b)*c➔(F+b)*c➔(a+b)*c

Example 2:

Consider the Grammar for string (a,a)

S->(L)|a

L->L,S|S

Leftmost derivation

S→(L)→(L,S)→(S,S)→(a,S)→(a,a)

Rightmost Derivation

S→(L)→(L,S)→(L,a)→(S,a)→(a,a)

The Language of a Grammar

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

L(G) = {w in T | S → w}

Sentential Forms

Derivations from the start symbol produce strings that have a special role called "sentential forms". That is if G = (V, T, P, S) is a CFG, then any string in (V T)* such that S →α is a sentential form. If S →α, then is a left – sentential form, and if S →α , then is a right – sentential form. Note that the language L(G) is those sentential forms that are in T*; that is they consist solely of terminals.

For example, E * (I + E) is a sentential form, since there is a derivation

E → E * E → E * (E) → E * (E + E) → E * (I + E)

However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

As an example of a left – sentential form, consider a * E, with the leftmost derivation.

E → E * E → I * E → a * E

Additionally, the derivation

E → E * E → E * (E) → E * (E + E)

Shows that

E * (E + E) is a right – sentential form.

3.3: Applications of Context – Free Grammars

- Parsers
- The YACC Parser Generator
- Markup Languages
- XML and Document typr definitions

The YACC Parser Generator

E→ E+E | E*E | (E)|id

```
%{ #include <stdio.h>
%}
%token ID id
%%
Exp  :  id   { $$ = $1 ; printf ("result is %d\n", $1);}
                | Exp '+' Exp    {$$ = $1 + $3;}
                | Exp '*' Exp    {$$ = $1 * $3; }
                |  '(' Exp ')'      {$$ = $2; }
                ;
%%

int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
[0-9]+            {yylval.ID = atoi(yytext); return id;}
[ \t \n]           ;
[+ * ( )]                   {return yytext[0];}
.                           {ECHO; yyerror ("unexpected character");}
%%
```

Example 2:

```
%{
 #include <stdio.h>
%}
%start line
%token <a_number> number
%type <a_number> exp term factor
%%
line : exp ';' {printf ("result is %d\n", $1);}
;
exp : term {$$ = $1;}
        | exp '+' term {$$ = $1 + $3;}
        | exp '-' term {$$ = $1 - $3;}
term : factor {$$ = $1;}
        | term '*' factor {$$ = $1 * $3;}
        | term '/' factor {$$ = $1 / $3;}
 ;
factor : number {$$ = $1;}
```

```
| '(' exp ')' {$$ = $2;}
;
 %%
 int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
[0-9]+   {yylval.a_number = atoi(yytext); return number;}
[ \t\n] ;
[-+*/();]  {return yytext[0];}
.                        {ECHO; yyerror ("unexpected character");}
%%
```

Markup Languages

Functions

•Creating links between documents
•Describing the format of the document

Example

The Things I *hate*

1. Moldy bread
2. People who drive too slow
   In the fast lane

HTML Source

```
<P> The things I <EM>hate</EM>:
<OL>
<LI> Moldy bread
<LI>People who drive too slow
In the fast lane
</OL>
```

HTML Grammar

•Char a | A | …

•Text e | Char Text
•Doc e | Element Doc
•Element Text |
<EM> Doc </EM>|
<p> Doc |
<OL> List </OL>| …
5. List-Item <LI> Doc
6. List e | List-Item List Start symbol

XML and Document type definitions.

1. A→E1,E2.
   A→BC
   B→E1
   C→E2
2. A→E1 | E2.
   A→E1
   A→E2
3. A→(E1)*
   A→BA
   A→ε
   B→E1
4. A→(E1)+
   A→BA
   A→B
   B→E1
5. A→(E1)?
   A→ε
   A→E1

3.4:Ambiguity

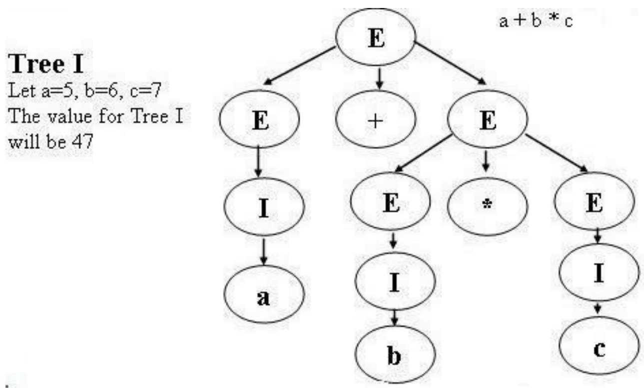
A context – free grammar G is said to be ambiguous if there exists some w L(G) which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or rightmost derivations.

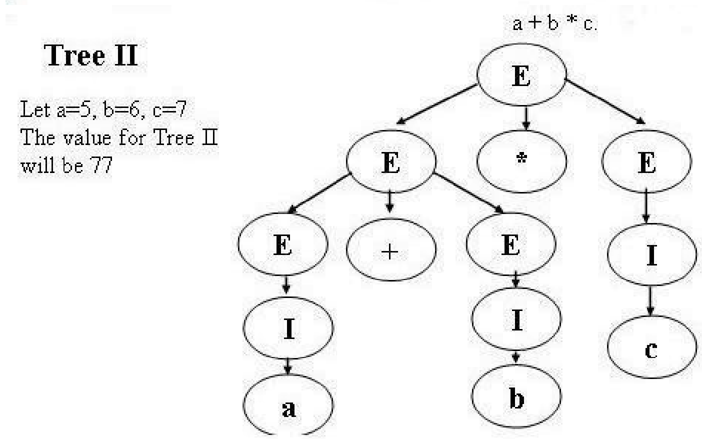
Ex:-

Consider the grammar G=(V,T,E,P) with V={E,I}, T={a,b,c,+,*,(,)}, and productions.

E→I,
E→E+E,
E→E*E,
E→(E),
I→a|b|c

Consider two derivation trees for a + b * c.





Now unambiguous grammar for the above
Example:

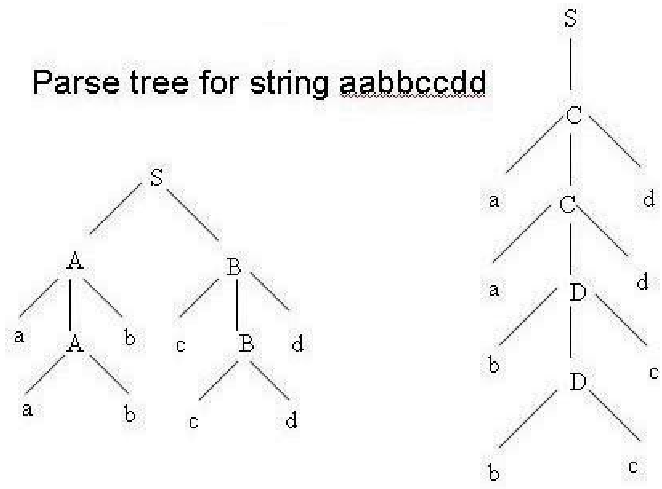
E→T, T→F, F→I, E→E+T, T→T*F,
F→(E), I→a|b|c

Inherent Ambiguity

A CFL L is said to be inherently ambiguous if all its grammars are ambiguous
Example:
Condider the Grammar for string aabbccdd

S→AB | C
A→ aAb | ab
B→cBd | cd
C→ aCd | aDd
D->bDc | bc

Parse tree for string aabbccdd


Parse tree for string aabbccdd

3.5:<u>Definition of pushdown Automata:</u>

As Fig. 5.1 indicates, a *pushdown automaton* consists of three components: 1) an input tape, 2) a control unit and 3) a stack structure. The input tape consists of a linear configuration of cells each of which contains a character from an alphabet. This tape can be moved one cell at a time to the left. The stack is also a sequential structure that has a first element and grows in either direction from the other end. Contrary to the tape head associated with the input tape, the head positioned over the current stack element can read and write special stack characters from that position. The current stack element is always the top element of the stack, hence the name ``stack''. The control unit contains both tape heads and finds itself at any moment in a particular state.

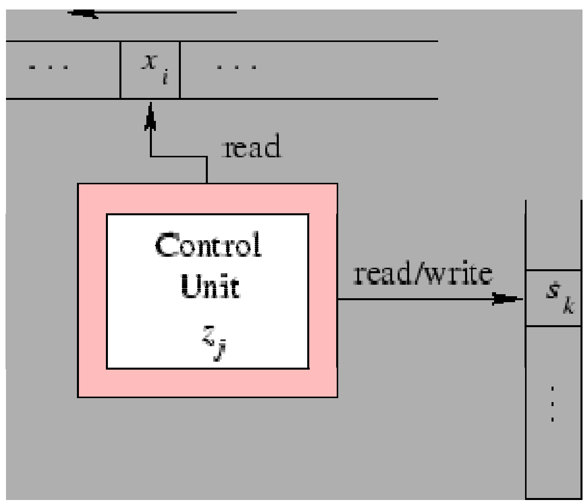


Figure 5.1: Conceptual Model of a Pushdown Automaton

A (non-deterministic) **finite state pushdown automaton** (abbreviated PDA or, when the context is clear, an automaton) is a 7-tuple $\mathcal{P} = (X, Z, \mathbf{S}, R, z_A, S_A, Z_F)$, where

- $X = \{x_1, \cdots, x_m\}$ is a finite set of *input symbols*. As above, it is also called an *alphabet*. The *empty symbol* λ is *not* a member of this set. It does, however, carry its usual meaning when encountered in the input.
- $Z = \{z_1, \cdots z_n\}$ is a finite set of states.
- $\mathbf{S} = \{s_1, \cdots, s_p\}$ is a finite set of stack symbols. In this case $\lambda \in \mathbf{S}$
- $R \subseteq ((X \cup \{\lambda\}) \times Z \times \mathbf{S}^*) \times (Z \times \mathbf{S}^*))$ is the *transition relation*.
- z_A is the *initial state*.
- S_A is the *initial stack symbol*.
- $Z_F \subseteq K$ is a distinguished set of *final states*

3.6 The language of a PDA

There are two ways to define the language of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ($L(P) \subseteq \Sigma^*$). because there are two notions of acceptance:

Acceptance by final state

$$L(P) = \{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \gamma) \wedge q \in F\}$$

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, γ), where $q \in F$ is one of the final states. Here it doesn't play a role what the contents of the stack are at the end.

In our example the PDA P_0 would accept 0110 because $(q_0, 0110, \#) \vdash_{P_0}^* (q_2, \epsilon, \epsilon)$ and $q_2 \in F$. Hence we conclude $0110 \in L(P_0)$.

On the other hand since there is no successful sequence of IDs starting with $(q_0, 0011, \#)$ we know that $0011 \notin L(P_0)$.

Acceptance by empty stack

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

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, ϵ), in this case the final state plays no role.

If we specify a PDA for acceptance by empty stack we will leave out the set of final states F and just use $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Our example automaton P_0 also works if we leave out F and use acceptance by empty stack.

We can always turn a PDA which use one acceptance method into one which uses the other. Hence, both acceptance criteria specify the same class of languages.

3.7:Equivalence of PDA and CFG

The aim is to prove that the following three classes of languages are same:

1. Context Free Language defined by CFG
2. Language accepted by PDA by final state
3. Language accepted by PDA by empty stack

It is possible to convert between any 3 classes. The representation is shown in figure 1.

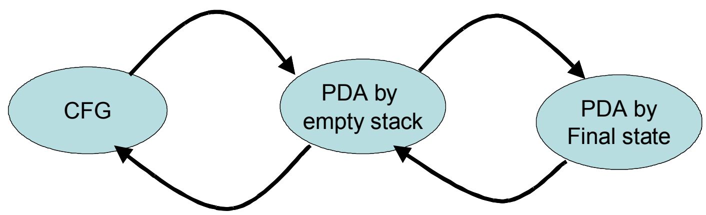


Figure 1: Equivalence of PDA and CFG

From CFG to PDA:

Given a CFG G, we construct a PDA P that simulates the leftmost derivations of G. The stack symbols of the new PDA contain all the terminal and non-terminals of the CFG. There is only one state in the new PDA; all the rest of the information is encoded in the stack. Most transitions are on □, one for each production. New transitions are added, each one corresponding to terminals of G. For every intermediate sentential form uA□ in the leftmost derivation of w (initially w = uv for some v), M will have A□ on its stack after reading u. At the end (case u = w) the stack will be empty.

Let G = (V, T, Q, S) be a CFG. The PDA which accepts L(G) by empty stack is given by:

P = ({q}, T, V □ T, δ, q, S) where δ is defined by:

1. For each variable A include a transition,
   δ(q, □, A) = {(q, b) | A □ b is a production of Q}

2. For each terminal a, include a transition
   δ(q, a, a) = {(q, □)}

CFG to PDA conversion is another way of constructing PDA. First construct CFG, and then convert CFG to PDA.

Example:

Convert the grammar with following production to PDA accepted by empty stack:

$S \square 0S1 \mid A$
$A \square 1A0 \mid S \mid \square$

Solution:

$P = (\{q\}, \{0, 1\}, \{0, 1, A, S\}, \delta, q, S)$, where δ is given by:

$$\delta(q, \square, S) = \{(q, 0S1), (q, A)\}$$
$$\delta(q, \square, A) = \{(q, 1A0), (q, S), (q, \square)\}$$
$$\delta(q, 0, 0) = \{(q, \square)\}$$
$$\delta(q, 1, 1) = \{(q, \square)\}$$

From PDA to CFG:

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA. An equivalent CFG is $G = (V, \Sigma, R, S)$, where $V = \{S, [pXq]\}$, where $p, q \square Q$ and $X \square \Gamma$, productions of R consists of

1. For all states p, G has productions $S \square [q_0 Z_0\, p]$
2. Let $\delta(q,a,X) = \{(r, Y_1Y_2\ldots Y_k)\}$ where $a \square \Sigma$ or $a = \square$, k can be 0 or any number and $r_1r_2 \ldots r_k$ are list of states. G has productions

$$[qXr_k] \ \square\ a[rY_1r_1]\ [r_1Y_2r_2]\ \ldots\ [r_{k-1}Y_kr_k]$$

$$\text{If } k = 0 \text{ then } [qXr] \square a$$

Example:

Construct PDA to accept if-else of a C program and convert it to CFG. (This does not accept if –if –else-else statements).

Let the PDA $P = (\{q\}, \{i, e\}, \{X,Z\}, \delta, q, Z)$, where δ is given by:

$\delta(q, i, Z) = \{(q, XZ)\}$, $\delta(q, e, X) = \{(q, \square)\}$ and $\delta(q, \square, Z) = \{(q, \square)\}$

Solution:

Equivalent productions are:

$S \rightarrow [qZq]$
$[qZq] \rightarrow i[qXq][qZq]$
$[qXq] \rightarrow e$
$[qZq] \rightarrow \epsilon$

If [qZq] is renamed to A and [qXq] is renamed to B, then the CFG can be defined by:

$G = (\{S, A, B\}, \{i, e\}, \{S \rightarrow A, A \rightarrow iBA \mid \epsilon, B \rightarrow e\}, S)$

Example:
Convert PDA to CFG. PDA is given by $P = (\{p,q\}, \{0,1\}, \{X,Z\}, \delta, q, Z))$, Transition

function δ is defined by:

$\delta(q, 1, Z) = \{(q, XZ)\}$
$\delta(q, 1, X) = \{(q, XX)\}$
$\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
$\delta(q, 0, X) = \{(p, X)\}$
$\delta(p, 1, X) = \{(p, \epsilon)\}$
$\delta(p, 0, Z) = \{(q, Z)\}$

Solution:

Add productions for start variable
$S \rightarrow [qZq] \mid [qZp]$

For $\delta(q, 1, Z)= \{(q, XZ)\}$
$[qZq] \rightarrow 1[qXq][qZq]$
$[qZq] \rightarrow 1[qXp][pZq]$
$[qZp] \rightarrow 1[qXq][qZp]$
$[qZp] \rightarrow 1[qXp][pZp]$

For $\delta(q, 1, X)= \{(q, XX)\}$
$[qXq] \rightarrow 1[qXq][qXq]$
$[qXq] \rightarrow 1[qXp][pXq]$
$[qXp] \rightarrow 1[qXq][qXp]$
$[qXp] \rightarrow 1[qXp][pXp]$

For $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
$[qXq] \rightarrow \epsilon$

For $\delta(q, 0, X) = \{(p, X)\}$
$[qXq] \rightarrow 0[pXq]$

$[qXp] \rightarrow 0[pXp]$

For $\delta(p, 1, X) = \{(p, \varepsilon)\}$

$[pXp] \rightarrow 1$

For $\delta(p, 0, Z) = \{(q, Z)\}$

$[pZq] \rightarrow 0[qZq]$

$[pZp] \rightarrow 0[qZp]$

Renaming the variables [qZq] to A, [qZp] to B, [pZq] to C, [pZp] to D, [qXq] to E [qXp] to F, [pXp] to G and [pXq] to H, the equivalent CFG can be defined by:

G = ({S, A, B, C, D, E, F, G, H}, {0,1}, R, S). The productions of R also are to be renamed accordingly.

3.8:Deterministic PDA

NPDA provides non-determinism to PDA. Deterministic PDA's (DPDA) are very useful for use in programming languages. For example Parsers used in YACC are DPDA's.

Definition:

A PDA $P= (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic if and only if,

1.$\delta(q,a,X)$ has at most one member for $q \in Q$, $a \in \Sigma$ or $a= \varepsilon$ and $X \in \Gamma$

2.If $\delta(q,a,X)$ is not empty for some $a \in \Sigma$, then $\delta(q, \varepsilon, X)$ must be empty

DPDA is less powerful than nPDA. The Context Free Languages could be recognized by nPDA. The class of language DPDA accept is in between than of Regular language and CFL. NPDA can be constructed for accepting language of palindromes, but not by DPDA.

Example:

Construct DPDA which accepts the language L = {wcw^R | w □ {a, b}*, c □ Σ}.

The transition diagram for the DPDA is given in figure 2.

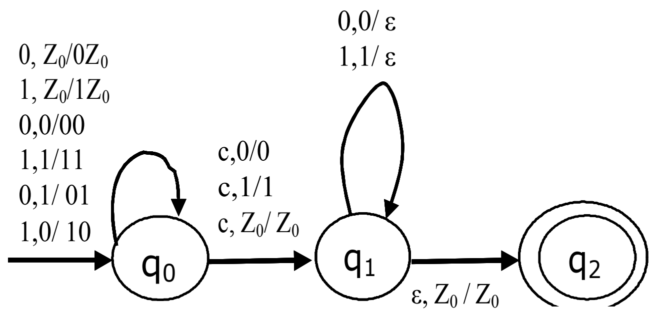


Figure 2: DPDA L = {wcw^R}

DPDA and Regular Languages:

The class of languages DPDA accepts is in between regular languages and CFLs. The DPDA languages include all regular languages. The two modes of acceptance are not same for DPDA.

To accept with final state:

If L is a regular language, L=L(P) for some DPDA P. PDA surely includes a stack, but the DPDA used to simulate a regular language does not use the stack. The stack is inactive always. If A is the FA for accepting the language L, then δ_P(q,a,Z)={(p,Z)} for all p, q □ Q such that δ_A(q,a)=p.

To accept with empty stack:

Every regular language is not N(P) for some DPDA P. A language L = N(P) for some DPDA P if and only if L has prefix property. Definition of prefix property of L states that if x, y □ L, then x should not be a prefix of y, or vice versa. Non-Regular language L=WcW^R could be accepted by DPDA with empty stack, because if you take any x, y□ L(WcW^R), x and y satisfy the prefix property. But the language, L={0*} could be accepted by DPDA with final state, but not with empty stack, because strings of this language do not satisfy the prefix property. So N(P) are properly included in CFL L, ie. N(P) □ L

DPDA and Ambiguous grammar:

DPDA is very important to design of programming languages because languages DPDA accept are unambiguous grammars. But all unambiguous grammars are not accepted by DPDA. For example $S \to 0S0|1S1|\epsilon$ is an unambiguous grammar corresponds to the language of palindromes. This is language is accepted by only nPDA. If L = N(P) for DPDA P, then surely L has unambiguous CFG.

If L = L(P) for DPDA P, then L has unambiguous CFG. To convert L(P) to N(P) to have

prefix property by adding an end marker \$ to strings of L. Then convert N(P) to CFG G'.

From G' we have to construct G to accept L by getting rid of \$ .So add a new production

$\$ \to \epsilon$ as a variable of G.

MODULE-4:

Context-Free and Non-Context-Free Languages

4.1 Normal forms for CFGS

4.2The pumping lemma for CFGS

4.3closure properties of CFLS

The goal is to take an arbitrary Context Free Grammar G = (V, T, P, S) and perform transformations on the grammar that preserve the language generated by the grammar but reach a specific format for the productions. A CFG can be simplified by eliminating

4.1 Normal forms for CFGS

How to simplify?

• Simplify CFG by eliminating

– Useless symbols

• Those variables or terminals that do not appear in any derivation of a terminal string starting from Start variable

– □□- productions

• A □□, where A is a variable

– Unit production

• A □B, A and B are variables

• Sequence to be followed

1. Eliminate □□- productions from G and obtain G1

2. Eliminate unit productions from G1 and obtain G2

3. Eliminate useless symbols from G2and obtain G3

1. Eliminate useless symbols:

Definition: Symbol X is useful for a grammar G = (V, T, P, S) if there is S *□ □X□ *□w, w□□*. If X is not useful, then it is useless.
Omitting useless symbols from a grammar does not change the language generated

• Example

$S \to aSb \mid \varepsilon \mid A$	$S \to A$
$A \to aA$	$A \to aA \mid \varepsilon$
A is a useless symbol	$B \to bB$
	B is a useless symbol

• Symbol X is useful if both

– X is generating

• If X *□ w,where w□T*

– X is reachable

• If S *□ □X□

• Theorem:

– Let G=(V,T,P,S) be a CFG and assume that L(G)□□, then G1=(V1,T1,P1,S) be a grammar without useless symbols by

1. Eliminating non generating symbols
2. Eliminating symbols that are non reachable

• Elimination in the order of 1 followed by 2

1. Eliminating non generating symbols

Generating symbols follow to one of the categories below:

1. Every symbol of T is generating
2. If A □ □ and □ is already generating, then A is generating

Non-generating symbols = V- generating symbols.

• Example : S □AB|a, A □a

– 1 followed by 2 gives S □□a

– 2 followed by 1 gives S □□a, A □a

• A is still useless

• Not completely all useless symbols eliminated

• Eliminate non generating symbols

– Every symbol of T is generating

– If A □□□and □□is already generating, then A is generating

• Example

1. G= ({S,A,B}, {a}, S □AB|a, A □a}, S) here B is non generating symbol

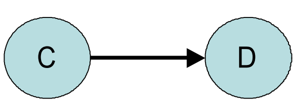
After eliminating B, G1= ({S,A}, {a}, {S □a, A □a},S)

2. S □aS|A|C, A □a, B □aa, C □aCb

After eliminating C gets, S □aS|A, A □a, B □aa

2. Eliminate symbols that are non reachable

– Draw dependency graph for all productions



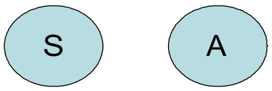
C □xDy

– If no edge reaching a variable X from Start

symbol, X is non reachable

• Example

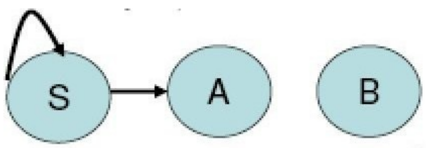
1. G= ({S,A}, {a}, {S □a, A □a},S)



After eliminating A, G1= ({S}, {a}, {S □a},S)

2. S □aS|A, A □a, B □aa

After eliminating B, S □aS|A, A □a

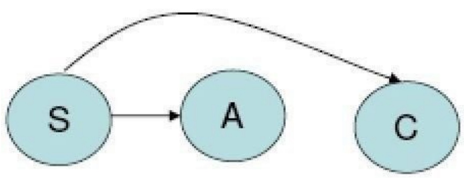


• Example

– S □AB | CA, B □BC|AB, A □a, C □AB|b

1. Eliminate non generating symbols V1 = {A,C,S} P1 = {S □CA, A □a, C □b }

2. Eliminate symbols that are non reachable



V2 = {A,C,S}

P2 = {S □□CA, A □a, C □b

Exercises

• Eliminate useless symbols from the grammar

1. P= {S □aAa, A □Sb|bCC, C □abb, E □aC}

2. P= {S □aBa|BC, A □aC|BCC,C □a, B □bcc, D □E, E □d }

3. P= {S □aAa, A □bBB, B □ab, C □aB }

4. P= {S □aS|AB, A □bA,B□AA }

Eliminate □□- productions

• Most theorems and methods about grammars G assume L(G) does not contain □

• Example: G with □□- productions

S → ABA, A→ aA | □, B → bB | □

The procedure to find out an equivalent G with out □-productions

1. Find nullable variables
2. Add productions with nullable variables removed.
3. Remove □-productions and duplicates

Step 1: Find set of nullable variables

Nullable variables: Variables that can be replaced by null (□). If A $^{*□}$ □ then A is a nullable variable.

In the grammar with productions S □ ABA, A □ aA | □, B □ bB | □, A is nullable because of the production A □ □. B is nullable because of the production B □ □. S is nullable because both A and B are nullable.

Step 1: Algorithm to find nullable variables

V: set of variables

N0 ←{A | A in V, production A → □}

repeat

Ni ← N_{i-1} U{A| A in V, A →α, α in N_{i-1}}

until $N_i = N_{i-1}$

• **Step 2:** For each production of the form A □□w, create all possible productions of the form A □□w', where w' is obtained from w by removing one or more occurrences of nullable variables

• Example:

S → ABA | BA | AA | AB | A | B | □

A → aA | □□| a

B → bB | □□| b

• **Step 3:** The desired grammar consists of the original productions together with the productions constructed in step 2, minus any productions of the form A □□

• Example:

S →ABA | BA | AA | AB | A | B

A → aA | a

B → bB | b

PROBLEM:

G = ({S,A,B,D}, {a}, { S →aS|AB, A → □□, B→ □, D →b},S)

• Solution:

Nullable variables = {S, A, B}

New Set of productions:

S →aS | a

S →AB | A | B

D →b

G1= ({S,B,D}, {a}, { S →aS|a|AB|A|B, D →b}, S)

• Eliminate □□- productions from the grammar

Eliminate unit production

Definition:

• Unit production is of form A □□B, A and B are variables

Unit productions could complicate certain proofs and they also introduce extra steps into derivations that technically need not be there. The algorithm for eliminating unit productions from the set of production P is given below:

• Algorithm

1. Add all non unit productions to P1
2. For each unit production A □□B, add to P1 all productions A □□□, where B □□□□is a non-unit production in P.
3. Delete all the unit productions

Example (1): Consider the grammar with production

S → ABA | BA | AA | AB | A | B

A→ aA | a

B →bB | b

Solution:

– Unit productions are S →A, S→B

– A and B are derivable

– Add productions from derivable

S→ ABA | BA | AA | AB | A | B | aA | a | bB | b

A → aA | a

B → bB | b

– Remove unit productions

S → ABA | BA | AA | AB | aA | a | bB | b

A → aA | a

B → bB | b

Example (2): S →Aa | B, A →a | bc | B, B → A | bb

Solution – Unit productions are

S → B, A → B, B → A, A and B are derivable

– Add productions from derivable and eliminate unit productions

S → bb | a | bc

A → a| bc | bb

B → bb | a | bc

Example (3) : Eliminate useless symbols, □□-productions and unit productions from S → a | aA|B|C, A → aB|□, B → aA, C → cCD, D → ddd

Soulution– Eliminate □□-productions

Nullable = {A}

P_1 = {S → a|aA|B|C, A → aB, B → aA|a, C → cCD, D → ddd}

-- Eliminate unit productions

Unit productions: S → B, S →C Derivable variables:B & C

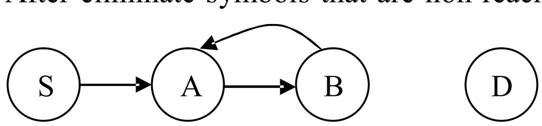
P_2 = {S → a|aA| cCD, A → aB, B → aA|a, C → cCD, D→ ddd}

– Eliminate useless symbols

• After eliminate non generating symbols

P_3 = {S → a|aA, A →aB, B → aA|a, D →ddd}

• After eliminate symbols that are non reachable



P_4 = {S → a|aA, A -->aB, B -->aA|a}

• So the equivalent grammar G1 = ({S,A,B}, {a}, {S -->a|aA, A -->aB, B -->aA|a}, S)

Simplified Grammar:

If you have to get a grammar without - productions, useless symbols and unit productions, follow the sequence given below:

1. Eliminate - productions from G and obtain G_1
2. Eliminate unit productions from G_1 and obtain G_2
3. Eliminate useless symbols from $G_2$and obtain G_3

Chomsky Normal Form (CNF)

• Every nonempty CFL without , has a grammar with productions of the form

1. A --> BC, where A, B, C V

2. A --> a, where A V and a T

• Algorithm:

1. Eliminate useless symbols, -productions and unit productions from the grammar

2. Elimination of terminals on RHS of a production

a) Add all productions of the form A --> BC or A --> a to P_1

b) Consider a production A -->$X_1X_2…X_n$ with some terminals of RHS. If X_i is a terminal say a_i, then add a new variable C_{ai} to V_1 and a new production C_{ai} -->ai to P_1. Replace Xi in A production of P by C_{ai}

c) Consider A -->$X_1X_2…X_n$, where n 3 and all X_i's are variables. Introduce new productions A -->X1C1, C_1-->X_2C_2, … , C_{n-2}-->$X_{n-1}X_n$ to P_1 and C_1, C_2, … ,C_{n-2} to V_1

Example (4): Convert to CNF:

S -->aAD, A --> aB | bAB, B -->b, D -->d

Solution – Step1: Simplify the grammar

• already simplified

– Step2a: Elimination of terminals on RHS

S -->aAD to S --> C_aAD, C_a-->a

A -->aB to A --> C_aB

A -->bAB to A --> C_bAB, C_b-->b

– Step2b: Reduce RHS with 2 variables

S --> C_aAD to S --> C_aC_1, C_1 -->AD

A --> C_bAB to A --> C_bC_2, C_2-->AB

• Grammar converted to CNF:

$G1=(\{S,A,B,D,C_a,C_b,C_1,C_2\}, \{a,b\},$

$\{S \text{-->} C_aC_1, A \text{-->} C_aB \mid C_bC_2, C_a\text{-->}a, C_b\text{-->}b, C_1 \text{-->}AD, C_2\text{-->}AB\}, S)$

Example (5): Convert to CNF:P={S -->ASB | □, A --> aAS | a, B -->SbS | A | bb}

Solution: – Step1: Simplify the grammar

- Eliminate □□-productions (S -->□)

  $P_1=\{S \text{-->}ASB|AB, A \text{-->}aAS|aA|a, B\text{-->}SbS|Sb|bS|b|A|bb\}$

- Eliminate unit productions (B-->A)

  $P_2=\{S \text{-->}ASB|AB, A \text{-->}aAS|aA|a, B\text{-->}SbS|Sb|bS|b|bb|aAS|aA|a\}$

- Eliminate useless symbols: no useless symbols

– Step2: Convert to CNF

$P_3=\{S \text{-->}AC_1|AB, A \text{-->} C_aC_2|C_aA|a, B \text{-->}SC_3 \mid SC_b \mid C_bS \mid b \mid C_bC_b| C_aC_2|C_aA|a, C_a\text{-->}a, C_b \text{-->}b, C_1 \text{-->}SB, C_2 \text{-->}AS, C_3 \text{-->} C_bS \}$

Exercises:

- Convert to CNF:

1. S -->aSa|bSb|a|b|aa|bb
2. S -->bA|aB, A -->bAA|aS|a, B -->aBB|bS|b
3. S-->Aba, A -->aab, B -->AC
4. S -->0A0|1B1|BB, A -->C, B -->S|A, C -->S| □
5. S -->aAa|bBb| □, A -->C|a, B -->C|b, C -->CDE|□, D -->A|B|ab

4.2:<u>The Pumping Lemma for CFL</u>

The *pumping lemma for regular languages* states that every sufficiently long string in a regular language contains a short sub-string that can be pumped. That is, inserting as many copies of the sub-string as we like always yields a string in the regular language.

The *pumping lemma for CFL's* states that there are always two short sub-strings close together that can be repeated, both the same number of times, as often as we like.

For example, consider a

CFL $L=\{a^nb^n \mid n$ □ $1\}$. Equivalent CNF grammar is having productions S □ AC | AB, A □ a, B □ b, C □ SB. The parse tree for the string a^4b^4

is given in figure 1. Both leftmost derivation and rightmost derivation have same parse tree because the grammar is unambiguous.

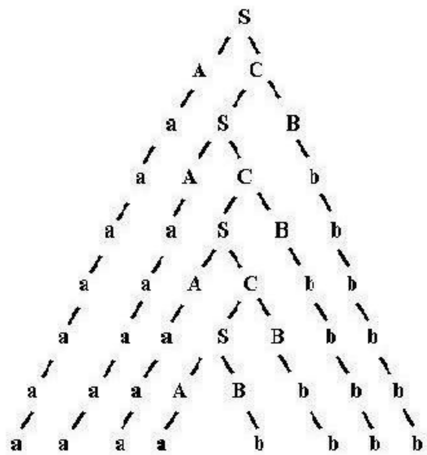


Figure 2: Extended Parse tree for
4 4

Figure : Parse tree for a^4b^4

Extend the tree by duplicating the terminals generated at each level on all lower levels. The extended parse tree for the string a^4b^4
is given in figure 2. Number of symbols at each level is at most twice of previous level. 1 symbols at level 0, 2 symbols at 1, 4 symbols at 2 …2^i symbols at level i. To have 2^n symbols at bottom level, tree must be having at least depth of n and level of at least n+1.

Pumping Lemma Theorem:

Let L be a CFL. Then there exists a constant k□ 0 such that if z is any string in L such that |z|

□ k, then we can write z = uvwxy such that

1. |vwx| □ k (that is, the middle portion is not too long).
2. vx □ □ (since v and x are the pieces to be "pumped", at least one of the strings we pump must not be empty).
3. For all i □ 0, uv^iwx^iy is in L.

Proof:

The parse tree for a grammar G in CNF will be a binary tree. Let $k = 2^{n+1}$, where n is the number of variables of G. Suppose z□ L(G) and |z| □ k. Any parse tree for z must be of depth at least n+1. The longest path in the parse tree is at least n+1, so this path must contain at least n+1 occurrences of the variables. By pigeonhole principle, some variables occur more than once along the path. Reading from bottom to top, consider the first pair of same variable along the path. Say X has 2 occurrences. Break z into uvwxy such that w is the string of terminals generated at the lower occurrence of X and vwx is the string generated by upper occurrence of X.

Example parse tree:

For the above example S has repeated occurrences, and the parse tree is shown in figure 3. w = ab is the string generated by lower occurrence of S and vwx = aabb is the string generated by upper occurrence of S. So here u=aa, v=a, w=ab, x=b, y=bb.

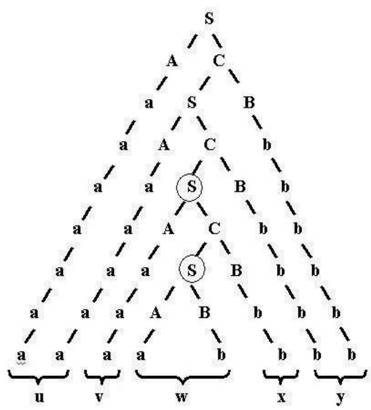


Figure 3: Parse tree for a^4b^4 with repeated occurrences of S

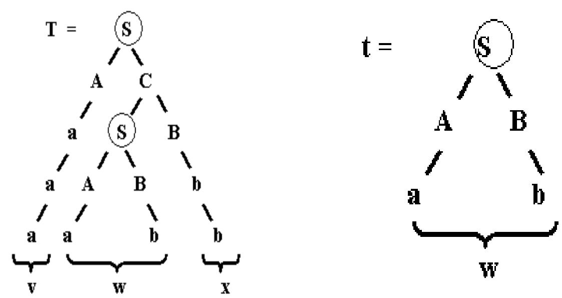


Figure 4: sub- trees

Let T be the subtree rooted at upper occurrence of S and t be subtree rooted at lower occurrence of S. These parse trees are shown in figure 4. To get $uv^2wx^2y \in L$, cut out t and replace it with copy of T. The parse tree for $uv^2wx^2y \in L$ is given in figure 5. Cutting out t and replacing it with copy of T as many times to get a valid parse tree for uv^iwx^iy for $i \geq 1$.

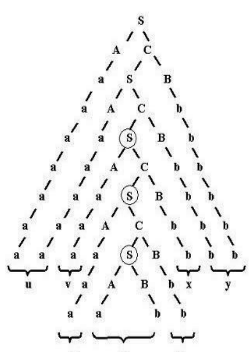


Figure 5: Parse tree

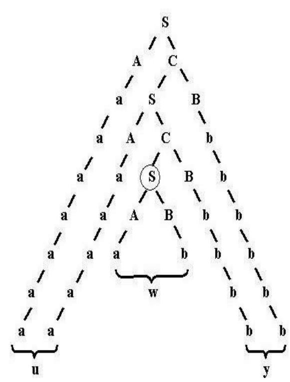


Figure 6: Parse tree for

To get $uwy \in L$, cut T out of the original tree and replace it with t to get a parse tree of uv^0wx^0y = uwy as shown in figure 6.

Pumping Lemma game:

1. To show that a language L is not a CFL, assume L is context free.
2. Choose an "appropriate" string z in L
3. Express z = uvwxy following rules of pumping lemma
4. Show that uv^kwx^ky is not in L, for some k
5. The above contradicts the Pumping Lemma
6. Our assumption that L is context free is wrong

Example:

Show that L = $\{a^ib^ic^i \mid i \square 1\}$ is not CFL

Solution:

Assume L is CFL. Choose an appropriate $z = a^nb^nc^n = uvwxy$. Since $|vwx| \square n$ then vwx can either consists of

1. All a's or all b's or all c's
2. Some a's and some b's
3. Some b's and some c's

Case 1: vwx consists of all a's

If $z = a^2b^2c^2$ and $u = \square$, $v = a$, $w = \square$, $x = a$ and $y = b^2c^2$ then, uv^2wx^2y will be $a^4b^2c^2 \square L$

Case 2: vwx consists of some a's and some b's

If $z = a^2b^2c^2$ and $u = a$, $v = a$, $w = \square$, $x = b$, $y = bc^2$, then uv^2wx^2y will be $a^3b^3c^2 \square L$

Case 3: vwx consists of some b's and some c's

If $z = a^2b^2c^2$ and $u = a^2b$, $v = b$, $w = c$, $x = \square$, $y = c$, then uv^2wx^2y will be $a^2b^3c^2 \square L$

If you consider any of the above 3 cases, uv^2wx^2y will not be having an equal number of a's, b's and c's. But Pumping Lemma says $uv^2wx^2y \square L$. Can't contradict the pumping lemma! Our original assumption must be wrong. So L is not context-free.

Example:

Show that L = $\{ww \mid w \square \{0, 1\}^*\}$ is not CFL

Solution:

Assume L is CFL. It is sufficient to show that $L1= \{0^m1^n0^m1^n \mid m,n \geq 0\}$, where n is pumping lemma constant, is a CFL. Pick any $z = 0^n1^n0^n1^n = uvwxy$, satisfying the conditions $|vwx| \leq n$ and $vx \neq \epsilon$.

This language we prove by taking the case of i = 0, in the pumping lemma satisfying the condition uv^iwx^iy for $i \geq 0$.

z is having a length of 4n. So if $|vwx| \leq n$, then $|uwy| \geq 3n$. According to pumping lemma, $uwy \in L$. Then uwy will be some string in the form of tt, where t is repeating. If so, n $|t| \geq$ 3n/2.

Suppose vwx is within first n 0's: let vx consists of k 0's. Then uwy begins with $0^{n-k}1^n$

$|uwy| = 4n-k$. If uwy is some repeating string tt, then $|t| = 2n-k/2$. t does end in 0 but tt ends with 1. So second t is not a repetition of first t.

Example: $z = 0^31^30^31^3$, $vx = 0^2$ then $uwy = tt = 01^30^31^3$, so first $t = 01^30$ and second $t = 0^21^3$. Both t's are not same.

Suppose vwx consists of 1st block of 0's and first block of 1's: vx consists of only 0's if $x= \epsilon$, then uwy is not in the form tt. If vx has at least one 1, then |t| is at least 3n/2 and first t ends with a 0, not a 1.

Very similar explanations could be given for the cases of vwx consists of first block of 1's and vwx consists of 1st block of 1's and 2nd block of 0's. In all cases uwy is expected to be in the form of tt. But first t and second t are not the same string. So uwy is not in L and L is not context free.

Example:

Show that $L=\{0^i1^j2^i3^j \mid i \geq 1, j \geq 1\}$ is not CFL

Solution:

Assume L is CFL. Pick $z = uvwxy = 0^n1^n2^n3^n$ where $|vwx| \leq n$ and $vx \neq \epsilon$
. vwx can consist of a substring of one of the symbols or straddles of two adjacent symbols.

Case 1: vwx consists of a substring of one of the symbols

Then uwy has n of 3 different symbols and fewer than n of 4^{th} symbol. Then uwy is not in L.

Case 2: vwx consists of 2 adjacent symbols say 1 & 2

Then uwy is missing some 1's or 2's and uwy is not in L.
If we consider any combinations of above cases, we get uwy, which is not CFL. This

contradicts the assumption. So L is not a CFL.

4.3:<u>Closure Properties of CFL</u>

Many operations on Context Free Languages (CFL) guarantee to produce CFL. A few do not produce CFL. *Closure properties* consider operations on CFL that are guaranteed to produce a CFL. The CFL's are closed under *substitution*, *union*, *concatenation*, *closure (star)*, *reversal*, *homomorphism* and *inverse homomorphism*. CFL's are not closed under *intersection* (but the intersection of a CFL and a regular language is always a CFL), *complementation*, and *set-difference*.

I. Substitution:

By substitution operation, each symbol in the strings of one language is replaced by an entire CFL language
.
Example:

$S(0) = \{a^nb^n \mid n \geq 1\}$, $S(1)=\{aa,bb\}$ is a substitution on alphabet $\Sigma =\{0, 1\}$.

Theorem:

If a substitution s assigns a CFL to every symbol in the alphabet of a CFL L, then s(L) is a CFL.

Proof:

Let G = (V, □, P, S) be grammar for the CFL L. Let G_a = (V_a, T_a, P_a, S_a) be the grammar corresponding to each terminal a □ □ and V □ V_a = □. Then G□= (V□, T□, P□, S) is a grammar for s(L) where

- V□ = V □ V_a
- T□= union of T_a's all for a □ □
- P□ consists of
  - All productions in any P_a for a □ □
  - The productions of P, with each terminal a is replaced by S_a everywhere a occurs.

Example:

L = {0^n1^n| n □ 1}, generated by the grammar S □ 0S1 | 01, s(0) = {a^nb^m | m □ n}, generated by the grammar S □ aSb | A; A □ aA | ab, s(1) = {ab, abc}, generated by the grammar S □ abA, A □ c |□

. Rename second and third S's to S_0 and S_1, respectively. Rename second A to B. Resulting grammars are:

S □ 0S1 | 01

S_0 □ aS_0b | A; A □ aA | ab

S_1 □ abB; B □ c | □

In the first grammar replace 0 by S_0 and 1 by S_1. The resulted grammar after substitution is:

S □ S_0SS_1 | S_0S_1

S_0□ aS_0b | A; A □aA | ab S_1□abB; B□ c | □

II. Application of substitution:

a. Closure under union of CFL's L_1 and L_2:

Use L={a, b}, s(a)=L_1 and s(b)=L_2. Then s(L)= L_1 □ L_2.

How t

o get grammar for L_1 □ L_2 ?

Add new start symbol S and rules S □ S_1 | S_2

The grammar for L_1 □ L_2 is G = (V, T, P, S) where V = {V_1 □ V_2 □ S}, S□ (V_1 □ V_2)
and P = {P_1 □ P_2 □ {S □ S_1 | S_2 }}

Example:

L_1 = {$a^n b^n$ | n □ 0}, L_2 = {$b^n a^n$ | n □ 0}. Their corresponding grammars are

G_1: S_1 □ aS_1b | □, G_2 : S_2 □ bS_2a | □

The grammar for L_1 □ L_2 is

G = ({S, S_1, S_2}, {a, b}, {S □ S_1 | S_2, S_1 □ aS_1b | □, S_2 □ bS_2a}, S).

b. Closure under concatenation of CFL's L_1 and L_2:

Let L={ab}, s(a)=L_1 and s(b)=L_2. Then s(L)=L_1L_2

How to get grammar for L 1L 2?

Add new start symbol and rule S □ S1S2

The grammar for L_1L_2 is G = (V, T, P, S) where V = V_1 □ V_2 □ {S}, S □ V_1 □ V_2
and P = P_1 □ P_2 □ {S □ S 1S 2}

Example:

L1 = {$a^n b^n$ | n □ 0}, L2 = {$b^n a^n$ | n □ 0} then L 1L 2= {$a^n b^{\{n+m\}} a^m$ | n, m □ 0}

Their corresponding grammars are
G_1: S_1 □ aS_1b | □, G_2 : S_2 □ bS_2a | □

The grammar for L_1L_2 is
G = ({S, S_1, S_2}, {a, b}, {S □ S_1S_2, S_1 □ aS_1b | □, S_2 □ bS_2a}, S).

c. Closure under Kleene's star (closure * and positive closure $^+$) of CFL's L_1:

Let L = {a}* (or L = $\{a\}^+$) and s(a) = L_1. Then s(L) = L_1* (or s(L) = L_1^+).

Example:

L_1 = {$a^n b^n$ | n □ 0} (L_1)* = {$a^{\{n1\}} b^{\{n1\}}$ ... $a^{\{nk\}} b^{\{nk\}}$ | k □ 0 and n*i* □ 0 for all *i*}

$L_2 = \{a^{\{n2\}} \mid n \geq 1\}$, $(L_2)^* = a^*$

How t
o get grammar for $(L_1)^*$:

Add new start symbol S and rules $S \rightarrow SS_1 \mid \varepsilon$.

The grammar for $(L_1)^*$ is

$G = (V, T, P, S)$, where $V = V_1 \cup \{S\}$, $S \notin V_{1,}$

$P= P_1 \cup \{S \rightarrow SS_1 \mid \varepsilon\}$

d. Closure under homomorphism of CFL L_i for every $a_i \in \Sigma$:

Suppose L is a CFL over alphabet Σ and h is a homomorphism on Σ. Let s be a substitution that replaces every $a \in \Sigma$, by h(a). ie $s(a) = \{h(a)\}$. Then $h(L) = s(L)$. ie $h(L) =\{h(a_1)\ldots h(a_k) \mid k \geq 0\}$ where $h(a_i)$ is a homomorphism for every $a_i \in \Sigma$.

III. Closure under

IV. Reversal:

L is a CFL, so L^R is a CFL. It is enough to reverse each production of a CFL for L, i.e., to substitute each production $A \rightarrow \alpha$ by $A \rightarrow \alpha^R$.

IV. Intersection:

The CFL's are not closed under intersection

Example:

The language $L = \{0^n1^n2^n \mid n \geq 1\}$ is not context-free. But $L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$ is a CFL and $L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$ is also a CFL. But $L = L1 \cap L_2$.

Corresponding grammars for L_1: $S \rightarrow AB$; $A \rightarrow 0A1 \mid 01$; $B \rightarrow 2B \mid 2$ and corresponding grammars for L_2: $S \rightarrow AB$; $A \rightarrow 0A \mid 0$; $B \rightarrow 1B2 \mid 12$.

However, $L = L_1 \cap L2$, thus intersection of CFL's is not CFL

a. CFL and Regular Language:

Theorem: If L is CFL and R is a regular language, then L □ R is a CFL.

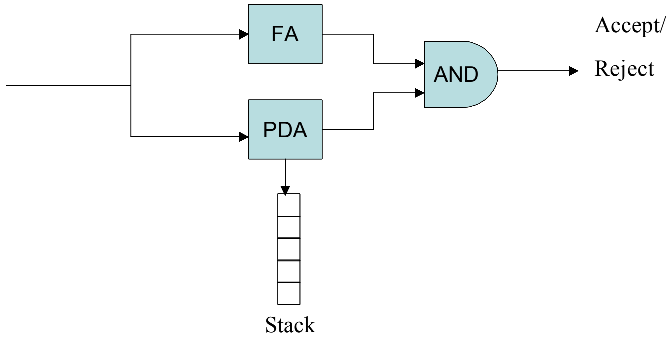


Figure 1: PDA for $L \cap R$

Proof:

P = (Q_P, □, □, $□_P$, q_P, Z_0, F_P) be PDA to accept L by final state. Let A = (Q_A, □, $□_A$, q_A, F_A) for DFSM to accept the Regular Language R. To get L □ R, we have to run a Finite Automata
in parallel with a push down automata as shown in figure 1. Construct PDA P□ = (Q, □, □, □, q_o, Z_0, F) where

- Q = (Q_p X Q_A)
- q_o = (q_p, q_A)
- F = (F_PX F_A)
- □ is in the form □ ((q, p), a, X) = ((r, s), g) such that
  1. s = $□_A$(p, a)
  2. (r, g) is in $□_P$(q, a, X)

That is for each move of PDA P, we make the same move in PDA P□ and also we carry along the state of DFSM A in a second component of P□. P□ accepts a string w if and only if
both P and A accept w. ie w is in L □ R. The moves ((q_p, q_A), w, Z) |-*P□ ((q, p), □, □) are possible if and only if (q_p, w, Z) |-*P (q, □,□) moves and p = □*(q_A, w) transitions are possible.

CFL and RL properties:

Theorem: The following are true about CFL's L, L_1, and L_2, and a regular language R.

1. **Closure of CFL's under set-difference with a regular language**.
2.

ie

1. L - R is a CFL.

   Proof:

R is regular and regular language is closed under complement. So R^C is also regular. We know that $L - R = L \cap R^C$. We have already proved the closure of intersection of a CFL and a regular language. So CFL is closed under set difference with a Regular language.

2. CFL is not closed under complementation

L^C is not necessarily a CFL

Proof:

Assume that CFLs were closed under complement. ie if L is a CFL then L^C is a CFL. Since CFLs are closed under union, $L_1^C \cup L_2^C$ is a CFL. By our assumption $(L_1^C \cup L_2^C)^C$ is a CFL. But $(L_1^C \cup L_2^C)^C = L_1 \cap L_2$, which we just showed isn't necessarily a CFL. Contradiction! . So our assumption is false. CFL is not closed under complementation.

CFLs are not closed under set-difference.

ie
$L_1 - L_2$ is not necessarily a CFL.

Proof:

Let $L1 = \Sigma^* - L$. Σ^* is regular and is also CFL. But $\Sigma^* - L = L^C$. If CFLs were closed under set difference, then $\Sigma^* - L = L^C$ would always be a CFL. But CFL's are not closed under complementation. So CFLs are not closed under set-difference.

MODULE -5

INTRODUCTION TO TURING MACHINES

5.1 problems that computers cannot solve

5.2 The turing machine

5.3programming techniques for turing machines

5.4 extensions to the basic turing machines

5.5 turing machines and computers

5.6: A language that is not recursively enumerable

5.7: An undecidable problem that is RE

5.8: Post correspondence problem

5.9: Other undecidable problem

5.1 The Turing machine

Definition:

A Turing Machine (TM) is an abstract, mathematical model that describes what can and cannot be computed. A Turing Machine consists of a tape of infinite length, on which input is provided as a finite sequence of symbols. A *head* reads the input tape. The Turing Machine starts at "start state" S_0. On reading an input symbol it optionally replaces it with another symbol, changes its internal state and moves one cell to the right or left.

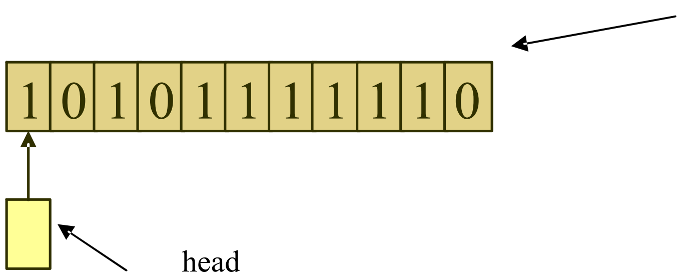
Notation for the Turing Machine :

TM = <S, T, S_0, □, H> where,

S	is a set of TM states
T	is a set of tape symbols
S_0	is the start state
H □ S	is a set of halting states
□ : S x T □S x T x {L,R}	is the transition function
{L,R}	is direction in which the head moves

L : Left R: Right

input symbols on infinite length tape



head

The Turing machine model uses an infinite tape as its unlimited memory. (This is important because it helps to show that there are tasks that these machines cannot perform, even though unlimited memory and unlimited time is given.) The input symbols occupy some of the tape's cells, and other cells contain blank symbols.

Some of the characteristics of a Turing machine are:

1. The symbols can be both read from the tape and written on it.
2. The TM head can move in either directions – Left or Right.
3. The tape is of infinite length
4. The special states, Halting states and Accepting states, take immediate effect.

Solved examples:

<u>TM Example 1:</u>

Turing Machine U+1:

Given a string of 1s on a tape (followed by an infinite number of 0s), add one more 1 at the end of the string.

Input : #111100000000…….
□□□□□□□
Output : #1111100000000……….

Initially the TM is in Start state S_0. Move right as long as the input symbol is 1. When a 0 is encountered, replace it with 1 and halt.
Transitions:
$(S_0, 1) \longrightarrow (S_0, 1, R)$
$(S_0, 0) \longrightarrow (h , 1, STOP)$

<u>TM Example 2 :</u>

TM: X-Y
Given two unary numbers x and y, compute |x-y| using a TM. For purposes of simplicity we shall be using multiple tape symbols.

Ex: 5 (11111) – 3 (111) = 2 (11)
#11111b1110000….. □
#___11b___000…

a) Stamp out the first 1 of x and seek the first 1 of y.

$(S_0, 1) \longrightarrow (S_1, \_, R)$
$(S_0, b) \longrightarrow (h, b, STOP)$
$(S_1, 1) \longrightarrow (S_1, 1, R)$
$(S_1, b) \longrightarrow (S_2, b, R)$

b) Once the first 1 of y is reached, stamp it out. If instead the input ends, then y has finished. But in x, we have stamped out one extra 1, which we should replace. So, go to some state s5 which can handle this.

$(S_2, 1) \rightarrow (S_3, \_, L)$
$(S_2,\_) \rightarrow (S_2, \_, R)$
$(S_2, 0) \rightarrow (S_5, 0, L)$

c) State s3 is when corresponding 1s from both x and y have been stamped out. Now go back to x to find the next 1 to stamp. While searching for the next 1 from x, if we reach the head of tape, then stop.

$(S_3, \_) \rightarrow (S_3, \_, L)$
$(S_3,b) \rightarrow (S_4, b, L)$
$(S_4, 1) \rightarrow (S_4, 1, L)$
$(S_4, \_) \rightarrow (S_0, \_, R)$
$(S_4, \#) \rightarrow (h, \#, STOP)$

d) State s5 is when y ended while we were looking for a 1 to stamp. This means we have stamped out one extra 1 in x. So, go back to x, and replace the blank character with 1 and stop the process.

$(S_5, \_) \rightarrow (S_5, \_, L)$
$(S_5,b) \rightarrow (S_6, b, L)$
$(S_6, 1) \rightarrow (S_6, 1, L)$
$(S_6, \_) \rightarrow (h, 1, STOP)$

Solved examples:

TM Example 1: Design a Turing Machine to recognize $0^n1^n2^n$
ex: #000111222_ _ _ _ _.......

Step 1: Stamp the first 0 with X, then seek the first 1 and stamp it with Y, and then seek the first 2 and stamp it with Z and then move left.

$$(S_0, 0) \rightarrow (S_1, X, R)$$
$$(S_1, 0) \rightarrow (S_1, 0, R)$$
$$(S_1, 1) \rightarrow (S_2, Y, R)$$
$$(S_2, 1) \rightarrow (S_2, 1, R)$$
$$(S_2, 2) \rightarrow (S_3, Z, L)$$

S_0 = Start State, seeking 0, stamp it with X
S1 = Seeking 1, stamp it with Y
S2 = Seeking 2, stamp it with Z

Step 2: Move left until an X is reached, then move one step right.

$$(S_3, 1) \rightarrow (S_3, 1, L)$$
$$(S_3, Y) \rightarrow (S_3, Y, L)$$
$$(S_3, 0) \rightarrow (S_3, 0, L)$$
$$(S_3, X) \rightarrow (S_0, X, R)$$

S3 = Seeking X, to repeat the process.

Step 3:Move right until the end of the input denoted by blank(_) is reached passing through X Y Z s only, then the accepting state S_A is reached.

$$(S_0, Y) \rightarrow (S_4, Y, R)$$
$$(S_4, Y) \rightarrow (S_4, Y, R)$$
$$(S_4, Z) \rightarrow (S_4, Z, R)$$
$$(S_4, \ ) \rightarrow (S_A, STOP)$$

S4 = Seeking blank

These are the transitions that result in halting states.

$$(S_4, 1) \rightarrow (h, 1, STOP)$$
$$(S_4, 2) \rightarrow (h, 2, STOP)$$
$$(S_4, \ ) \rightarrow (S_A, STOP)$$
$$(S_0, 1) \rightarrow (h, 1, STOP)$$
$$(S_0, 2) \rightarrow (h, 2, STOP)$$
$$(S_1, 2) \rightarrow (h, 2, STOP)$$
$$(S_2, \ ) \rightarrow (h, STOP)$$

<u>TM Example 2 :</u> Design a Turing machine to accept a Palindrome

ex: #1011101_ _ _ _ _…….

Step 1: Stamp the first character (0/1) with _, then seek the last character by moving till a _ is reached. If the last character is not 0/1 (as required) then halt the process immediately.

$$\delta(S_0, 0) \rightarrow (S_1, , R)$$
$$\delta(S_0, 1) \rightarrow (S_2, , R)$$
$$\delta(S_1,) \rightarrow (S_3, , L)$$
$$\delta(S_3, 1) \rightarrow (h, 1, STOP)$$
$$\delta(S_2,) \rightarrow (S_5, , L)$$
$$\delta(S_5, 0) \rightarrow (h, 0, STOP)$$

Step 2: If the last character is 0/1 accordingly, then move left until a blank is reached to start the process again.

$$(S_3, 0) \rightarrow (S_4, , L)$$
$$\delta(S_4, 1) \rightarrow (S_4, 1, L)$$
$$\delta(S_4, 0) \rightarrow (S_4, 0, L)$$
$$\delta(S_4,) \rightarrow (S_0, , R)$$
$$\delta(S_5, 1) \rightarrow (S_6, , L)$$
$$\delta(S_6, 1) \rightarrow (S_6, 1, L)$$
$$\delta(S_6, 0) \rightarrow (S_6, 0, L)$$
$$\delta(S_6,) \rightarrow (S_0, , R)$$

Step 3 : If a blank (_) is reached when seeking next pair of characters to match or when seeking a matching character, then accepting state is reached.

$$\delta(S_3,) \rightarrow \delta(A, , STOP)$$
$$\delta(S_5,) \rightarrow \delta(A, , STOP)$$
$$\delta(S_0,) \rightarrow \delta(A, , STOP)$$

The sequence of events for the above given input are as follows:

$\#s_0 10101\_\ \_\ \_$

$\#\_s_2 0101\_\ \_\ \_$

$\#\_0s_2 101\_\ \_\ \_$

. . . .

$\#\_0101s_5\_\ \_\ \_$

$\#\_010s_6\_\ \_\ \_\ \_$

$\#\_s_6 0101\_\ \_\ \_$

$\#\_s_0 0101\_\ \_\ \_$

. . . .

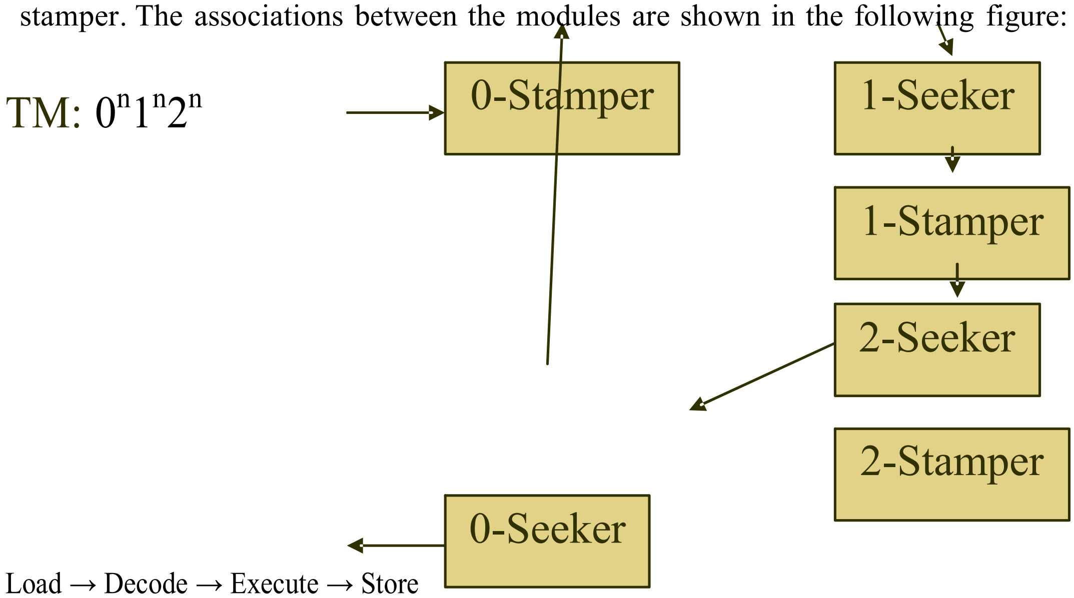
$\#\_\ \_\ \_\ \_\ s_5\ \_\ \_\ \_\ \_\ \_\ \_$

$\#\_\ \_\ \_\ \_\ s_A\ \_\ \_\ \_\ \_\ \_\ \_$

Modularization of TMs

Designing complex TM s can be done using modular approach. The main problem can be divided into sequence of modules. Inside each module, there could be several state transitions.

For example, the problem of designing Turing machine to recognize the language $0^n1^n2^n$ can be divided into modules such as 0-stamper, 1-stamper, 0-seeker, 1-seeker, 2-seeker and 2-stamper. The associations between the modules are shown in the following figure:



TM = (S,S0,H,T,d)
Suppose, S={a,b,c,d}, S0=a, H={b,d} T={0,1}
δ : (a,0) (b,1,R) , (a,1) (c,1,R) ,
(c,0) (d,0,R) and so on
then TM spec:
$abcd$abd01$a0b1Ra1c1Rc0d0R…… .
where $ is delimiter

This spec along with the actual input data would be the input to the UTM.
This can be encoded in binary by assigning numbers to each of the characters appearing in the TM spec.

The encoding can be as follows:

$: 0000	0 : 0101
a : 0001	1 : 0110
b : 0010	L : 0111
c : 0011	R : 1000
d : 0100	

So the TM spec given in previous slide can be encoded as:
0000.0001.0010.0011.0100.0000.0001.0000.0010.0100 ……
Hence TM spec can be regarded just as a number.

Sequence of actions in UTM:
Initially UTM is in the start state S0.

- Load the input which is TM spec.
- Go back and find which transition to apply.
- Make changes, where necessary.
- Then store the changes.
- Then repeat the steps with next input.

Hence, the sequence goes through the cycle:

→ Load → Decode → Execute → Store

5.3:Extensions to Turing Machines

Proving Equivalence

For any two machines M_1 from class C_1 and M_2 from class C_2:

M_2 is said to be at least as expressive as M_1
if $L(M_2) = L(M_1)$ or if M_2 can *simulate* M_1.

M_1 is said to be at least as expressive as M_2

if $L(M_1) = L(M_2)$ or if M_1 can simulate M_2.

Composite Tape TMs

Track 0	0	1	1	0	1	0	1	0	0 …
Track 1	0	0	1	1	1	1	1	1	0 …

A composite tape consists of many *tracks* which can be read or written *simultaneously.*

A composite tape TM (CTM) contains more than one tracks in its tape.

Equivalence of CTMs and TMs

A CTM is simply a TM with a complex alphabet..

T = {a, b, c, d}
T' = {00, 01, 10, 11}

Turing Machines with Stay Option

Turing Machines with stay option has a third option for movement of the TM head: left, right or *stay*.

STM = <S, T, □, s_0, H>

□: S x T à S x T x {L, R, S}

Equivalence of STMs and TMs

STM = TM:
Just don't use the S option…

TM = STM:

For L and R moves of a given STM build a TM that moves correspondingly L or R…

TM = STM:

For S moves of the STM, do the following:
1.Move right,
2.Move back left without changing the tape
3.STM: □(s,a) |-- (s',b,S)

TM: □(s,a) |-- (s'', b, R)
□(s'',*) |-- (s',*,L)

2-way Infinite Turing Machine

In a 2-way infinite TM (2TM), the tape is infinite on both sides. There is no # that delimits the left end of the tape.

Equivalence of 2TMs and TMs

2TM = TM:
Just don't use the left part of the tape…

TM = 2TM:
Simulate a 2-way infinite tape on a one-way infinite tape…

… -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 …

0 –1 1 –2 2 –3 3 –4 4 –5 5 …

Multi-tape Turing Machines

A multi-tape TM (MTM) utilizes many tapes.

Equivalence of MTMs and TMs

MTM = TM:
Use just the first tape…

TM = MTM:
Reduction of multiple tapes to a single tape.

Consider an MTM having *m* tapes. A single tape TM that is equivalent can be constructed by reducing *m* tapes to a single tape.

A 0 1 2 3 4 5 6 7 …

B 0 1 2 3 4 5 6 7 …

C 0 1 2 3 4 5 6 7 …

TM A0 B0 C0 A1 B1 C1 A2 B2 C2 A3 B3 ..

Non-deterministic TM

A non-deterministic TM (NTM) is defined as:

NTM = <S, T, s_0, □, H>

where □: S x T à$2^{SxTx\{L,R\}}$

Ex: (s_2,a) à {(s_3,b,L) (s_4,a,R)}

Equivalence of NTMs and TMs

A “concurrent” view of an NTM:

(s_2,a) à {(s_3,b,L) (s_4,a,R)}

è at (s_2,a), two TMs are spawned:

(s_2,a) à (s_3,b,L)

(s_2,a) à (s_4,a,R)

5.5: A language that is not recursively enumerable

Decidable

A problem P is *decidable* if it can be solved by a Turing machine T that always halt. (We say that P has an effective algorithm.)

Note that the corresponding language of a decidable problem is *recursive*.

Undecidable

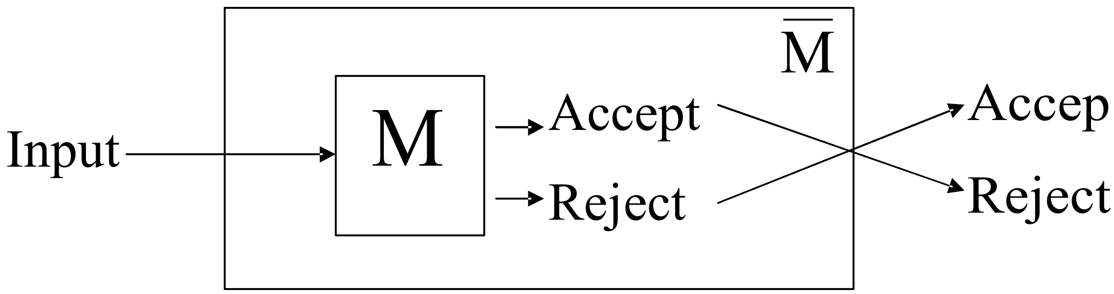
A problem is *undecidable* if it cannot be solved by any Turing machine that halts on all inputs.

Note that the corresponding language of an undecidable problem is *non-recursive*.

Complements of Recursive Languages

Theorem: If L is a recursive language, L is also recursive.

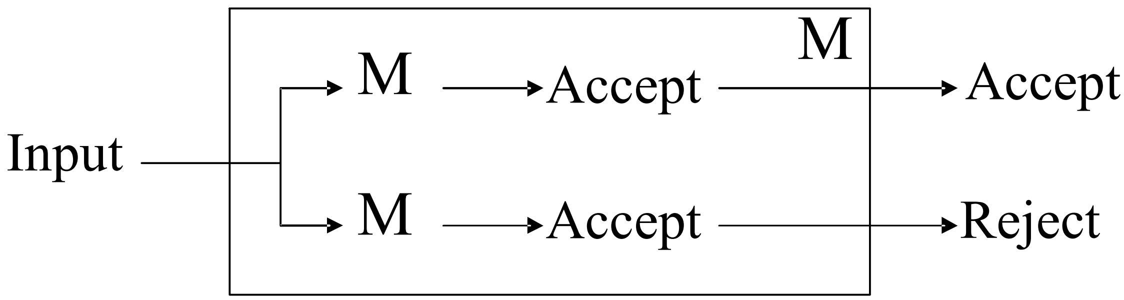
Proof: Let M be a TM for L that always halt. We can construct another TM M from M for L that always halts as follows:



Complements of RE Languages

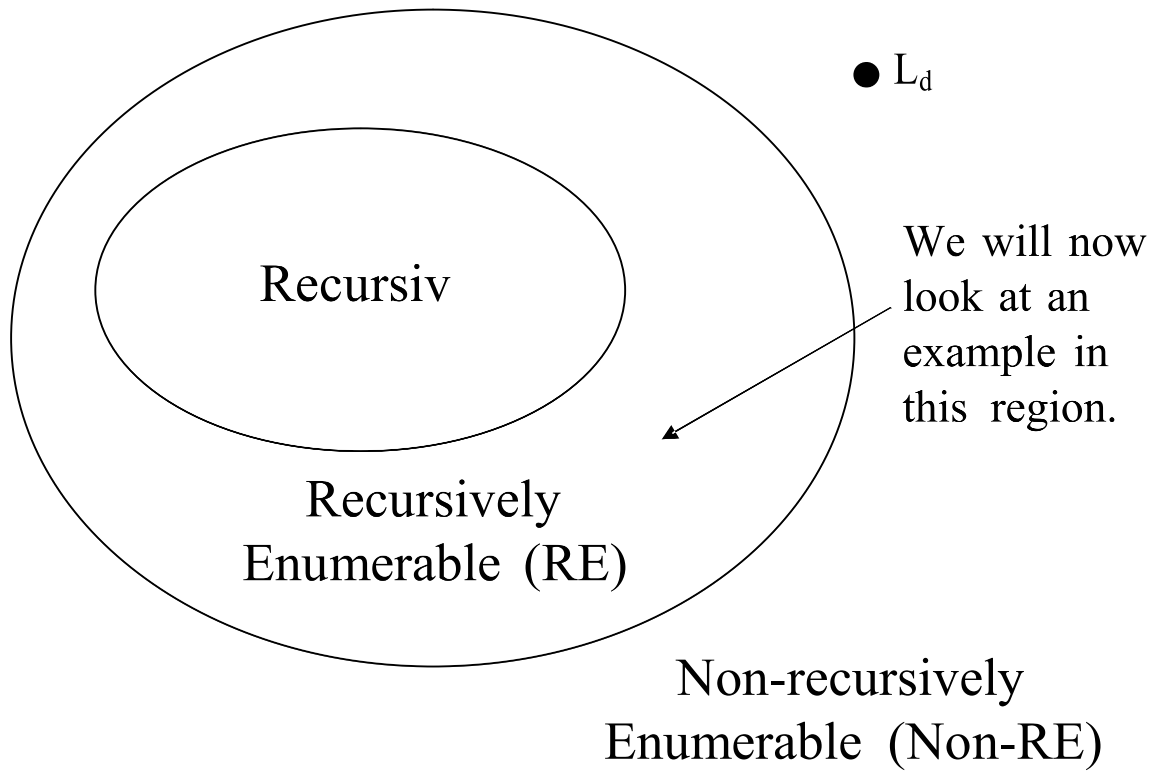
Theorem: If both a language L and its complement L are RE, L is recursive.

Proof: Let M1 and M2 be TM for L and L respectively. We can construct a TM M from M1 and M2 for L that always halt as follows:



A Non-recursive RE Language

- We are going to give an example of a RE language that is not recursive, i.e., a language L that can be accepted by a TM, but there is no TM for L that always halt.
- Again, we need to make use of the binary encoding of a TM.



A Non-recursive RE Language

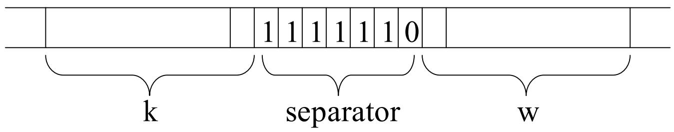
- Recall that we can encode each TM uniquely as a binary number and enumerate all TM's as T1, T2, …, Tk, … where the encoded value of the kth TM, i.e., Tk, is k.
- Consider the language Lu:
  Lu = {(k, w) | Tk accepts input w}
  This is called the *universal language*.

Universal Language

- Note that designing a TM to recognize Lu is the same as solving the problem of *given k and w, decide whether Tk accepts w as its input*.
- We are going to show that Lu is RE but non-recursive, i.e., Lu can be accepted by a TM, but there is no TM for Lu that always halt.

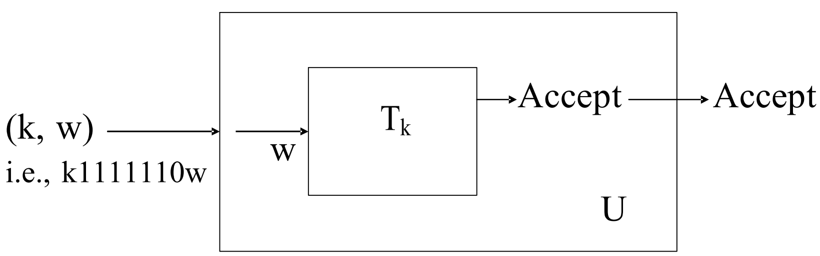
Universal Turing Machine

- To show that L_u is RE, we construct a TM U, called the *universal Turing machine*, such that $L_u = L(U)$.
- U is designed in such a way that given k and w, it will mimic the operation of T_k on input w:



U will move back and forth to mimic T_k on input w.

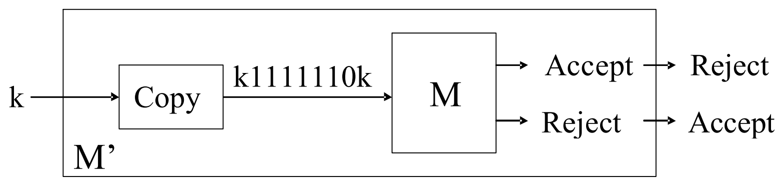
Universal Turing Machine



Why cannot we use a similar method to construct a TM for L_d?

Universal Language

- Since there is a TM that accepts L_u, L_u is RE. We are going to show that L_u is non-recursive.
- If L_u is recursive, there is a TM M for L_u that always halt. Then, we can construct a TM M’ for L_d as follows:



A Non-recursive RE Language

- Since we have already shown that Ld is non-recursively enumerable, so M’ does not exist and there is no such M.
- Therefore the universal language is recursively enumerable but non-recursive.

Halting Problem

Consider the halting problem:

Given (k,w), determine if Tk halts on w.

It’s corresponding language is:

Lh = { (k, w) | Tk halts on input w}

The halting problem is also undecidable, i.e., Lh is non-recursive. To show this, →we can make use of the universal language problem.

We want to show that if the halting problem can be solved (decidable), the universal language problem can also be solved.

→So we will try to reduce an instance (a particular problem) in Lu to an instance in Lh in such a way that if we know the answer for the latter, we will know the answer for the former.

Class Discussion

Consider a particular instance (k,w) in Lu, i.e., we want to determine if Tk will accept w. <u>Construct</u> an instance I=(k’,w’) in Lh from (k,w) so that if we know whether Tk’ will halt on w’, we will know whether Tk will accept w.

Halting Problem

Therefore, if we have a method to solve the halting problem, we can also solve the universal language problem. (Since for any particular instance I of the universal language problem, we can construct an instance of the halting problem, solve it and get the answer for I.) However, since the universal problem is undecidable, we can conclude that the halting problem is also undecidable.

Modified Post Correspondence Problem

- We have seen an undecidable problem, that is, given a Turing machine M and an input w, determine whether M will accept w (universal language problem).
- We will study another undecidable problem that is not related to Turing machine directly.

  Given two lists A and B:

  A = w1, w2, …, wk   B = x1, x2, …, xk

  The problem is to determine if there is a sequence of one or more integers i1, i2, …, im such that:

  w1wi1wi2…wim = x1xi1xi2…xim

  (wi, xi) is called a corresponding pair.

Example

	A	B
i	w_i	x_i
1	11	1
2	1	111
3	0111	10
4	10	0

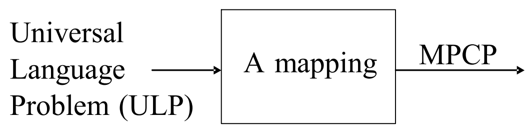
This MPCP instance has a solution: 3, 2, 2, 4:

$w_1w_3w_2w_2w_4 = x_1x_3x_2x_2x_4 = 1101111110$

5.6: An undecidable problem that is RE

Undecidability of PCP

To show that MPCP is undecidable, we will reduce the universal language problem (ULP) to MPCP:

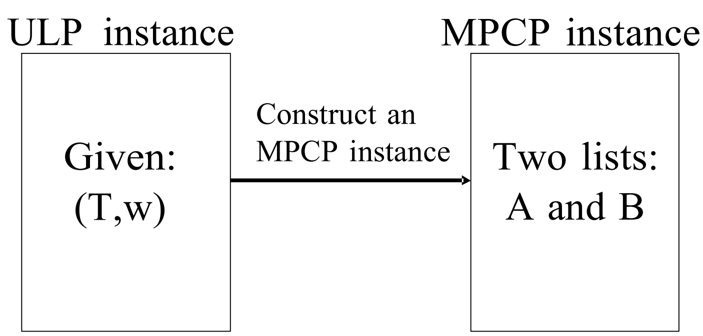


If MPCP can be solved, ULP can also be solved. Since we have already shown that ULP is undecidable, MPCP must also be undecidable.

Mapping ULP to MPCP

- Mapping a universal language problem instance to an MPCP instance is not as easy.
- In a ULP instance, we are given a Turing machine M and an input w, we want to determine if M will accept w. To map a ULP instance to an MPCP instance successfully, the mapped MPCP instance should have a solution if and only if M accepts w.

Mapping ULP to MPCP



If T accepts w, the two lists can be matched. Otherwise, the two lists cannot be matched.

Mapping ULP to MPCP

- We assume that the input Turing machine T:
  - Never prints a blank

  - Never moves left from its initial head position.
- These assumptions can be made because:
  - **Theorem** (p.346 in Textbook): Every language accepted by a TM M2 will also be accepted by a TM M1 with the following restrictions: (1) M1's head never moves left from its initial position. (2) M1 never writes a blank.

Mapping ULP to MPCP

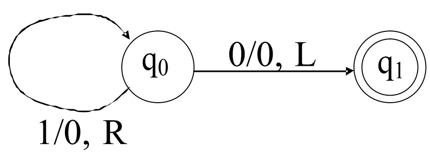
Given T and w, the idea is to map the transition function of T to strings in the two lists in such a way that a matching of the two lists will correspond to a concatenation of the tape contents at each time step.

We will illustrate this with an example first.

Example of ULP to MPCP

- Consider the following Turing machine:

$T = (\{q_0, q_1\},\{0,1\},\{0,1,\#\}, \delta, q_0, \#, \{q_1\})$



$\delta(q_0,1)=(q_0,0,R)$ $\delta(q_0,0)=(q_1,0,L)$

- Consider input w=110.

Example of ULP to MPCP

- Now we will construct an MPCP instance from T and w. There are <u>five</u> types of strings in list A and B:
- Starting string (<u>first pair</u>):

List A	List B
#	#$q_0$110#

Example of ULP to MPCP

- Strings from the transition function δ:

List A	List B	
$q_0$1	0q_0	(from $\delta(q_0,1)=(q_0,0,R)$)
0$q_0$0	$q_1$00	(from $\delta(q_0,0)=(q_1,0,L)$)
1$q_0$0	$q_1$10	(from $\delta(q_0,0)=(q_1,0,L)$)

Example of ULP to MPCP

- Strings for copying:

List A	List B
#	#
0	0
1	1

Example of ULP to MPCP

- Strings for consuming the tape symbols at the end:

List A	List B	List A	List B
0q1	q1	0q11	q1
1q1	q1	1q10	q1
q10	q1	0q10	q1
q11	q1	1q10	q1

Class Discussion

Consider the input w = 101. Construct the corresponding MPCP instance I and show that T will accept w by giving a solution to I.

Class Discussion (cont'd)

	List A	List B		List A	List B
1.	#	$\#q_0101\#$	9.	$0q_1$	q_1
2.	q_01	$0q_0$	10.	$1q_1$	q_1
3.	$0q_00$	q_100	11.	q_10	q_1
4.	$1q_00$	q_110	12.	q_11	q_1
5.	#	#	13.	$0q_11$	q_1
6.	0	0	14.	$1q_10$	q_1
7.	1	1	15.	$0q_10$	q_1
8.	$q_1\#\#$	#	16.	$1q_10$	q_1

Mapping ULP to MPCP

- We summarize the mapping as follows. Given T and w, there are five types of strings in list A and B:
- Starting string (first pair):

List A	List B
#	#q0w#

where q0 is the starting state of T.

Mapping ULP to MPCP

- Strings from the transition function δ:

List A	List B	
qX	Yp	from δ(q,X)=(p,Y,R)
ZqX	pZY	from δ(q,X)=(p,Y,L)
q#	Yp#	from δ(q,#)=(p,Y,R)
Zq#	pZY#	from δ(q,#)=(p,Y,L)

where Z is any tape symbol except the blank.

Mapping ULP to MPCP

- Strings for copying:

List A	List B
X	X

where X is any tape symbol (including the blank).

Mapping ULP to MPCP

- Strings for consuming the tape symbols at the end:

List A	List B
Xq	q
qY	q
XqY	q

where q is an accepting state, and each X and Y is any tape symbol except the blank.

Mapping ULP to MPCP

- Ending string:

List A	List B
q##	#

where q is an accepting state.

- Using this mapping, we can prove that the original ULP instance has a solution if and only if the mapped MPCP instance has a solution. (Textbook, p.402, Theorem 9.19)

8.3 Post's Correspondence Problem
(PCP)

Input: Two sequences, A = w1; : : : ;wk and
B = x1; : : : ; xk, where each wi and xi is a string
over some alphabet §.
Question: Is there a sequence i1; : : : ; im such
that 1 · ij · k for 1 · j · m and
wi1 ¢ ¢ ¢wim = xi1 ¢ ¢ ¢ xim?

Example:
A = 1; 10111; 10
B = 111; 10; 0

Solution: $2, 1, 1, 3$:

$$\overbrace{1\ 0\ 1\ 1\ 1}^{w_2}\ \overbrace{1}^{w_1}\ \overbrace{1}^{w_1}\ \overbrace{1\ 0}^{w_3}$$

$$\underbrace{1\ 0}_{x_2}\ \underbrace{1\ 1\ 1}_{x_1}\ \underbrace{1\ 1\ 1}_{x_1}\ \underbrace{0}_{x_3}$$

Given $v \in \{0,1\}^*$, consider the following instance of MPCP:

- Let $w_1 = \$$ and $x_1 = \$q_0v\$$.
- For each $X \in \Gamma \cup \{\$\}$, include the pair $\langle X, X \rangle$.
- For all $q \in Q - F$, $p \in Q$, X, Y, $Z \in \Gamma$, include

– $\langle qX, Yp \rangle$ if $\delta(q, X) = (p, Y, R)$;

– $\langle qX, Yp \rangle$ if $\delta(q, X) = (p, Y, R)$;

– $\langle ZqX, pZY \rangle$ and $\langle \$qX, \$qBY \rangle$ if $\delta(q, X) = (p, Y, L)$;

– $\langle q\$, Yp\$ \rangle$ if $\delta(q, B) = (p, Y, R)$;

- $\langle Zq\$, pZY\$\rangle$ and $\langle \$q\$, \$pBY\$\rangle$ if $\delta(q, B) = (p, Y, L)$.

- For each $q \in F$, $X \in \Gamma$, include $\langle Xq, q\rangle$, $\langle qX, q\rangle$, and $\langle q\$\$, \$\rangle$.

It can be shown that this instance has a solution iff $v \in L_U$.

Note that this instance has alphabet $Q \cup \Gamma \cup \{\$\}$, which is independent of v.

MPCP $\leq$ PCP:

Let (A, B) be an instance of MPCP over Σ, and let $*$ and $\$$ be distinct symbols not in Σ.

From $A = w_1, \ldots, w_k$, we construct $A' = w'_1, \ldots, w'_{k+1}$ as follows:

- Insert $*$ after each symbol in $w_1, \ldots, w_k$.
- Also, insert $*$ before the first symbol in w_1.
- Let $w'_{k+1} = \$$.

From $B = x_1, \ldots, x_k$, we construct $B' = x'_1, \ldots, x'_{k+1}$ as follows:

- Insert $*$ before each symbol in $x_1, \ldots, x_k$.
- Let $x'_{k+1} = *\$$.

It is easily seen that (A, B) has a solution for MPCP iff (A', B') has a solution for PCP.

The construction is clearly computable.

5.7: other undecidable problem

A problem P is *decidable* if it can be solved by a Turing machine T that always halt. (We say that P has an effective algorithm.)

Note that the corresponding language of a decidable problem is *recursive*.

Undecidable

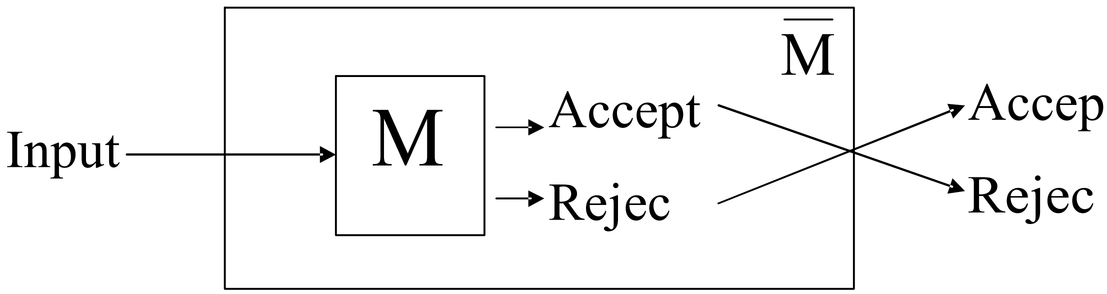
A problem is *undecidable* if it cannot be solved by any Turing machine that halts on all inputs.

Note that the corresponding language of an undecidable problem is *non-recursive*.

Complements of Recursive Languages

Theorem: If L is a recursive language, L is also recursive.

Proof: Let M be a TM for L that always halt. We can construct another TM M from M for L that always halts as follows:



Complements of RE Languages

Theorem: If both a language L and its complement L are RE, L is recursive.

Proof: Let M1 and M2 be TM for L and L respectively. We can construct a TM M from M1 and M2 for L that always halt as follows: