MODULE-I

What is TOC?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

- **states** (represented in the figure by circles),
- and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

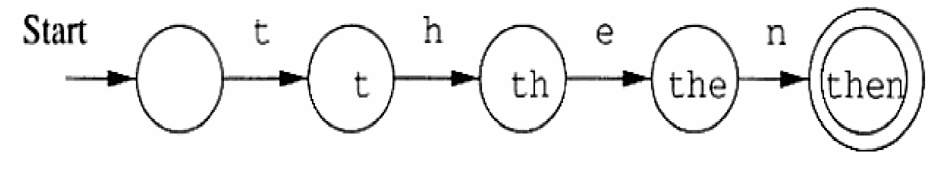


Figure 1.2: A finite automaton modeling recognition of then

Introduction to formal proof:

Basic Symbols used :

U – Union

∩- Conjunction

□ - Empty String

Φ – NULL set

7- negation

‘ – compliment

= > implies

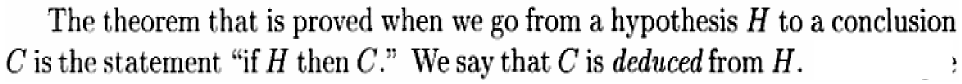
Additive inverse: a+(-a)=0
Multiplicative inverse: a*1/a=1
Universal set U={1,2,3,4,5}
Subset A={1,3}
A’ ={2,4,5}
Absorption law: AU(A ∩B) = A, A∩(AUB) = A

De Morgan’s Law:
(AUB)’ =A’ ∩ B’
(A∩B)’ = A’ U B’
Double compliment
(A’)’ =A
A ∩ A’ = Φ

Logic relations:
a → b = > **7**a U b
7(a∩b)=**7**a U **7**b

Relations:
Let a and b be two sets a relation R contains aXb.
Relations used in TOC:
Reflexive: a = a
Symmetric: aRb = > bRa
Transition: aRb, bRc = > aRc
If a given relation is reflexive, symmentric and transitive then the relation is called equivalence relation.

Deductive proof: Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.

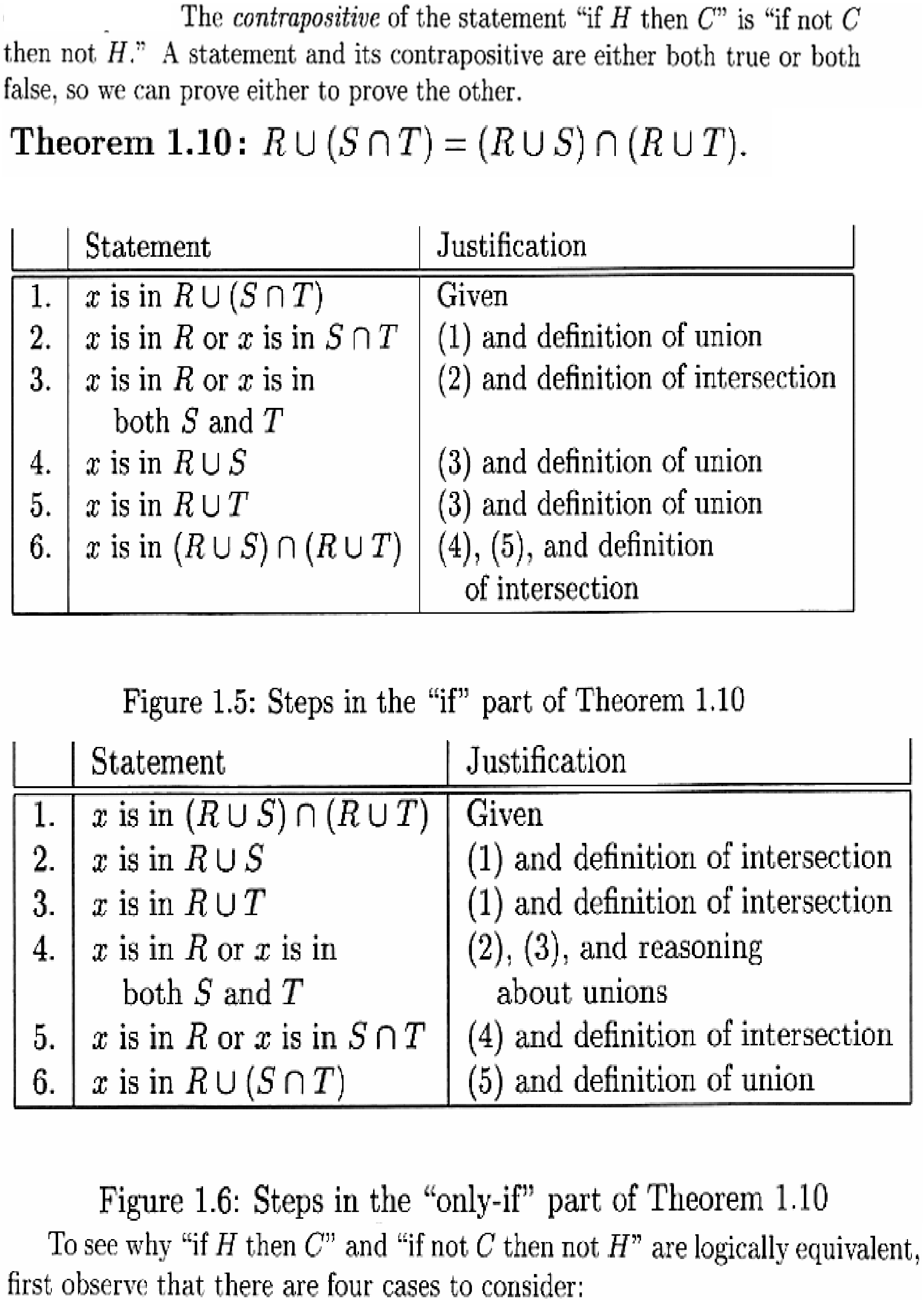


The theorem that is proved when we go from a hypothesis H to a conclusion C is the statement "if H then C." We say that C is *deduced* from H.

Additional forms of proof:
Proof of sets
Proof by contradiction
Proof by counter example

Direct proof (AKA) Constructive proof:
If *p* is true then *q* is true
Eg: if a and b are odd numbers then product is also an odd number.
Odd number can be represented as 2n+1
a=2x+1, b=2y+1
product of a X b = (2x+1) X (2y+1)
= 2(2xy+x+y)+1 = 2z+1 (odd number)

Proof by contrapositive:



The *contrapositive* of the statement "if H then C" is "if not C then not H." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

Theorem 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

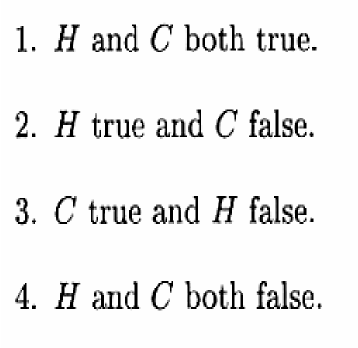
	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5), and definition of intersection

Figure 1.5: Steps in the "if" part of Theorem 1.10

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $R \cup S$	(1) and definition of intersection
3.	x is in $R \cup T$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3), and reasoning about unions
5.	x is in R or x is in $S \cap T$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

To see why "if H then C" and "if not C then not H" are logically equivalent, first observe that there are four cases to consider:



1. H and C both true.

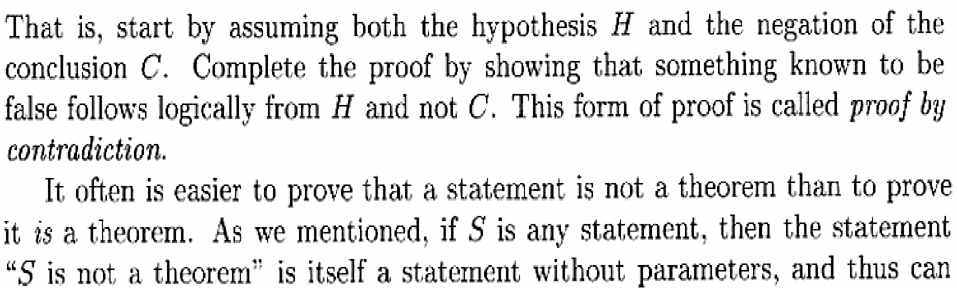
2. H true and C false.

3. C true and H false.

4. H and C both false.

Proof by Contradiction:

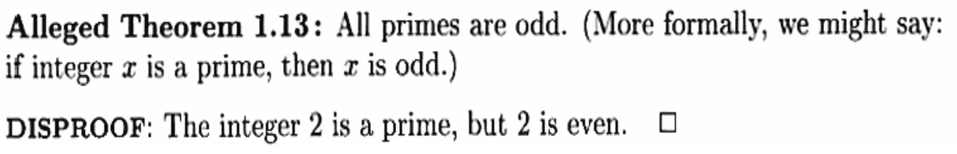
H and not C implies falsehood.



That is, start by assuming both the hypothesis H and the negation of the conclusion C. Complete the proof by showing that something known to be false follows logically from H and not C. This form of proof is called *proof by contradiction.*

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if S is any statement, then the statement "S is not a theorem" is itself a statement without parameters, and thus can

Be regarded as an observation than a theorem.



Alleged Theorem 1.13: All primes are odd. (More formally, we might say: if integer x is a prime, then x is odd.)

DISPROOF: The integer 2 is a prime, but 2 is even. □

For any sets a,b,c if $a \cap b = \Phi$ and c is a subset of b the prove that $a \cap c = \Phi$

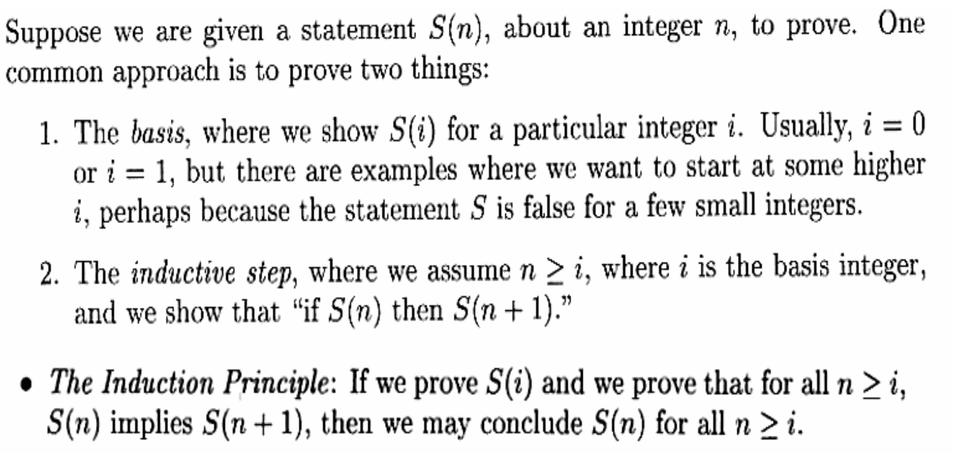
Given : $a \cap b = \Phi$ and c subset b

Assume: $a \cap c \neq \Phi$

Then $\forall x,\ x \in a$ *and* $x \in c \Rightarrow x \in b$

$\Rightarrow a \cap b \neq \Phi \Rightarrow a \cap c = \Phi$(i.e., the assumption is wrong)

Proof by mathematical Induction:



Suppose we are given a statement $S(n)$, about an integer n, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer i. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher i, perhaps because the statement S is false for a few small integers.

2. The *inductive step*, where we assume $n \geq i$, where i is the basis integer, and we show that "if $S(n)$ then $S(n+1)$."

- *The Induction Principle*: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.

Languages :

The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

Symbols :

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♠, *a*, 0, 1, #, begin, or do.

Alphabets :

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by Σ. When more than one alphabets are considered for discussion, then subscripts may be used (e.g. Σ_1, Σ_2 etc) or sometimes other symbol like G may also be introduced.

$\Sigma = \{0, 1\}$

$\Sigma = \{a, b, c\}$

$\Sigma = \{a, b, c, \&, z\}$

Example : $\Sigma = \{\#, \nabla, ♠, \beta\}$

Strings or Words over Alphabet :

A string or word over an alphabet Σ is a finite sequence of concatenated symbols of Σ.

Example : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

Length of a string :
The number of symbols in a string w is called its length, denoted by |w|.

Example : | 011 | = 4, |11| = 2, | b | = 1

Convention : We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to denote strings over an alphabet. That is, $a, b, c \in \Sigma$ (symbols) and u, v, w, x, y, z are strings.

Some String Operations :
Let $x = a_1a_2a_3 \in a_n$ and $y = b_1b_2b_3 \in b_m$ be two strings. The concatenation of x and y denoted by xy, is the string $a_1a_2a_3 \cdots a_nb_1b_2b_3 \cdots b_m$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

Example : Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: ε, 0, 01, 011.
Suffixes: ε, 1, 11, 011.
Substrings: ε, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and ε is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and x $\neq$ y.

In the above example, all prefixes except 011 are proper prefixes.

Powers of Strings : For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive definition of x^n as follows:
x^n = e, if n = 0 ; otherwise $x^n = xx^{n-1}$

Example : If $x = 011$, then $x^3 = 011011011$, $x^1 = 011$ and $x^0 = e$

Powers of Alphabets :

We write Σ^k (for some integer k) to denote the set of strings of length k with symbols from Σ. In other words,

$\Sigma^k = \{ w \mid w$ is a string over Σ and $|w| = k\}$. Hence, for any alphabet, Σ^0 denotes the set of all strings of length zero. That is, $\Sigma^0 = \{ e \}$. For the binary alphabet { 0, 1 } we have the following.

$\Sigma^0 = \{e\}$.

$\Sigma^1 = \{0, 1\}$.

$\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

The set Σ^* contains all the strings that can be generated by iteratively concatenating symbols from Σ any number of times.

Example : If $\Sigma = \{ a, b \}$, then $\Sigma^* = \{ \varepsilon$, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}.

Please note that if $\Sigma = F$, then Σ^* that is $\phi^* = \{e\}$. It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention

The set of all nonempty strings over an alphabet Σ is denoted by Σ^* That is,

$$\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

Note that Σ^* is infinite. It contains no infinite strings but strings of arbitrary lengths.

Reversal :

For any string $w = a_1 a_2 a_3 \cdots a_n$ the reversal of the string is $w^R = a_n a_{n-1} \cdots a_3 a_2 a_1$.

An inductive definition of reversal can be given as follows:

Languages :

A language over an alphabet is a set of strings over that alphabet. Therefore, a language L is any subset of Σ^*. That is, any $L \subseteq \Sigma^*$ is a language.

Example :

1. F is the empty language.
2. Σ^* is a language for any Σ.
3. {e} is a language for any Σ. Note that, $\phi \neq \{e\}$. Because the language F does not contain any string but {e} contains one string of length zero.
4. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.
5. The set of all strings over {a, b, c} that starts with a.

Convention : Capital letters A, B, C, L, etc. with or without subscripts are normally used to denote languages.

Set operations on languages : Since languages are set of strings we can apply set operations to languages. Here are some simple examples (though there is nothing new in it).

Union : A string $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$

Example : { 0, 11, 01, 011 } $\cup$ { 1, 01, 110 } = { 0, 11, 01, 011, 111 }

Intersection : A string, x□ L$_1$ $\cap$ L$_2$ iff x □ L$_1$ and x □ L$_2$.

Example : { 0, 11, 01, 011 } $\cap$ { 1, 01, 110 } = { 01 }

Complement : Usually, Σ^* is the universe that a complement is taken with respect to. Thus for a language L, the complement is L(bar) = { $x \in \Sigma^*$ | $x \notin L$ }.

Example : Let L = { x | |x| is even }. Then its complement is the language { $x \in \Sigma^*$ | $|x|$ is odd }.

Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.

Reversal of a language :

The reversal of a language *L*, denoted as L^R, is defined as: $L^R = \{w^R \mid w \in L\}$.

Example :

1. Let L = { 0, 11, 01, 011 }. Then L^R = { 0, 11, 10, 110 }.

2. Let L = { 1^n0^n | n is an integer }. Then L^R = { 1^n0^n | n is an integer }.

Language concatenation : The concatenation of languages L_1 and L_2 is defined as $L_1 L_2$ = { xy | $x \in L_1$ and $y \in L_2$ }.

Example : { *a*, *ab* }{ *b*, *ba* } = { *ab*, *aba*, *abb*, *abba* }.

Note that ,

1. $L_1 L_2 \neq L_2 L_1$ in general.
2. $L\Phi = \Phi$
3. $L\{\varepsilon\} = L = \{\varepsilon\}$

Iterated concatenation of languages : Since we can concatenate two languages, we also repeat this to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation L^n denotes the concatenation of L with itself n times. This is defined formally as follows:

$L_0 = \{e\}$
$L^n = LL^{n-1}$

Example : Let L = { a, ab }. Then according to the definition, we have

$L_0 = \{e\}$
$L_1 = L\{e\} = L = \{a, ab\}$
$L_2 = L\,L_1 = \{a, ab\}\{a, ab\} = \{aa, aab, aba, abab\}$
$L_3 = L\,L_2 = \{a, ab\}\{aa, aab, aba, abab\}$
$= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$

and so on.

Kleene's Star operation : The Kleene star operation on a language L, denoted as L^* is defined as follows :

L^* = (Union n in N) L^n

= $L^0 \cup L^1 \cup L^2 \cup \cdots$

= { x | x is the concatenation of zero or more strings from L }

Thus L^* is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define

$L^+ = LL^*$, i.e., all strings derivable by one or more concatenations of strings in L. That is

L^+ = (Union n in N and n >0) L^n

$= L^1 \cup L^2 \cup L^3 \cup \cdots$

Example : Let L = { a, ab }. Then we have,

$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$

$= \{e\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

$L^+ = L^1 \cup L^2 \cup L^3 \cup \cdots$

$= \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

Note : ε is in L^*, for every language L, including .

The previously introduced definition of Σ^* is an instance of Kleene star.

$$\begin{tikzcd}
\text{Grammar} \arrow[r, "\text{(Generates)}"] & \text{Language} & \text{Automata} \arrow[l, "\text{(Recognizes)}"']
\end{tikzcd}$$

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

- Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.

- The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton with output).
- The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet (whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
- The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.

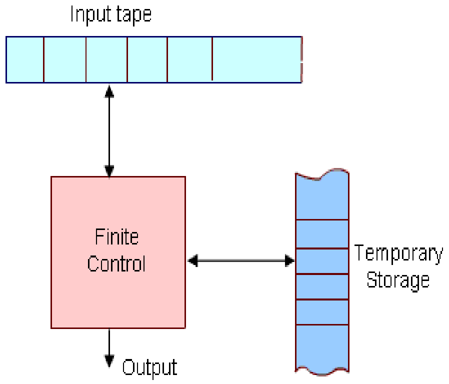


Figure 1: The figure above shows a diagrammatic representation of a generic automation.

Operation of the automation is defined as follows.
At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next (as defined by the transition function) is called a *move.* Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

Finite Automata

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

States, Transitions and Finite-State Transition System :

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system

containing only a finite number of states and transitions among them is called a *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

Deterministic Finite (-state) Automata

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from Σ.
2. A *tape head* for reading symbols from the tape
3. A *control* , which itself consists of 3 things:
   - finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
   - a current state, initially set to a start state,

- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition:

Deterministic Finite State Automaton : A Deterministic Finite State Automaton (DFA) is a 5-tuple : $M = (Q, \Sigma, \delta, q_0, F)$

- Q is a finite set of states.
- $\Sigma$is a finite set of input symbols or alphabet
- $\delta : Q \times \Sigma \to Q$ is the "next state" transition function (which is total). Intuitively, is δ a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q,a)$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$is the set of accept or final states.

Acceptance of Strings :

A DFA accepts a string $w = a_1 a_2 \cdots a_n$ if there is a sequence of states $q_0, q_1, \cdots, q_n$ in Q such that

1. $q_0$is the start state.
2. $\delta(q_i, q_{i+1}) = a_{i+1}$ for all $0 < i < n$.
3. $q_n \in F$

Language Accepted or Recognized by a DFA :

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by $L(M)$ i.e. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. The notion of acceptance can also be made more precise by extending the transition function δ.

Extended transition function :

Extend $\delta : Q \times \Sigma \rightarrow Q$ (which is function on symbols) to a function on strings, i.e. . $\delta^{\wedge} : Q \times \Sigma^{*} \rightarrow Q$

That is, $\delta^{\wedge}(q, w)$ is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:
The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$$L(M) = \{\, w \in \Sigma^{*} \mid M \text{ accepts } w \,\}$$

$$= \{\, w \in \Sigma^{*} \mid \delta^{\wedge}(q_0, w) \in F \,\}$$

Example 1 :

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

$Q = \{q_0, q_1\}$

q_0 is the start state

$F = \{q_1\}$

$\delta(q_0, 0) = q_0$ $\quad$ $\delta(q_1, 0) = q_1$

$\delta(q_0, 1) = q_1$ $\quad$ $\delta(q_1, 1) = q_1$

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1

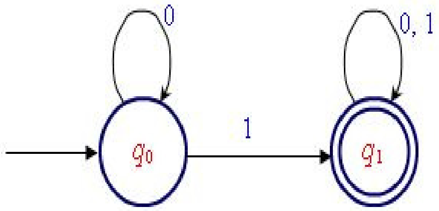
We can describe the same DFA by transition table or state transition diagram as following:

Transition Table :

	0	1
$\rightarrow q_0$	q_0	q_1

$*q_1$	q_1	q_1

It is easy to comprehend the transition diagram.



Explanation : We cannot reach find state q_1 w/0 or in the i/p string. There can be any no. of 0's at the beginning. (The self-loop at q_0 on label 0 indicates it). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

Transition table :

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

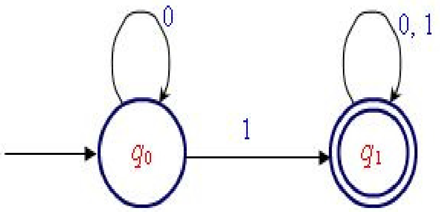
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_1

(State) Transition diagram :

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeled a iff $\delta(q, a) = p$. (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.

5. 

6. Here is an informal description how a DFA operates. An input to a DFA can be any string $w \in \Sigma^*$ Put a pointer to the start state q. Read the input string w from left to right, one symbol at a time, moving the pointer according to the transition function, δ. If the next symbol of w is a and the pointer is on state p, move the pointer to $\delta(p, a)$. When the end of the input string w is encountered, the pointer is on some state, r. The string is said to be accepted by the DFA if $r \in F$ and rejected if $r \notin F$. Note that there is no formal mechanism for moving the pointer.
7. A language $L \in \Sigma^*$ is said to be regular if L = L(M) for some DFA M.

Regular Expressions: Formal Definition

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

Definition : Let S be an alphabet. The regular expressions are defined recursively as follows.

Basis :

i) ϕ is a RE

ii) $\in$ is a RE

iii) $\forall a \in S$, a is RE.

These are called primitive regular expression i.e. Primitive Constituents

Recursive Step :

If r_1 and r_2 are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

iii) r_1^*

iv) (r_1)

Closure : r is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no of applications of the recursive step (given in 2).

Example : Let Σ = { 0,1,2 }. Then (0+21)*(1+ F) is a RE, because we can construct this expression by applying the above rules as given in the following step.

Steps	RE Constructed	Rule Used
1	1	Rule 1(iii)
2	ϕ	Rule 1(i)
3	1+ϕ	Rule 2(i) & Results of Step 1, 2
4	(1+ϕ)	Rule 2(iv) & Step 3
5	2	1(iii)
6	1	1(iii)
7	21	2(ii), 5, 6
8	0	1(iii)
9	0+21	2(i), 7, 8
10	(0+21)	2(iv), 9
11	(0+21)*	2(iii), 10
12	(0+21)*	2(ii), 4, 11

Language described by REs : Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.

Notation : If r is a RE over some alphabet then L(r) is the language associate with r . We can define the language L(r) associated with (or described by) a REs as follows.

1. ϕ is the RE describing the empty language i.e. $L(\phi) = \phi$.

2. $\in$ is a RE describing the language {$\in$} i.e. $L(\in) = \{\in\}$.

3. $\forall a \in S$, a is a RE denoting the language {a} i.e . L(a) = {a} .

4. If r_1 and r_2 are REs denoting language $L(r_1)$ and $L(r_2)$ respectively, then

i) $r_1 + r_2$ is a regular expression denoting the language $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

ii) $r_1 r_2$ is a regular expression denoting the language $L(r_1 r_2)=L(r_1)\ L(r_2)$

iii) r_1^* is a regular expression denoting the language $L(r_1^*) = (L(r1))^*$

iv) (r_1) is a regular expression denoting the language $L((r_1)) = L(r_1)$

Example : Consider the RE (0*(0+1)). Thus the language denoted by the RE is

L(0*(0+1)) = L(0*) L(0+1) ....by 4(ii)

= L(0)*L(0) ∪ L(1)

= {∈ , 0,00,000,. } {0}   {1}∪

= {∈ , 0,00,000,........} {0,1}

= {0, 00, 000, 0000,..........,1, 01, 001, 0001,...............}

Precedence Rule

Consider the RE $ab + c$. The language described by the RE can be thought of either $L(a)L(b+c)$ or $L(ab)$ $L(c)$ as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression- (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i) The star operator precedes concatenation and concatenation precedes union (+) operator.

ii) It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language $L(ab)$ $L(c)$ i.e. it should be grouped as $((ab)+c)$.

∪

We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE $a(b+c)$ is $L(a)L(b+c)$.

Example : The RE ab^*+b is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* \cup L(b)$

Example : The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* \cup L(b)$.

Example : It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Example : The regular expression r =(00)*(11)*1 denotes the set of all strings with an even number of 0's followed by an odd number of 1's i.e. $L(r) = \{0^{2n}1^{2m+1} \mid n \geq 0,\ m \geq 0\}$

Note : The notation r^+ is used to represent the RE rr^*. Similarly, r^2 represents the RE rr, r^3 denotes r^2r, and so on.

An arbitrary string over Σ = {0,1} is denoted as (0+1)*.

Exercise : Give a RE r over {0,1} s.t. $L(r)$={$\omega \in \Sigma^* \mid \omega$ has at least one pair of consecutive 1's}

Solution : Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes before is completely arbitrary. Considering these observations we can write the REs as (0+1)*11(0+1)*.

Example : Considering the above example it becomes clean that the RE (0+1)*11(0+1)*+(0+1)*00(0+1)* represents the set of string over {0,1} that contains the substring 11 or 00.

Example : Consider the RE 0*10*10*. It is not difficult to see that this RE describes the set of strings over {0,1} that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

Example : Consider the language of strings over {0,1} containing two or more 1's.

Solution : There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

i) 0*10*1(0+1)*

ii) (0+1)*10*10*

Example : Consider a RE r over {0,1} such that

$L(r)$ = { $\omega \in \{0,1\}^*$ | ω has no pair of consecutive 1's}

Solution : Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0*100*. So it looks like the RE is (0*100*)*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is r = (0*100*)(1+ $\in$)+0*(1+$\in$).

Alternative Solution :

The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as r = (0+10)*(1+$\in$).This is a shorter expression but represents the same language.

Regular Expression and Regular Language :

Equivalence(of REs) with FA :

Recall that, language that is accepted by some FAs are known as Regular language. The two concepts : REs and Regular language are essentially same i.e. (for) every regular language can be developed by (there is) a RE, and for every RE there is a Regular Langauge. This fact is rather suprising, because RE approach to describing language is fundamentally differnet from the FA approach. But REs and FA are equivalent in their descriptive power. We can put this fact in the focus of the following Theorem.

Theorem : A language is regular iff some RE describes it.

This Theorem has two directions, and are stated & proved below as a separate lemma

RE to FA :

REs denote regular languages :

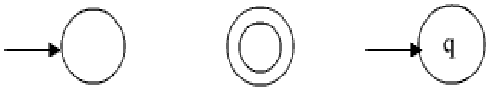
Lemma : If $L(r)$ is a language described by the RE r, then it is regular i.e. there is a FA such that $L(M) \cong L(r)$.

Proof : To prove the lemma, we apply structured index on the expression r. First, we show how to construct FA for the basis elements: ϕ, $\in$and for any $a \in \Sigma$. Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the languages accepted by the original smaller automata.

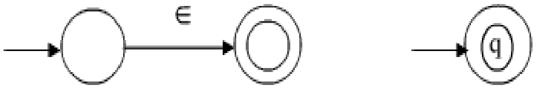
Use of NFAs is helpful in the case i.e. we construct NFAs for every REs which are represented by transition diagram only.

Basis :

- Case (i) **:** $r = \phi$. Then $L(r) = \phi$. Then $L(r) = \phi$ and the following NFA N recognizes $L(r)$. Formally $N = (Q, \{q\}, \Sigma, \delta, q, F, \phi)$ where $Q = \{q\}$ and $\delta(q,a) = \phi \; \forall \; a \in S, \; F = \phi$.

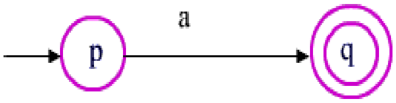


- Case (ii) **:** $r = \in$. $L(r) = \{\in\}$, and the following NFA N accepts L(r). Formally $N = (\{q\}, \Sigma, \delta, q, \{q\})$ where $\delta(q,a) = \phi \quad \forall a \in \Sigma$.



Since the start state is also the accept step, and there is no any transition defined, it will accept the only string $\in$ and nothing else.

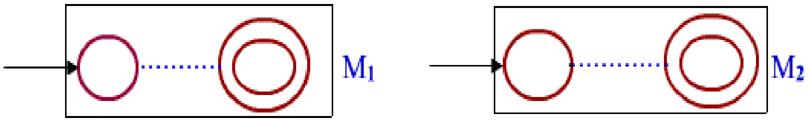
- Case (iii) : $r = a$ for some $a \in \Sigma$. Then $L(r) = \{a\}$, and the following NFA N accepts $L(r)$.



Formally, $N = (\{p,q\}, \Sigma, \delta, p, \{q\})$ where $\delta(p,q) = \{q\}$, $\delta(s, b) = (\phi)$ for $s \neq p$ or $b \neq a$

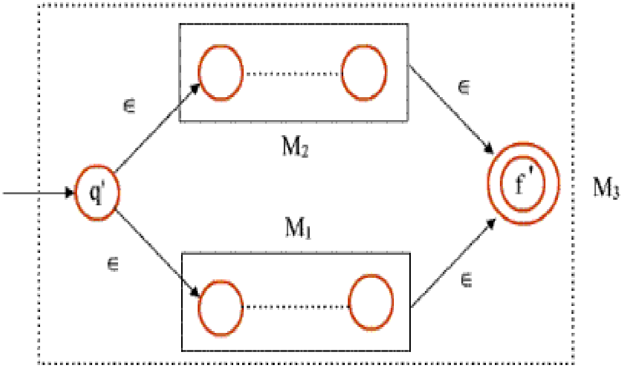
Induction :

Assume that the start of the theorem is true for REs r_1 and r_2. Hence we can assume that we have automata M_1 and M_2 that accepts languages denoted by REs r_1 and r_2, respectively i.e. $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The FAs are represented schematically as shown below.



Each has an initial state and a final state. There are four cases to consider.

- Case (i) **:** Consider the RE $r_3 = r_1 + r_2$ denoting the language $L(r_1) \cup L(r_2)$. We construct FA M_3, from M_1 and M_2 to accept the language denoted by RE r_3 as follows **:**



Create a new (initial) start state q' and give $\in$- transition to the initial state of M_1 and M_2.This is the initial state of M_3.

- Create a final state f' and give $\in$-transition from the two final state of M_1 and M_2. f' is the only final state of M_3 and final state of M_1 and M_2 will be ordinary states in M_3.
- All the state of M_1 and M_2 are also state of M_3.

- All the moves of M_1 and M_2 are also moves of M_3. [Formal Construction]

It is easy to prove that $L(M_3) = L(r_3)$

Proof: To show that $L(M_3) = L(r_3)$ we must show that

= $L(r_1) \cup L(r_2)$

= $L(M_1) = L(M_2)$ by following transition of M_3

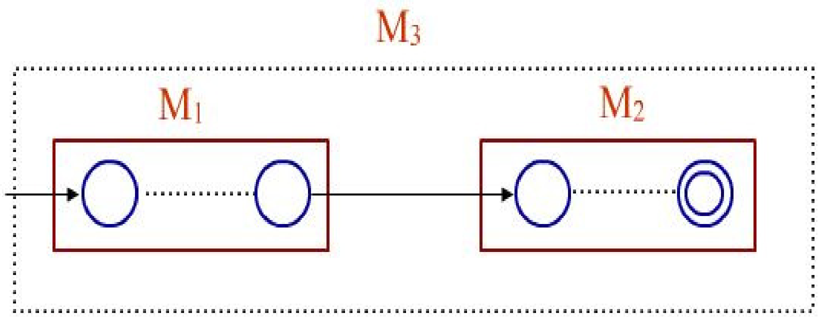
Starts at initial state q' and enters the start state of either M_1 or M_2 follwoing the transition i.e. without consuming any input. WLOG, assume that, it enters the start state of M_1. From this point onward it has to follow only the transition of M_1 to enter the final state of M_1, because this is the only way to enter the final state of M by following the e-transition.(Which is the last transition & no input is taken at hte transition). Hence the whole input w is considered while traversing from the start state of M_1 to the final state of M_1. Therefore M_1 must accept w_3.

Say, $w \in L(M_1)$ or $w \in L(M_2)$.

WLOG, say $w \in L(M_1)$

Therefore when M_1 process the string w , it starts at the initial state and enters the final state when w consumed totally, by following its transition. Then M_3 also accepts w, by starting at state q' and taking $\in$-transition enters the start state of M_1-follows the moves of M_1 to enter the final state of M_1 consuming input w thus takes $\in$-transition to f'. Hence proved

- Case(ii) **:** Consider the RE $r_3 = r_1r_2$ denoting the language $L(r_1)L(r_2)$. We construct FA M_3 from M_1 & M_2 to accept $L(r_3)$ as follows **:**



Create a new start state q' and a new final state

1. Add $\in$- transition from
    - ○ q' to the start state of M_1
    - ○ q' to f'
    - ○ final state of M_1 to the start state of M_1
2. All the states of M_1 are also the states of M_3. M_3 has 2 more states than that of M_1 namely q' and f'.
3. All the moves of M_1 are also included in M_3.

By the transition of type (b), M_3 can accept $\in$.
By the transition of type (a), M_3 can enters the initial state of M_1 w/o any input and then follow all kinds moves of M_1 to enter the final state of M_1 and then following $\in$-transition can enter f'. Hence if any $w \in \Sigma^*$ is accepted by M_1 then w is also accepted by M_3. By the transition of type (b), strings accepted by M_1 can be repeated by any no of times & thus accepted by M_3. Hence M_3 accepts $\in$ and any string accepted by M_1 repeated (i.e. concatenated) any no of times. Hence $L(M_3) = (L(M_1))^* = (L(r)_1)^* = r_1^*$

Case(iv) **:** Let r_3=(r_1). Then the FA M_1 is also the FA for (r_1), since the use of parentheses does not change the language denoted by the expression

Non-Deterministic Finite Automata

Nondeterminism is an important abstraction in computer science. Importance of nondeterminism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonean cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.

In the context of FA nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

- multiple next state.
- $\in$- transitions.

Multiple Next State :

- In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in Σ).
- This means that - in a state q and with input symbol a - there could be one, more than one or zero next state to go, i.e. the value of $\delta(q,a)$ is a subset of Q. Thus $\delta(q,a) = \{q_1, q_2, \cdots, q_k\}$ which means that any one of $q_1, q_2, \cdots, q_k$ could be the next state.
- The zero next state case is a special one giving $\delta(q,a) = \phi$, which means that there is no next state on input symbol when the automata is in state q. In such a case, we may think that the automata "hangs" and the input will be rejected.

$\in$- transitions :

In an -transition, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q,\in) = \{q_1, q_2, \cdots, q_k\}$ implying that the next state could by any one of $q_1, q_2, \cdots, q_k$ w/o consuming the next input symbol.

Acceptance :

Informally, an NFA is said to accept its input ω if it is possible to start in some start state and process ω, moving according to the transition rules and making choices along the way whenever the next state is not uniquely defined, such that when ω is completely processed (i.e. end of ω is reached), the automata is in an accept state. There may be several possible paths through the automation in response to an input ω since the start state is not determined and there are choices along the way because of multiple next states. Some of these paths may lead to accpet states while others may not. The automation is said to accept ω if at least one computation path on input ω starting from at least one start state leads to an accept state- otherwise, the automation rejects input ω. Alternatively, we can say that, ω is accepted iff there exists a path with label ω from some start state to some accept state. Since there is no mechanism for determining which state to start in or which of the possible next moves to take (including the ω - transitions) in response to an input symbol we can think that the automation is having some "guessing" power to chose the correct one in case the input is accepted

Example 1 : Consider the language $L = \{\omega \in \{0, 1\}^* \mid$ The 3rd symbol from the right is 1$\}$. The following four-state automation accepts L.

The m/c is not deterministic since there are two transitions from state q_1 on input 1 and no transition (zero transition) from q_4 on both 0 & 1.

For any string ω whose 3rd symbol from the right is a 1, there exists a sequence of legal transitions leading from the start state q, to the accept state q_4. But for any string ω where 3rd symbol from the right is 0, there is no possible sequence of legal tranisitons leading from q_1 and q_4. Hence m/c accepts L. How does it accept any string $\omega \in L$?

Formal definition of NFA :

Formally, an NFA is a quituple $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

$$\delta : Q \times (\Sigma \cup \{\in\}) \to P(Q)$$

where $P(Q)$ is the power set of Q i.e. 2^Q.

The Langauge of an NFA :

From the discussion of the acceptance by an NFA, we can give the formal definition of a language accepted by an NFA as follows :

If $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then the langauge accepted by N is writtten as $L(N)$ is given by $L(N) = \{\omega \mid \hat{\delta}(q_0, \omega) \cap F = \phi\}$.

That is, $L(N)$ is the set of all strings w in Σ^* such that $\hat{\delta}(q_0, \omega)$ contains at least one accepting state.

Removing ▯-transition:

$\in$- transitions do not increase the power of an *NFA* . That is, any $\in$- *NFA* (*NFA* with $\in$transition), we can always construct an equivalent *NFA* without $\in$-transitions. The equivalent *NFA* must keep track where the $\in$*NFA* goes at every step during computation. This can be done by adding extra transitions for removal of every $\in$- transitions from the $\in$- *NFA* as follows.

If we removed the $\in$- transition $\delta(p,\in) = q$ from the $\in$- *NFA* , then we need to moves from state *p* to all the state r on input symbol $q \in \Sigma$ which are reachable from state q (in the $\in$- *NFA*) on same input symbol *q*. This will allow the modified *NFA* to move from state *p* to all states on some input symbols which were possible in case of $\in$-*NFA* on the same input symbol. This process is stated formally in the following theories.

Theorem if *L* is accepted by an $\in$- *NFA N* , then there is some equivalent $NFA \;\; N'$ without $\in$transitions accepting the same language *L*

Proof:

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the given $\in - NFA$ with

We construct $N' = (Q, \Sigma, \delta', q_0, F')$

Where, $\delta'(q,a) = \left\{ p \,\middle|\, p \in \hat{\delta}(q,a) \right\}$ for all $q \in Q,$ and $a \in \Sigma,$ and

$$F' = \left\{ {}^{F}_{F} U(q_0) \text{ if } \hat{\delta}(q_0,\in) \bigcap F \neq \phi \text{ otherwise.} \right.$$

Other elements of *N'* and *N*

We can show that $L(N) = L(N')$ i.e. *N'* and *N* are equivalent.

We need to prove that $\forall w \in \Sigma^*$

$$w \in L(N) \;\text{ iff }\; w \in L(N')$$ i.e.

$$\forall w \in \Sigma^* \quad \hat{\delta}'(q_0, w) \in F' \;\text{ iff }\; \hat{\delta}(q_0, w) \in F$$

We will show something more, that is,

$$\forall w \in \Sigma^* \quad \hat{\delta}'(q_0, w) = \hat{\delta}(q_0, w)$$

We will show something more, that is, $|w|$

Basis : $|w| = 1$, then $x = a \in \Sigma$

But $\hat{\delta}'(q_0, a) = \hat{\delta}(q_0, a)$ by definition of δ'.

Induction hypothesis Let the statement hold for all $w \in \Sigma^*$ with $|w| \le n$.

$$\begin{aligned}
\hat{\delta}'(q_0, w) &= \hat{\delta}'(q_0, xa) && \text{By definition of extension of } \hat{\delta}' \\
&= \delta'\left(\hat{\delta}'(q_0, x), a\right) && \text{By inductions hypothesis.} \\
&= \delta'\left(\hat{\delta}(q_0, x), a\right) && \text{Assuming that} \\
&= \delta'(R, a) && \hat{\delta}(q_0, x) = R, \text{where } R \subseteq Q \\
&= \bigcup_{p \in R} \delta'(p, a) && \\
&= \bigcup_{p \in R} \hat{\delta}(p, a) && \text{By definition of } \delta' \\
&= \hat{\delta}(q_0, xa) && \text{Since } R = \hat{\delta}(q_0, x) \\
&= \hat{\delta}(q_0, w) &&
\end{aligned}$$

To complete the proof we consider the case

When $|w| = 0$ i.e. $w = \in$ then

$\delta'(q_0,\in)=\{q_0\}$ and by the construction of F', $q_0 \in F'$ wherever $\hat{\delta}(q_0,\in)$ constrains a state in *F*.

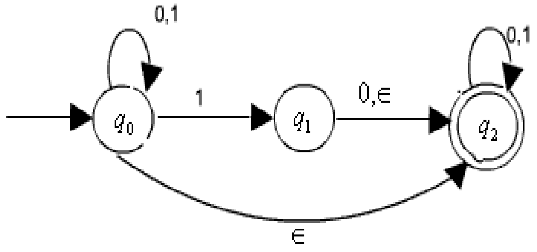
If $F' = F$ (and thus $\hat{\delta}(q_0,\in)$ is not in *F*), then $\forall w$ with $|w| = 1,\ w$ leads to an accepting state in *N'* iff it leac to an accepting state in *N* (by the construction of *N'* and *N*).

Also, if ($w = \in$, thus *w* is accepted by N' iff *w* is accepted by *N* (iff $q_0 \in F$)

If $F' = F \cup \{q_0\}$ (and, thus in *M* we load $\hat{\delta}(q_0,\in)$ in *F*), thus $\in$ is accepted by both *N'* and *N* .

Let $|w| \geq 1$. If *w* cannot lead to q_0 in *N* , then $w \in L(N)$. (Since can add $\in$ transitions to get an accept state). So there is no harm in making q_0 an accept state in *N'*.

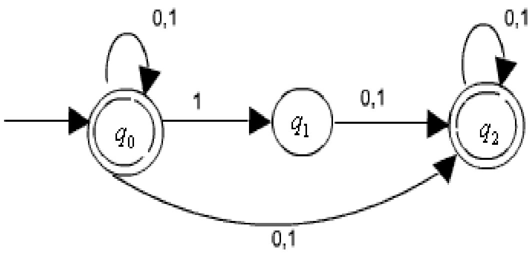
Ex: Consider the following *NFA* with $\in$- transition.



Transition Diagram δ

	0	1	$\in$
$\rightarrow q_0$	$\{q_0\}$	$\{q_0,q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	ϕ	$\{q_2\}$
$F\ \ q_2$	$\{q_2\}$	ϕ	$\{q_2\}$

Transition diagram for δ' for the equivalent *NFA* without $\in$- moves



	0	1
$\underset{F}{\rightarrow} q_0$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_2\}$	$\{q_2\}$
$F\ \ q_2$	$\{q_2\}$	$\{q_2\}$

Since $\delta(q_0, \in) = q_2 \in -NFA$ the start state q_0 must be final state in the equivalent *NFA* .

Since $\delta(q_0, \in) = q_2$ and $\delta(q_2, 0) = q_2$ and $\delta(q_2, 1) = q_2$ we add moves $\delta(q_0, 0) = q_2$ and $\delta(q_0, 1) = q_2$ in the equivalent *NFA* . Other moves are also constructed accordingly.

∈-closures:

The concept used in the above construction can be made more formal by defining the ∈-closure for a state (or a set of states). The idea of ∈-closure is that, when moving from a state *p* to a state *q* (or from a set of states S_i to a set of states S_j) an input $a \in \Sigma$, we need to take account of all ∈-moves that could be made after the transition. Formally, for a given state *q*,

∈-closures: $(q) = \{p \mid p \text{ can be reached from } q \text{ by zero or more } \in \text{-moves}\}$

Similarly, for a given set $R \subseteq Q$

∈-closures:
$(R) = \{p \in Q \mid p \text{ can be reached from any } q \in R \text{ by following zero or more } \in \text{-moves}\}$

So, in the construction of equivalent *NFA N'* without ∈-transition from any *NFA* with ∈moves. the first rule can now be written as $\delta'(q, a) = \in\text{-closure}(\delta(q, a))$

Equivalence of *NFA* and *DFA*

It is worth noting that a *DFA* is a special type of *NFA* and hence the class of languages accepted by *DFA* s is a subset of the class of languages accepted by *NFA* s. Surprisingly, these two classes are in fact equal. *NFA* s appeared to have more power than *DFA* s because of generality enjoyed in terms of $\in$-transition and multiple next states. But they are no more powerful than *DFA* s in terms of the languages they accept.

Converting *DFA* to *NFA*

Theorem: Every *DFA* has as equivalent *NFA*

Proof: A *DFA* is just a special type of an *NFA* . In a *DFA* , the transition functions is defined from $Q \times \Sigma \text{ to } Q$ whereas in case of an *NFA* it is defined from $Q \times \Sigma \text{ to } 2^Q$ and $D = (Q, \Sigma, \delta, q_0, F)$ be *a DFA* . We construct an equivalent *NFA* $N = (Q', \Sigma, \delta', q_0, F)$ as follows.

$\{q_i\} \in Q', \forall q_i \in Q$

$\delta'(\{p\}, a) = \{\delta(p, a)\}$, i. e

If $\delta(p, a) = q$, and $\delta'(\{p\}, a) = \{q\}$.

All other elements of *N* are as in *D*.

If $w = a_1 a_2 \cdots, a_n \in L(D)$ then there is a sequence of states $q_0, q_1, q_2 \cdots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i \text{ and } q_n \in F$

Then it is clear from the above construction of *N* that there is a sequence of states (in *N*) $\{q_0\}, \{q_1\}, \{q_2\}, \cdots, \{q_n\}$ such that $\delta'(\{q_{i-1}\}, a_i) = \{q_i\}$ and $\{q_n\} \in F$ and hence $w \in L(N)$.

Similarly we can show the converse.

Hence , $L(N) = L(D)$

Given any *NFA* we need to construct as equivalent *DFA* i.e. the *DFA* need to simulate the behaviour of the *NFA* . For this, the *DFA* have to keep track of all the states where the NFA could be in at every step during processing a given input string.

There are 2^n possible subsets of states for any *NFA* with *n* states. Every subset corresponds to one of the possibilities that the equivalent *DFA* must keep track of. Thus, the equivalent *DFA* will have 2^n states.

The formal constructions of an equivalent *DFA* for any *NFA* is given below. We first consider an *NFA* without $\in$transitions and then we incorporate the affects of $\in$transitions later.

Formal construction of an equivalent *DFA* for a given *NFA* without $\in$transitions.

Given an $N = (Q, \Sigma, \delta, q_0, F)$ without $\in$- moves, we construct an equivalent *DFA*

$D = \left(Q^D, \Sigma, \delta^D, q_0^D, F^D\right)$ as follows

$Q^D = P(Q)$ i.e. $Q^D = \{S \mid S \subseteq Q\}$,

$q_0^D = \{q_0\}$,

$F^D = \left\{q^D \in Q^D \middle| q^D \cap F \neq \phi\right\}$ (i.e. every subset of *Q* which as an element in *F* is considered as a final stat in *DFA D*)

$\delta^D(\{q_1, q_2, \cdots, q_k\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \cdots \cup \delta(q_k, a)$

for all $a \in \Sigma$ and $q^D = \{q_1, q_2, \cdots, q_k\}$

where $q_i \in Q,\ 1 \leq i \leq k.$

That is, $\delta^D\left(q^D, a\right) = \bigcup_{q_i \in q^D} \delta(q_i, a)$

To show that this construction works we need to show that *L(D)=L(N)* i.e.

$\forall w \in \Sigma^*\ \ \hat{\delta}^D\left(q_0^D, w\right) \in F^D \text{ iff } \hat{\delta}(q_0, w) \cap F \neq \phi$

Or, $\forall w \in \Sigma^*\ \ \hat{\delta}^D(\{q_0\}, w) \cap F \neq \text{ iff } \hat{\delta}(q_0, w) \cap F \neq \phi$

We will prove the following which is a stranger statement thus required.

$$\forall w \in \Sigma^*, \quad \hat{\delta}^D(\{q_0\}, w) = \hat{\delta}(q_0, w)$$

Proof : We will show by inductions on $|w|$

Basis If $|w|$=0, then *w* =$\in$

So, $\delta^D(\{q_0\},\in) = \{q_0\} = \delta(q_0,\in)$, by definition.

Inductions hypothesis : Assume inductively that the statement holds $\forall w \in \Sigma^*$ of length less than or equal to *n*.

Inductive step

Let $|w| = n+1$, then $w = xa$ with $|x| = n \text{ and } a \in \Sigma.$

Now,

$$\begin{aligned}
\hat{\delta}^D(\{q_0\}, w) &= \hat{\delta}^D(\{q_0\}, xa) \\
&= \delta^D\left(\hat{\delta}^D(\{q_0\}, x), a\right), \text{ by inductive extension of } \delta^D \\
&= \delta^D\left(\hat{\delta}(q_0, x), a\right), \text{by induction hypothesis} \\
&= \bigcup_{q_i \in \delta(q_0,x)} \delta(q_i, a), \text{ by definition of } \delta^D \\
&= \delta(q_0, xa) \quad \text{by definition of } \hat{\delta} \text{ (extension of } \delta) \\
&= \delta(q_0, w)
\end{aligned}$$

Now, given any *NFA* with $\in$-transition, we can first construct an equivalent *NFA* without $\in$-transition and then use the above construction process to construct an equivalent *DFA* , thus, proving the equivalence of *NFA* s and *DFA* s..

It is also possible to construct an equivalent *DFA* directly from any given *NFA* with $\in$-transition by integrating the concept of $\in$-closure in the above construction.

Recall that, for any $S \subseteq Q,$

$\in$- closure :
$(S) = \{q \in Q \mid q \text{ can be reached from any } p \in S \text{ by following zero or more } \in -\text{transitions}\}$

In the equivalent *DFA* , at every step, we need to modify the transition functions δ^D to keep track of all the states where the *NFA* can go on ∈-transitions. This is done by replacing $\delta(q,a)$ by ∈-closure $(\delta(q,a))$, i.e. we now compute $\delta^D(q^D,a)$ at every step as follows:

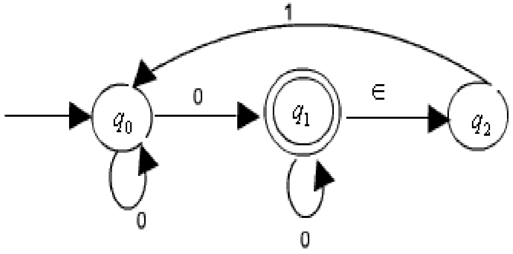
$$\delta^D(q^D,a) = \left\{q \in Q \middle| q \in \ \in\text{-closure}\left(\delta(q^D,a)\right)\right\}.$$

Besides this the initial state of the *DFA D* has to be modified to keep track of all the states that can be reached from the initial state of *NFA* on zero or more -transitions.

This can be done by changing the initial state q_0^D to ∈-closure (q_0^D) .
It is clear that, at every step in the processing of an input string by the *DFA D* , it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any ∈-*NFA*

If the number of states in the *NFA* is *n* , then there are 2^n states in the *DFA* . That is, each state in the *DFA* is a subset of state of the *NFA* .

But, it is important to note that most of these 2^n states are inaccessible from the start state and hence can be removed from the *DFA* without changing the accepted language. Thus, in fact, the number of states in the equivalent *DFA* would be much less than 2^n .
Example : Consider the NFA given below.



	0	1	∈
$\rightarrow q_0$	$\{q_0,q_1\}$	ϕ	ϕ
$F\ q_1$	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_0\}$

Since there are 3 states in the NFA

There will be $2^3 = 8$ states (representing all possible subset of states) in the equivalent *DFA* . The transition table of the *DFA* constructed by using the subset constructions process is produced here.

	0	1
ϕ	ϕ	ϕ
$\rightarrow q_0$	$\{q_0,q_1,q_2\}$	ϕ
$F\{q_1\}$	$\{q_1,q_2\}$	$\{q_0\}$
$\{q_2\}$	ϕ	$\{q_0\}$
$F\{q_0,q_1\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$
$\{q_0,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$
$F\{q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_0\}$
$F\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$

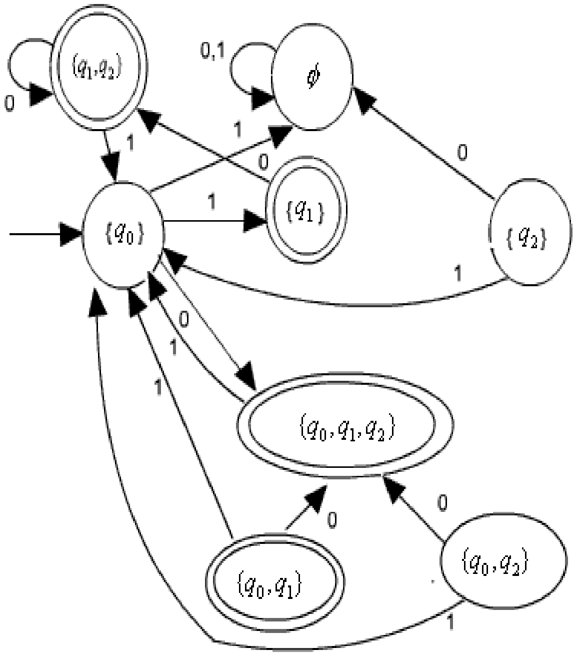
The start state of the *DFA* is $\in$- closures $(q_0) = \{q_0\}$

The final states are all those subsets that contains q_1 (since $q_1 \in F$ in the *NFA*).

Let us compute one entry,

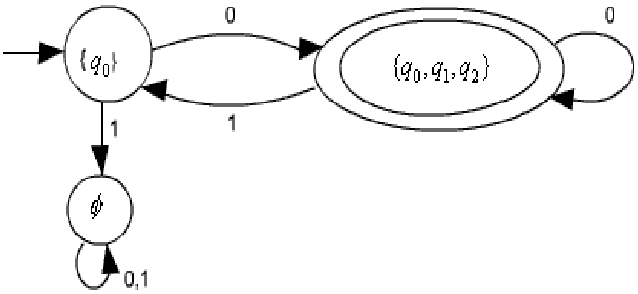
$$\begin{aligned}\delta^D(\{q_0,0\}) &= \in -\text{closure}(\delta(q_0,0)) \\ &= \in -\text{closure}(\{q_0,q_1\}) \\ &= \{q_0,q_1,q_2\}\end{aligned}$$

Similarly, all other transitions can be computed



	0	1
$\rightarrow\{q_0\}$	$\{q_0,q_1,q_2\}$	ϕ
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$

Corresponding Transition fig. for DFA. Note that states $\{q_1\},\{q_2\},\{q_1,q_2\},\{q_0,q_2\}$ and $\{q_0,q_1\}$ are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



It is interesting to note that we can avoid encountering all those inaccessible or unnecessary states in the equivalent *DFA* by performing the following two steps inductively.

1. If q_0 is the start state of the NFA, then make $\in$- closure (q_0) the start state of the equivalent *DFA* . This is definitely the only accessible state.
2. If we have already computed a set δ of states which are accessible. Then $\forall a \in \Sigma$. compute $\left(\delta^D(S,a)\right)$ because these set of states will also be accessible.

Following these steps in the above example, we get the transition table given below

MODULE-II

Regular Expressions: Formal Definition

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

Definition : Let S be an alphabet. The regular expressions are defined recursively as follows.

Basis :

i) ϕ is a RE

ii) $\in$ is a RE

iii) $\forall\ a \in S$, a is RE.

These are called primitive regular expression i.e. Primitive Constituents

Recursive Step :

If r_1 and r_2 are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

iii) r_1^*

iv) (r_1)

Closure : r is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no of applications of the recursive step (given in 2).

Example : Let Σ = { 0,1,2 }. Then (0+21)*(1+ F) is a RE, because we can construct this expression by applying the above rules as given in the following step.

Steps	RE Constructed	Rule Used
1	1	Rule 1(iii)
2	ϕ	Rule 1(i)
3	1+ ϕ	Rule 2(i) & Results of Step 1, 2

4	$(1+\phi)$	Rule 2(iv) & Step 3
5	2	1(iii)
6	1	1(iii)
7	21	2(ii), 5, 6
8	0	1(iii)
9	0+21	2(i), 7, 8
10	(0+21)	2(iv), 9
11	(0+21)*	2(iii), 10
12	(0+21)*	2(ii), 4, 11

Language described by REs : Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.

Notation : If r is a RE over some alphabet then $L(r)$ is the language associate with r . We can define the language $L(r)$ associated with (or described by) a REs as follows.

1. ϕ is the RE describing the empty language i.e. $L(\phi) = \phi$.

2. $\in$ is a RE describing the language $\{\in\}$ i.e. $L(\in) = \{\in\}$.

3. $\forall a \in S$, a is a RE denoting the language $\{a\}$ i.e . L(a) = {a} .

4. If r_1 and r_2 are REs denoting language $L(r_1)$ and $L(r_2)$ respectively, then

i) $r_1 + r_2$ is a regular expression denoting the language $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

ii) $r_1 r_2$ is a regular expression denoting the language $L(r_1 r_2)=L(r_1)\, L(r_2)$

iii) r_1^* is a regular expression denoting the language $L(r_1^*) = (L(r1))^*$

iv) (r_1) is a regular expression denoting the language $L((r_1)) = L(r_1)$

Example : Consider the RE (0*(0+1)). Thus the language denoted by the RE is

L(0*(0+1)) = L(0*) L(0+1) ....by 4(ii)

= L(0)*L(0) $\cup$ L(1)

= {$\in$, 0,00,000,........} {0} $\cup$ {1}

= {$\in$, 0,00,000,........} {0,1}

= {0, 00, 000, 0000,..........,1, 01, 001, 0001,...............}

Precedence Rule

Consider the RE $ab + c$. The language described by the RE can be thought of either $L(a)L(b+c)$ or $L(ab) \cup L(c)$ as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression- (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i) The star operator precedes concatenation and concatenation precedes union (+) operator.

ii) It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language $L(ab) \cup L(c)$ i.e. it should be grouped as $((ab)+c)$.

We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE $a(b+c)$ is $L(a)L(b+c)$.

Example : The RE ab^*+b is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* \cup L(b)$

Example : The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* \cup L(b)$.

Example : It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Example : The regular expression r =(00)*(11)*1 denotes the set of all strings with an even number of 0's followed by an odd number of 1's i.e. $L(r) = \{0^{2n}1^{2m+1} \mid n \geq 0,\ m \geq 0\}$

Note : The notation r^+ is used to represent the RE rr^*. Similarly, r^2 represents the RE rr, r^3 denotes $r^2 r$, and so on.

An arbitrary string over Σ = {0,1} is denoted as (0+1)*.

Exercise : Give a RE r over {0,1} s.t. $L(r)$={$\omega \in \Sigma^* \mid \omega$ has at least one pair of consecutive 1's}

Solution : Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes before is completely arbitrary. Considering these observations we can write the REs as (0+1)*11(0+1)*.

Example : Considering the above example it becomes clean that the RE (0+1)*11(0+1)*+(0+1)*00(0+1)* represents the set of string over {0,1} that contains the substring 11 or 00.

Example : Consider the RE 0*10*10*. It is not difficult to see that this RE describes the set of strings over {0,1} that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

Example : Consider the language of strings over {0,1} containing two or more 1's.

Solution : There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

i) 0*10*1(0+1)*

ii) (0+1)*10*10*

Example : Consider a RE *r* over {0,1} such that

$L(r)$ = { $\omega \in \{0,1\}^* \mid \omega$ has no pair of consecutive 1's}

Solution : Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0*100*. So it looks like the RE is (0*100*)*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is r = (0*100*)(1+ ∈)+0*(1+∈).

Alternative Solution :
The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as r = (0+10)*(1+∈).This is a shorter expression but represents the same language.

Regular Expression:

FA to regular expressions:

FA to RE (REs for Regular Languages) :

Lemma : If a language is regular, then there is a RE to describe it. i.e. if $L = L(M)$ for some DFA M, then there is a RE r such that $L = L(r)$.

Proof : We need to construct a RE r such that $L(r) = \{w \mid w \in L(M)\}$. Since M is a DFA, it has a finite no of states. Let the set of states of M is Q = {1, 2, 3,..., n} for some integer n. [Note : if the n states of M were denoted by some other symbols, we can always rename those to indicate as 1, 2, 3,..., n]. The required RE is constructed inductively.

Notations : $r_{ij}^{(k)}$ is a RE denoting the language which is the set of all strings w such that w is the label of a path from state i to state j $(1 \le i, j \le n)$ in M, and that path has no intermediate state whose number is greater then k. (i & j (begining and end pts) are not considered to be "intermediate" so i and /or j can be greater than k)

We now construct $r_{ij}^{(k)}$ inductively, for all $i, j \in Q$ starting at $k = 0$ and finally reaching $k = n$.

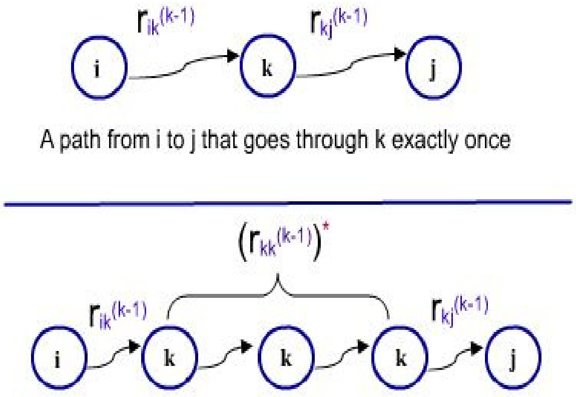
Basis : $k = 0$, $r_{ij}^{(0)}$ i.e. the paths must not have any intermediate state (since all states are numbered 1 or above). There are only two possible paths meeting the above condition **:**

1. A direct transition from state i to state j.
   - $r_{ij}^{(0)}$ = a if then is a transition from state i to state j on symbol the single symbol a.
   - $r_{ij}^{(0)} = a_1 + a_2 + \cdots + a_m$ if there are multiple transitions from state i to state j on symbols $a_1, a_2, \cdots, a_m$.
   - $r_{ij}^{(0)} = f$ if there is no transition at all from state i to state j.
2. All paths consisting of only one node i.e. when $i = j$. This gives the path of length 0 (i.e. the RE $\in$ denoting the string $\in$) and all self loops. By simply adding Î to various cases above we get the corresponding REs i.e.
   - $r_{ii}^{(0)} = \in + a$ if there is a self loop on symbol a in state i .
   - $r_{ii}^{(0)} = \in + a_1 + a_2 + \cdots + a_m$ if there are self loops in state i as multiple symbols $a_1, a_2, \cdots, a_m$.
   - $r_{ii}^{(0)} = \in$ if there is no self loop on state i.

Induction :

Assume that there exists a path from state i to state j such that there is no intermediate state whose number is greater than k. The corresponding Re for the label of the path is $r_{ij}^{(k)}$.
There are only two possible cases **:**

1. The path dose not go through the state k at all i.e. number of all the intermediate states are less than k. So, the label of the path from state i to state j is tha language described by the RE $r_{ij}^{(k-1)}$.
2. The path goes through the state k at least once. The path may go from i to j and k may appear more than once. We can break the into pieces as shown in the figure 7.



A path from i to j that goes through k exactly once

A path from i to j that goes through k more than once

Figure 7

1. The first part from the state i to the state k which is the first recurence. In this path, all intermediate states are less than k and it starts at iand ends at k. So the RE $r_{ik}^{(k-1)}$ denotes the language of the label of path.
2. The last part from the last occurence of the state k in the path to state j. In this path also, no intermediate state is numbered greater than k. Hence the RE $r_{kj}^{(k-1)}$ denoting the language of the label of the path.
3. In the middle, for the first occurence of k to the last occurence of k , represents a loop which may be taken zero times, once or any no of times. And all states between two consecutive k's are numbered less than k.

Hence the label of the path of the part is denoted by the RE $\left(r_{ij}^{(k-1)}\right)^*$.The label of the path from state i to state j is the concatenation of these 3 parts which is

$$r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}$$

Since either case 1 or case 2 may happen the labels of all paths from state i to j is denoted by the following RE

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} + r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}$$

We can construct $r_{ij}^{(k)}$ for all $i, j \in \{1,2,..., n\}$ in increasing order of k starting with the basis $k = 0$ upto $k = n$ since $r_{ij}^{(k)}$ depends only on expressions with a small superscript (and hence will be available). WLOG, assume that state 1 is the start state and $j_1, j_2, \cdots, j_m$ are the m final states where $ji \in \{1, 2, ... , n\}$, $1 \le i \le m$ and $m \le n$. According to the convention used, the language of the automatacan be denoted by the RE

$r_{1j_1}^{(n)} + r_{1j_2}^{(n)} + \cdots + r_{1j_m}^{(n)}$

Since $r_{1j_i}^{(n)}$ is the set of all strings that starts at start state 1 and finishes at final state j_i following the transition of the FA with any value of the intermediate state (1, 2, ... , n) and hence accepted by the automata.

Regular Grammar:

A *grammar* $G = (N, \Sigma, P, S)$ is right-linear if each production has one of the following three forms:

- $A \to cB$,
- $A \to c$,
- $A \to \in$

Where A, $B \in N'$ (with $A = B$ allowed) and $c \in \Sigma$. A *grammar G* is left-linear if each production has once of the following three forms.

$A \to Bc$, $A \to c$, $A \to \in$

A right or left-linear grammar is called a regular grammar.

Regular grammar and Finite Automata are equivalent as stated in the following theorem.

Theorem : A language L is regular iff it has a regular grammar. We use the following two lemmas to prove the above theorem.

Lemma 1 : If L is a regular language, then L is generated by some right-linear grammar.

Proof : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts L.

Let $Q = \{q_0, q_1, \cdots, q_n\}$ and $\Sigma = \{a_1, a_2, \cdots, a_m\}$.

We construct the right-linear grammar $G = (N, \Sigma, P, S)$ by letting

$N = Q$, $S = q_0$ and $P = \{A \to cB \mid \delta(A, c) = B\} \cup \{A \to c \mid \delta(A, c) \in B\}$

[Note: If $B \in F$, then $B \to \in P$]

Let $w = a_1 a_2 \ldots a_k \in L(M)$. For M to accept w, there must be a sequence of states $q_0, q_1, \ldots, q_k$ such that

$\delta(q_0, a_1) = q_1$

$\delta(q_1, a_2) = q_2$

$\cdots$

$\delta(q_{k-1}, a_k) = q_k$

and $q_k \in F$

By construction, the grammar G will have one production for each of the above transitions. Therefore, we have the corresponding derivation.

$$S = q_0 \underset{G}{\Rightarrow} a_1 q_1 \underset{G}{\Rightarrow} a_1 a_2 q_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k q_k \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k = w$$

Hence $w \in L(g)$.

Conversely, if $w = a_1 a_2 \ldots a_k \in L(G)$, then the derivation of w in G must have the form as given above. But, then the construction of G from M implies that

$\delta(q_0, a_1 a_2 \cdots a_k) = q_k$, where $q_k \in F$, completing the proof.

Lemma 2 : Let $G = (N, \Sigma, P, S)$ be a right-linear grammar. Then $L(G)$ is a regular language.

Proof: To prove it, we construct a FA M from G to accept the same language.

$M = (Q, \Sigma, \delta, q_0, F)$ is constructed as follows:

$Q = N \cup \{q_f\}$ (q_f is a special sumbol not in N)

$q_0 = S$, $F = \{q_f\}$

For any $q \in N$ and $a \in \Sigma$ and δ is defined as

$\delta(q, a) = \{p \mid q \rightarrow ap \in P\}$ if $q \rightarrow a \notin P$

and $\delta(q, a) = \{p \mid q \rightarrow ap \in P\} \cup \{q_f\}$, if $q \rightarrow a \in P$.

We now show that this construction works.

Let $w = a_1 a_2 \ldots a_k \in L(G)$. Then there is a derivation of w in G of the form

$$S \underset{G}{\Rightarrow} a_1q_1 \underset{G}{\Rightarrow} a_1a_2q_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1a_2 \cdots a_{k-1}a_k \left(= w\right)$$

By contradiction of M, there must be a sequence of transitions

$$\delta\left(q_0, a_1\right) = q_1$$
$$\delta\left(q_1, a_2\right) = q_2$$
$$\cdots$$
$$\delta\left(q_{k-1}, a_k\right) = q_f$$

implying that $w = a_1a_2 ... a_k \in L(M)$ i.e. w is accepted by M.

Conversely, if $w = a_1a_2 \cdots a_k$ is accepted by M, then because q_f is the only accepting state of M, the transitions causing w to be accepted by M will be of the form given above. These transitions corresponds to a derivationof w in the grammar G. Hence $w \in L\left(G\right)$, completing the proof of the lemma.

Given any left-linear grammar G with production of the form $A \rightarrow cB \mid c \mid \in$, we can construct from it a right-linear grammar $\hat{G}$ by replacing every production of G of the form $A \rightarrow cB$ with $A \rightarrow Bc$

It is easy to prove that $L\left(G\right) = \left(L\left(\hat{G}\right)\right)^R$. Since $\hat{G}$ is right-linear, $L\left(\hat{G}\right)$ is regular. But then so are $\left(L\left(\hat{G}\right)\right)^R$ i.e. $L(G)$ because regular languages are closed under reversal.

Putting the two lemmas and the discussions in the above paragraph together we get the proof of the theorem-

A language L is regular iff it has a regular grammar
Example : Consider the grammar

$$G: S \rightarrow 0A \mid 0$$
$$A \rightarrow 1S$$

It is easy to see that G generates the language denoted by the regular expression (01)*0.
The construction of lemma 2 for this grammar produces the follwoing FA.
This FA accepts exactly (01)*1.

Decisions Algorithms for CFL

In this section, we examine some questions about CFLs we can answer. A CFL may be represented using a CFG or PDA. But an algorithm that uses one representation can be made to work for the others, since we can construct one from the other.

Testing Emptiness :

Theorem : There are algorithms to test emptiness of a CFL.

Proof : Given any CFL L, there is a CFG G to generate it. We can determine, using the construction described in the context of elimination of useless symbols, whether the start symbol is useless. If so, then $L(G) = \phi$; otherwise not.

Testing Membership :

Given a CFL L and a string x, the membership, problem is to determine whether $x \in L$?

Given a PDA P for L, simulating the PDA on input string x doesnot quite work, because the PDA can grow its stack indefinitely on $\in$ input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG $G = (N, \Sigma, P, S)$ is given such that $L = L(G)$.

Let us first present a simple but inefficient algorithm.

Convert G to $G' = (N', \Sigma', P', S')$ in CNF generating $L(G) - \{\in\}$. If the input string $x = \in$, then we need to determine whether $S \overset{*}{\underset{G}{\Rightarrow}} \in$ and it can easily be done using the technique given in the context of elimination of $\in$-production. If , $x \neq \in$ then $x \in L(G')$ iff $x \in L(G)$. Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form. Hence, if the length of the input string x is n, then it takes exactly n steps to derive x (provided x is in $L(G')$).

Let the maximum number of productions for any nonterminal in G' is K. So at every step in derivation, there are atmost k choices. We may try out all these choices, systematically., to derive the string x in G'. Since there are atmost $K^{|x|}$ i.e. K^n choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time) membership algorithm.

Pumping Lemma:

Limitations of Finite Automata and Non regular Languages :

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language $L = \{a^n b^n | n \geq 0\}$

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between a's and b's how many a's it has seen so far. Because it would have to compare that with the number of b's to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of a's is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that FA s have finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to be remembered at any point is strictly limited. The argument given above to show that a^nb^n is non regular is informal. We now present a formal method for showing that certain languages such as a^nb^n are non regular

Properties of CFL's

Closure properties of CFL:

We consider some important closure properties of CFLs.

Theorem : If L_1 and L_2 are CFLs then so is $L_1 \cup L_2$

Proof : Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be CFGs generating. Without loss of generality, we can assume that $N_1 \cap N_2 = \phi$. Let S_3 is a nonterminal not in N_1 or N_2. We construct the grammar $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$,

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\}$

We now show that $L(G_3) = L(G_1) \cup L(G_2) = L_1 \cup L_2$

Thus proving the theorem.

Let $w \in L_1$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$. All productions applied in their derivation are also in G_3. Hence $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. $w \in L(G_3)$

Similarly, if $w \in L_2$, then $w \in L(G_3)$

Thus $L_1 \cup L_2 \subseteq L(G_3)$.

Conversely, let $w \in L(G_3)$. Then $S_3 \overset{*}{\underset{G_3}{\Rightarrow}} w$ and the first step in this derivation must be either $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1$ or $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_2$. Considering the former case, we have $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$

Since N_1 and N_2 are disjoint, the derivation $S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ must use the productions of P_1 only (which are also in P_3) Since $S_1 \in N_1$ is the start symbol of G_1. Hence, $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$ giving $w \in L(G_1)$.

Using similar reasoning, in the latter case, we get $w \in L(G_2)$. Thus $L(G_3) \subseteq L_1 \cup L_2$.

So, $L(G_3) = L_1 \cup L_2$, as claimed

Theorem : If L_1 and L_2 are CFLs, then so is $L_1 L_2$.

Proof : Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be the CFGs generating L_1 and L_2 respectively. Again, we assume that N_1 and N_2 are disjoint, and S_3 is a nonterminal not in N_1 or N_2. we construct the CFG $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 S_2\}$

We claim that $L(G_3) = L(G_1) L(G_2) = L_1 L_2$

To prove it, we first assume that $x \in L_1$ and $y \in L_2$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$. We can derive the string xy in G_3 as shown below.

$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 S_2 \overset{*}{\underset{G_3}{\Rightarrow}} x S_2 \overset{*}{\underset{G_3}{\Rightarrow}} xy$

since $P_1 \subseteq P$ and $P_2 \subseteq P$. Hence $L_1 L_2 \subseteq L(G_3)$.

For the converse, let $w \in L(G_3)$. Then the derivation of w in G_3 will be of the form

$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1S_2 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. the first step in the derivation must see the rule $S_3 \rightarrow S_1S_2$. Again, since N_1 and N_2 are disjoint and $S_1 \in N_1$ and $S_2 \in N_2$, some string x will be generated from S_1 using productions in P_1 (which are also in P_3) and such that $xy = w$.

Thus $S_3 \Rightarrow S_1S_2 \Rightarrow xS_2 \Rightarrow xy = w$

Hence $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$.

This means that w can be divided into two parts x, y such that $x \in L_1$ and $y \in L_2$. Thus $w \in L_1L_2$.This completes the proof

Theorem : If L is a CFL, then so is L^*.

Proof : Let $G = (N, \Sigma, P, S)$ be the CFG generating L. Let us construct the CFG $G' = (N, \Sigma, P', S)$ from G where $P' = P \cup \{S \rightarrow SS \mid \in\}$.

We now prove that $L(G') = (L(G))^* = L^*$, which prove the theorem.

G' can generate $\in$ in one step by using the production $S \rightarrow \in$ since $P \subseteq P'$, G' can generate any string in L.

Let $w \in L^n$ for any $n > 1$ we can write $w = w_1w_2 \cdots w_n$ where $w_i \in L$ for $1 \le i \le n$. w can be generated by G' using following steps.

$$S \overset{n-1}{\underset{G}{\Rightarrow}} SS\cdots S \overset{*}{\underset{G}{\Rightarrow}} w_1ss\cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2ss\cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2\cdots w_n = w$$

First $(n\text{-}1)$-steps uses the production $S \rightarrow SS$ producing the sentential form of n numbers of S 's. The nonterminal S in the i-th position then generates w_i using production in P (which are also in P')

It is also easy to see that G can generate the empty string, any string in L and any string $w \in L^n$ for $n > 1$ and none other.

Hence $L(G') = (L(G))^* = L^*$

Theorem : CFLs are not closed under intersection

Proof : We prove it by giving a counter example. Consider the language $L_1 = \{a^ib^ic^j \mid i, j \ge 0\}$.The following CFG generates L_1 and hence a CFL

$S \rightarrow XC$

$X \rightarrow aXb \mid \in$

$C \rightarrow cC \mid \in$

The nonterminal X generates strings of the form $a^n b^n, n \geq 0$ and C generates strings of the form c^m , $m \geq 0$. These are the only types of strings generated by X and C. Hence, S generates L_1.

Using similar reasoning, it can be shown that the following grammar $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$ and hence it is also a CFL.

$S \rightarrow AX$

$A \rightarrow aA \mid \in$

$X \rightarrow bXc \mid \in$

But, $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ and is already shown to be not context-free.

Hence proof.

Theorem : A CFL's are not closed under complementations

Proof : Assume, for contradiction, that CFL's are closed under complementation. SInce, CFL's are also closed under union, the language $\overline{\overline{L_1} \cup \overline{L_2}}$, where L_1 and L_2 are CFL's must be CFL. But by DeMorgan's law

$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$

This contradicts the already proved fact that CFL's are not closed under intersection.

But it can be shown that the CFL's are closed under intersection with a regular set.

Theorem : If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

Proof : Let $P = (Q_p, \Sigma, \Gamma, \delta_p, q_p, z_0, F_p)$ be a PDA for L and let $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ be a DFA for R.

We construct a PDA M from P and D as follows

$M = (Q_p \times Q_D, \Sigma, \Gamma, \delta_M, (q_p, q_D), z_0, F_p \times F_D)$

where δ_M is defined as

$\delta_M((p,q), a, X)$ contains $((r,s), \alpha)$ iff

$\delta_D(q,a) = s$ and $\delta_p(p,a,X)$ contains (r,α)

The idea is that M simulates the moves of P and D parallely on input w, and accepts w iff both P and D accepts. That means, we want to show that

$$L(M) = L(P) \cap L(D) = L \cap R$$

We apply induction on n, the number of moves, to show that

$$\left((q_p, q_D), w, z_0\right) \overset{n}{\underset{M}{\mapsto}} \left((p,q), \in, \gamma\right)$$ iff

$$\left(q_p, w, z_0\right) \overset{n}{\underset{P}{\mapsto}} \left(p, \in, \gamma\right)$$ and $$\hat{\delta}(q_D, w) = q$$

Basic Case is n=0. Hence $p = q_p$, $q = q_D$, $\gamma = z_0$ and $w = \in$. For this case it is trivially true

Inductive hypothesis : Assume that the statement is true for n -1.

Inductive Step : Let $w = xa$ and

Let $$\left((q_p, q_D), x_a, z_0\right) \overset{n-1}{\underset{M}{\mapsto}} \left((p', q'), a, \alpha\right) \overset{1}{\underset{M}{\mapsto}} \left((p,q), \in, \gamma\right)$$

By inductive hypothesis, $\left(q_p, x, z_0\right) \overset{n-1}{\underset{P}{\mapsto}} \left(p', \in, \alpha\right)$ and $\hat{\delta}_D(q_D, x) = q'$

From the definition of δ_M and considering the n-th move of the PDA M above, we have

$$\delta_p(p', a, \alpha) = (p, \in, \gamma)$$ and $$\delta_D(q', a) = q$$

Hence $\left(q_p, xa, z_0\right) \overset{n-1}{\underset{p}{\mapsto}} \left(p', a, \alpha\right) \overset{1}{\underset{p}{\mapsto}} \left(p, \in, \gamma\right)$ and $\hat{\delta}_D(q_D, w) = q$

If $p \in F_p$ and $q \in F_D$, then $p,q \in F_p \times F_D$ and we got that if M accepts w, then both P and D accepts it.
We can show that converse, in a similar way. Hence $L \cap R$ is a CFL (since it is accepted by a PDA M)
This property is useful in showing that certain languages are not context-free.

Example : Consider the language

$$L = \left\{ w \in (a,b,c)^* \mid w \text{ contains equal number of } a\text{'s}, b\text{'s and } c\text{'s} \right\}$$

Intersecting L with the regular set $R = a^* b^* c^*$, we get

$L \cap R = L \cap a^* b^* c^*$

$\quad = \{a^n b^n c^n \mid n \geq 0\}$

Which is already known to be not context-free. Hence L is not context-free

Theorem : CFL's are closed under reversal. That is if L is a CFL, then so is L^R

Proof : Let the CFG $G = (N, \Sigma, P, S)$ generates L. We construct a CFG $G' = (N, \Sigma, P', S)$ where $P = \{A \to \alpha | A \to \alpha^R \in P\}$. We now show that $L(G') = L^R$, thus proving the theorem.

We need to prove that

$A \overset{n}{\underset{G}{\Rightarrow}} \alpha$ iff $A \overset{n}{\underset{G}{\Rightarrow}} \alpha^R$.

The proof is by induction on n, the number of steps taken by the derivation. We assume, for simplicity (and of course without loss of generality), that G and hence G' are in CNF.

The basis is n=1 in which case it is trivial. Because α must be either $a \in \Sigma$ or BC with $B, C \in N$.

Hence $A \overset{1}{\underset{G}{\Rightarrow}} a$ iff $A \overset{1}{\underset{G}{\Rightarrow}} a$

Assume that it is true for (n-1)-steps. Let $A \overset{n}{\underset{G}{\Rightarrow}} \alpha$. Then the first step must apply a rule of the form $A \to BC$ and it gives

$A \overset{1}{\underset{G}{\Rightarrow}} BC \overset{n-1}{\underset{G}{\Rightarrow}} \beta\gamma = \alpha$ where $B \overset{*}{\underset{G}{\Rightarrow}} \beta^R$ and $C \overset{*}{\underset{G}{\Rightarrow}} \gamma^R$

By constructing of G', $A \to CB \in P'$

Hence

$A \overset{1}{\underset{G}{\Rightarrow}} CB \overset{n-1}{\underset{G}{\Rightarrow}} \gamma^R \beta^R = \alpha^R$

The converse case is exactly similar

Substitution :

$\forall a \in \Sigma$, let L_a be a language (over any alphabet). This defines a function S, called substitution, on Σ which is denoted as $s(a) = L_a$ - for all $a \in \Sigma$

This definition of substitution can be extended further to apply strings and langauge as well.

If $w = a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is a string in Σ^*, then

$s(w) = s(a_1 a_2 \cdots a_n) = s(a_1) s(a_2) \cdots s(a_n)$.

Similarly, for any language L,

$s(L) = \{s(w) \mid w \in L\}$

The following theorem shows that CFLs are closed under substitution.

Thereom : Let $L \subseteq \Sigma^*$ is a CFL, and s is a substitution on Σ such that $s(a) = L_a$ is a CFL for all $a \in \Sigma$, thus $s(L)$ is a CFL

Proof : Let $L = L(G)$ for a CFG $G = (N, \Sigma, P, S)$ and for every $a \in \Sigma$, $L_a = L(G_a)$ for some $G_a = (N_a, \Sigma_a, P_a, S_a)$. Without loss of generality, assume that the sets of nonterminals N and N_a's are disjoint.

Now, we construct a grammar G', generating $s(L)$, from G and G_a's as follows :

- $G' = (N', \Sigma', P', S)$
- $N' = N \cup \bigcup_{a_i \in \Sigma} N_{a_i}$
- $\Sigma' = \bigcup_{a_i \in \Sigma} \Sigma_{a_i}$
- P' consists of
  1. $\bigcup_{a_i \in \Sigma} P_{a_i}$ and
  2. The production of P but with each terminal a in the right hand side of a production replaced by S_a everywhere.

We now want to prove that this construction works i.e. $w \in L(G')$ iff $w \in s(L)$.

If Part : Let $w \in s(L)$ then according to the definition there is some string $x = a_1 a_2 \cdots a_n \in L$ and $x_i \in S(a_i)$ for $i = 1, 2, \cdots, n$ such that $w = x_1 x_2 \cdots x_n \left(= s(a_1) s(a_2) \cdots s(a_n)\right)$

We will show that $S \overset{*}{\underset{G}{\Rightarrow}} w$.

From the construction of G', we find that, there is a derivation $S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ corresponding to the string $x = a_1 a_2 \cdots a_n$ (since G' contains all productions of G but every ai replaced with S_{a_i} in the RHS of any production).

Every S_{a_i} is the start symbol of G_{a_i} and all productions of G_{a_i} are also included in G'.
Hence

$$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$

$$\overset{*}{\underset{G}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$$

$$\overset{*}{\Rightarrow} x_1 x_2 \cdots x_n = w$$

Therefore, $w \in L(G')$

(Only-if Part) Let $w \in L(G')$. Then there must be a derivative as follows :

$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ (using the production of G include in G' as modified by (step 2) of the construction of P'.)

Each S_{a_i} ($i = 1, 2, \cdots, n$) can only generate a string $x_i \in L_{a_i}$, since each N_{a_i}'s and N are disjoin. Therefore, we get

$$S \overset{*}{\underset{G}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$ since $S_{a_1} \overset{*}{\underset{G_{a_1}}{\Rightarrow}} x_1$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 x_2 S_{a_3} \cdots S_{a_n}$ since $S_{a_2} \overset{*}{\underset{G_{a_2}}{\Rightarrow}} x_2$

$\overset{*}{\underset{G}{\Rightarrow}} x_1 x_2 \cdots x_n$

$= w$

The string $w = x_1 x_2 \cdots x_n$ is formed by substituting strings x_i for each a_i's and hence $w \in s(L)$.

Theorem : CFL's are closed under homomorphism

Proof : Let $L \subseteq \Sigma^*$ be a CFL, and *h* is a homomorphism on Σ i.e $h : \Sigma \rightarrow \Delta^*$ for some alphabets Δ. consider the following substitution S:Replace each symbol $a \in \Sigma$ by the language consisting of the only string h(a), i.e. $s(a) = \{h(a)\}$ for all $a \in \Sigma$. Then, it is clear that, $h(L) = s(L)$. Hence, CFL's being closed under substitution must also be closed under homomorphism.

Grammar

A grammar is a mechanism used for describing languages. This is one of the most simple but yet powerful mechanism. There are other notions to do the same, of course.

In everyday language, like English, we have a set of symbols (alphabet), a set of words constructed from these symbols, and a set of rules using which we can group the words to construct meaningful sentences. The grammar for English tells us what are the words in it and the rules to construct sentences. It also tells us whether a particular sentence is well-formed (as per the grammar) or not. But even if one follows the rules of the english grammar it may lead to some sentences which are not meaningful at all, because of impreciseness and ambiguities involved in the language. In english grammar we use many other higher level constructs like noun-phrase, verb-phrase, article, noun, predicate, verb etc. A typical rule can be defined as

$$< \text{sentence} > \rightarrow < \text{noun-phrase} > < \text{predicate} >$$

meaning that "a sentence can be constructed using a 'noun-phrase' followed by a predicate".

Some more rules are as follows:

$$< \text{noun-phrase} > \rightarrow < \text{article} >< \text{noun} >$$

$$< \text{predicate} > \rightarrow < \text{verb} >$$

with similar kind of interpretation given above.

If we take {a, an, the} to be <article>; cow, bird, boy, Ram, pen to be examples of <noun>; and eats, runs, swims, walks, are associated with <verb>, then we can construct the sentence- a cow runs, the boy eats, an pen walks- using the above rules. Even though all sentences are well-formed, the last one is not meaningful. We observe that we start with the higher level construct <sentence> and then reduce it to <noun-phrase>, <article>, <noun>, <verb> successively, eventually leading to a group of words associated with these constructs.

These concepts are generalized in formal language leading to formal grammars. The word 'formal' here refers to the fact that the specified rules for the language are explicitly stated in terms of what strings or symbols can occur. There can be no ambiguity in it.

Formal definitions of a Grammar

A grammar G is defined as a quadruple.

$$G = (N, \Sigma, P, S)$$

N is a non-empty finite set of non-terminals or variables,

Σ is a non-empty finite set of terminal symbols such that $N \cap \Sigma = \phi$

$S \in N$, is a special non-terminal (or variable) called the start symbol, and $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ is a finite set of production rules.

The binary relation defined by the set of production rules is denoted by $\rightarrow$, i.e. $\alpha \rightarrow \beta$ iff $(\alpha, \beta) \in P$.

In other words, P is a finite set of production rules of the form $\alpha \rightarrow \beta$, where $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^*$

Production rules:

The production rules specify how the grammar transforms one string to another. Given a string $\delta\alpha\gamma$, we say that the production rule $\alpha \rightarrow \beta$ is applicable to this string, since it is possible to use the rule $\alpha \rightarrow \beta$ to rewrite the α (in $\delta\alpha\gamma$) to β obtaining a new string $\delta\beta\gamma$. We say that $\delta\alpha\gamma$ derives $\delta\beta\gamma$ and is denoted as

$$\delta\alpha\gamma \Rightarrow \delta\beta\gamma$$

Successive strings are dervied by applying the productions rules of the grammar in any arbitrary order. A particular rule can be used if it is applicable, and it can be applied as many times as described.

We write $\alpha \overset{*}{\Rightarrow} \beta$ if the string β can be derived from the string α in zero or more steps; $\alpha \overset{+}{\Rightarrow} \beta$ if β can be derived from α in one or more steps.

By applying the production rules in arbitrary order, any given grammar can generate many strings of terminal symbols starting with the special start symbol, S, of the grammar. The set of all such terminal strings is called the language generated (or defined) by the grammar.

Formaly, for a given grammar $G = (N, \Sigma, P, S)$ the language generated by G is

$$L(G) = \left\{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \right\}$$

That is $w \in L(G)$ iff $S \overset{*}{\Rightarrow} w$.

If $w \in L(G)$, we must have for some $n \geq 0$, $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_n = w$, denoted as a derivation sequence of *w*, The strings $S = \alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_n = w$ are denoted as sentential forms of the derivation.

Example : Consider the grammar $G = (N, \Sigma, P, S)$, where $N = \{S\}$, $\Sigma = \{a, b\}$ and P is the set of the following production rules

$$\{ S \rightarrow ab, S \rightarrow aSb \}$$

Some terminal strings generated by this grammar together with their derivation is given below.

$S \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aabb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

It is easy to prove that the language generated by this grammar is

$$L(G) = \{a^i b^i \mid i \geq 1\}$$

By using the first production, it generates the string ab (for i =1).

To generate any other string, it needs to start with the production $S \rightarrow aSb$ and then the non-terminal S in the RHS can be replaced either by ab (*in which we get the string aabb*) or the same production $S \rightarrow aSb$ can be used one or more times. Every time it adds an '*a*' to the left and a '*b*' to the right of S, thus giving the sentential form $a^i S b^i,\ i \geq 1$. When the non-terminal is replaced by ab (which is then only possibility for generating a terminal string) we get a terminal string of the form $a^i b^i,\ i \geq 1$.

There is no general rule for finding a grammar for a given language. For many languages we can devise grammars and there are many languages for which we cannot find any grammar.

Example: Find a grammar for the language $L = \{a^n b^{n+1} \mid n \geq 1\}$.

It is possible to find a grammar for L by modifying the previous grammar since we need to generate an extra b at the end of the string $a^n b^n,\ n \geq 1$. We can do this by adding a production $S \rightarrow Bb$ where the non-terminal B generates $a^i b^i,\ i \geq 1$ as given in the previous example.

Using the above concept we devise the follwoing grammar for L.

$G = (N, \Sigma, P, S)$ where, $N = \{ S, B \}$, $P = \{ S \rightarrow Bb, B \rightarrow ab, B \rightarrow aBb \}$

Parse Trees:

There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. But perhaps more importantly, the tree, known as a "parse tree"

Construction of a Parse tree:

Let us fix on a grammar $G = (V, T, P, S)$. The *parse trees* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.

2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.

3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \rightarrow X_1X_2 \cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \rightarrow \epsilon$ is a production of G.

Example 5.10: Figure 5.5 shows a parse tree for the palindrome grammar of Fig. 5.1. The production used at the root is $P \rightarrow 0P0$, and at the middle child of the root it is $P \rightarrow 1P1$. Note that at the bottom is a use of the production $P \rightarrow \epsilon$. That use, where the node labeled by the head has one child, labeled ϵ, is the only time that a node labeled ϵ can appear in a parse tree. □

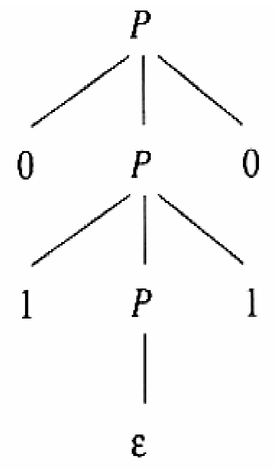


Figure 5.5: A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

Yield of a Parse tree:

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root variable. The fact that the yield is derived from the root will be proved shortly. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ϵ.

2. The root is labeled by the start symbol.

Ambiguity in languages and grammars:

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

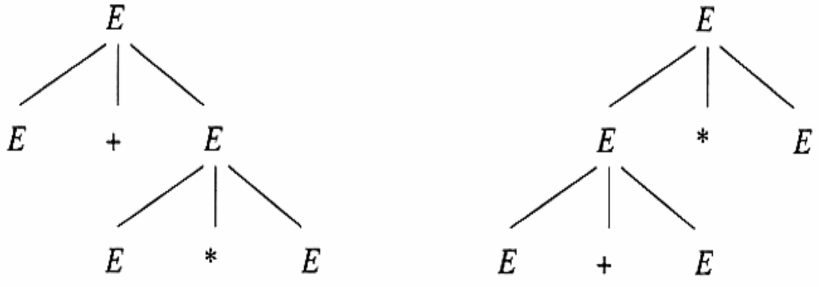
grammar lets us generate expressions with any sequence of $*$ and $+$ operators, and the productions $E \to E + E \mid E * E$ allow us to generate these expressions in any order we choose.

Example 5.25: For instance, consider the sentential form $E + E * E$. It has two derivations from E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$

2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$. Figure 5.17 shows the two parse trees, which we should note are distinct trees.



(a)

(b)

Figure 5.17: Two parse trees with the same yield

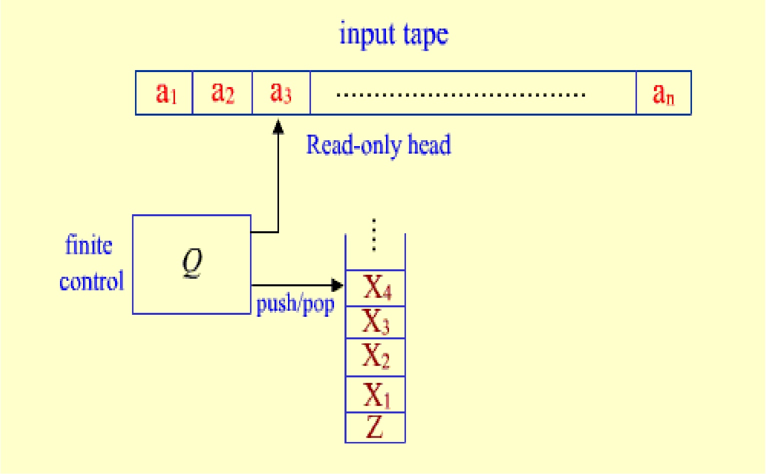
we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

MODULE-III

Push down automata:

Regular language can be charaterized as the language accepted by finite automata. Similarly, we can characterize the context-free language as the langauge accepted by a class of machines called "Pushdown Automata" (PDA). A pushdown automation is an extension of the NFA.

It is observed that FA have limited capability. (in the sense that the class of languages accepted or characterized by them is small). This is due to the "finite memory" (number of states) and "no external memory" involved with them. A PDA is simply an NFA augmented with an "external stack memory". The addition of a stack provides the PDA with a last-in, first-out memory management cpapability. This "Stack" or "pushdown store" can be used to record a potentially unbounded information. It is due to this memory management capability with the help of the stack that a PDA can overcome the memory limitations that prevents a FA to accept many interesting languages like $\{a^n b^n \mid n \geq 0\}$. Although, a PDA can store an unbounded amount of information on the stack, its access to the information on the stack is limited. It can push an element onto the top of the stack and pop off an element from the top of the stack. To read down into the stack the top elements must be popped off and are lost. Due to this limited access to the information on the stack, a PDA still has some limitations and cannot accept some other interesting languages.



As shown in figure, a PDA has three components: an input tape with read only head, a finite control and a pushdown store.

The input head is read-only and may only move from left to right, one symbol (or cell) at a time. In each step, the PDA pops the top symbol off the stack; based on this symbol, the input symbol it is currently reading, and its present state, it can push a sequence of symbols onto the stack, move its read-only head one cell (or symbol) to the right, and enter a new state, as defined by the transition rules of the PDA.

PDA are nondeterministic, by default. That is, $\in$- transitions are also allowed in which the PDA can pop and push, and change state without reading the next input symbol or moving its read-only head. Besides this, there may be multiple options for possible next moves.

Formal Definitions : Formally, a PDA M is a 7-tuple $M = \left(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\right)$ where,

- Q is a finite set of states,
- Σ is a finite set of input symbols (input alphabets),
- Γ is a finite set of stack symbols (stack alphabets),
- δ is a transition function from $Q \times \left(\Sigma \cup \{\in\}\right) \times \Gamma$ to subset of $Q \times \Gamma^*$
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$, is the initial stack symbol, and
- $F^* \subseteq Q$, is the final or accept states.

Explanation of the transition function, δ:

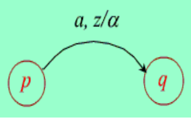
If, for any $a \in \Sigma$, $\delta(q, a, z) = \left\{\left(p_1, \beta_1\right), \left(p_2, \beta_2\right), \cdots, \left(p_k, \beta_k\right)\right\}$. This means intitutively that whenever the PDA is in state q reading input symbol a and z on top of the stack, it can nondeterministically for any i, $1 \le i \le k$

- go to state p_i
- pop z off the stack
- push β_i onto the stack (where $\beta_i \in \Gamma^*$) (The usual convention is that if $\beta_i = X_1 X_2 \cdots X_n$, then X_1 will be at the top and X_n at the bottom.)
- move read head right one cell past the current symbol a.

If $a = \in$, then $\delta(q, \in, z) = \left\{\left(p_1, \beta_1\right), \left(p_2, \beta_2\right), \cdots \left(p_k, \beta_k\right)\right\}$ means intitutively that whenver the PDA is in state q with z on the top of the stack regardless of the current input symbol, it can nondeterministically for any i, $1 \le i \le k$,

- go to state p_i
- pop z off the stack
- push β_i onto the stack, and
- leave its read-only head where it is.

State transition diagram : A PDA can also be depicted by a state transition diagram. The labels on the arcs indicate both the input and the stack operation. The transition $\delta(p,a,z) = \{(q,\alpha)\}$ for $a \in \Sigma \cup \{\in\}$, $p,q \in Q, z \in \Gamma$ and $\alpha \in \Gamma^*$ is depicted by



Final states are indicated by double circles and the start state is indicated by an arrow to it from nowhere.

Configuration or Instantaneous Description (ID) :

A configuration or an instantaneous description (ID) of PDA at any moment during its computation is an element of $Q \times \Sigma^* \times \Gamma^*$ describing the current state, the portion of the input remaining to be read (i.e. under and to the right of the read head), and the current stack contents. Only these three elements can affect the computation from that point on and, hence, are parts of the ID.

The start or inital configuartion (or ID) on input ω is (q_0, ω, z_0). That is, the PDA always starts in its start state, q_0 with its read head pointing to the leftmost input symbol and the stack containing only the start/initial stack symbol, z_0 .

The "next move relation" one figure describes how the PDA can move from one configuration to another in one step.

Formally,

$(q, a\omega, z\alpha) \vdash_M (p, \omega, \beta\alpha)$

iff $(p,\beta) \in \delta(q,a,z)$

'a' may be $\in$ or an input symbol.

Let I, J, K be IDs of a PDA. We define we write $I \overset{i}{\vdash}_M K$, if ID I can become K after exactly i moves. The relations $\overset{N}{\vdash}_M$ and $\overset{*}{\vdash}_M$ define as follows

$I \overset{0}{\vdash}_N K$

$I \overset{n+1}{\vdash}_N J$ if $\exists K$ such that $I \overset{n}{\vdash}_N K$ and $K \overset{1}{\vdash}_N J$

$I \overset{*}{\vdash}_N J$ if $\exists\ n \geq 0$ such that $I \overset{n}{\vdash}_N J$.

That is, $\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$. We say that $I \vdash_N^* J$ if the ID J follows from the ID I in zero or more moves.

(Note **:** subscript M can be dropped when the particular PDA M is understood.)

Language accepted by a PDA M

There are two alternative definiton of acceptance as given below.

1. Acceptance by final state **:**

Consider the PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Informally, the PDA M is said to accept its input ω by final state if it enters any final state in zero or more moves after reading its entire input, starting in the start configuration on input ω.

Formally, we define $L(M)$, the language accepted by final state to be

$$\{\ \omega \in \Sigma^* \mid (q_0, \omega, Z_0) \vdash_N^* (p, \in, \beta) \text{ for some } p \in F \text{ and } \beta \in \Gamma^*\}$$

2. Acceptance by empty stack (or Null stack) **:** The PDA M accepts its input ω by empty stack if starting in the start configuration on input ω, it ever empties the stack w/o pushing anything back on after reading the entire input. Formally, we define $N(M)$, the language accepted by empty stack, to be

$$\{\ \omega \in \Sigma^* \mid (q_0, \omega, z_0) \vdash_N^* (p, \in, \in) \text{ for some } p \in Q\ \}$$

Note that the set of final states, F is irrelevant in this case and we usually let the F to be the empty set i.e. F = Q.

Example 1 : Here is a PDA that accepts the language $\{a^n b^n \mid n \geq 0\}$.

$M = (Q, \Sigma, \Gamma, \delta, q_1, Z, F)$

$Q = \{q_1, q_2, q_3, q_4\}$

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

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

$F = \{q_1, q_4\}$, and δ consists of the following transitions

1. $\delta(q_1, a, z) = \{(q_2, az)\}$

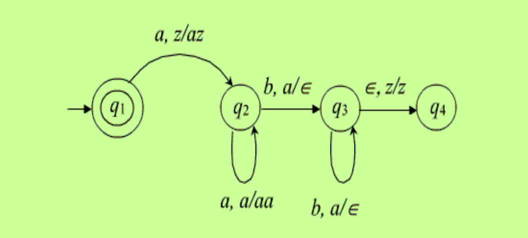
2. $\delta(q_2, a, a) = \{(q_2, aa)\}$

3. $\delta(q_2, b, a) = \{(q_3, \in)\}$

4. $\delta(q_3, b, a) = \{(q_3, \in)\}$

5. $\delta(q_3, \in, z) = \{(q_4, z)\}$

The PDA can also be described by the adjacent transition diagram.



Informally, whenever the PDA M sees an input a in the start state q_1 with the start symbol z on the top of the stack it pushes a onto the stack and changes state to q_2. (to remember that it has seen the first 'a'). On state q_2 if it sees anymore a, it simply pushes it onto the stack. Note that when M is on state q_2, the symbol on the top of the stack can only be a. On state q_2 if it sees the first b with a on the top of the stack, then it needs to start comparison of numbers of a's and b's, since all the a's at the begining of the input have already been pushed onto the stack. It start this process by popping off the a from the top of the stack and enters in state q3 (to remember that the comparison process has begun). On state q_3, it expects only b's in the input (if it sees any more a in the input thus the input will not be in the proper form of anbn). Hence there is no more on input a when it is in state q_3. On state q_3 it pops off an a from the top of the stack for every b in the input. When it sees the last b on state q3 (i.e. when the input is exaushted), then the last a from the stack will be popped off and the start symbol z is exposed. This is the only possible case when the input (i.e. on $\in$-input) the PDA M will move to state q_4 which is an accept state.
we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

Let the input be aabb. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

$(q_1, aabb, z) \vdash (q_2, abb, az)$ (using transition 1)

$\vdash (q_2, bb, aaz)$ (using transition 2)

$\vdash (q_3, b, az)$ (using transition 3)

$\vdash (q_3,\in,z)$ (using transition 4), $\vdash (q_4,\in,z)$ (using transition 5) , q_4 is final state. Hence , accept. So the string $aabb$ is rightly accepted by M

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be $aabab$.

$$(q_1,aabab,z) \vdash (q_2,abab,az)$$

$$\vdash (q_2,bab,aaz)$$

$$\vdash (q_3,ab,az)$$

No further move is defined at this point.

Hence the PDA gets stuck and the string $aabab$ is not accepted.

Example 2 : We give an example of a PDA M that accepts the set of balanced strings of parentheses [] by empty stack.
The PDA M is given below.

$M = (\{q\},\{[,]\},\{z,[\},\delta,q,z,\phi)$ where δ is defined as

$$\delta(q,[,z) = \{(q,[z)\}$$
$$\delta(q,[,]) = \{(q,[[)\}$$
$$\delta(q,],[) = \{(q,\in)\}$$
$$\delta(q,\in,z) = \{(q,\in)\}$$

Informally, whenever it sees a [, it will push the] onto the stack. (first two transitions), and whenever it sees a] and the top of the stack symbol is [, it will pop the symbol [off the stack. (The third transition). The fourth transition is used when the input is exhausted in order to pop z off the stack (to empty the stack) and accept. Note that there is only one state and no final state. The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].

$$(q,[[\,][\,]][\,],z) \vdash (q,[\,][\,]][\,],[z) \vdash (q,\,][\,]][\,],[[z) \vdash (q,[\,]][\,],[z) \vdash (q,\,]][\,],[[z)$$

$$\vdash (q,\,]][\,],[[z) \vdash (q,\,][\,],[\,z) \vdash (q,[\,],z) \vdash (q,\,],[z) \vdash (q,\in,z) \vdash (q,\in,\in)$$

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, since each kind of machine can simulate the other.Given any arbitrary PDA M that accpets the language L by final state or empty stack, we can always construct an equivalent PDA M with a single final state that accpets exactly the same language L. The construction process of M' from M and the proof of equivalence of M & M' are given below.

There are two cases to be considered.

CASE I : PDA M accepts by final state, Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ Let *qf* be a new state not in Q.

Consider the PDA $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, z_0, \{q_f\})$ where δ' as well as the following transition.

$\delta'(q, \in, X)$ contains (q_f, X) $\forall\ q \in F$ and $X \in \Gamma$. It is easy to show that M and M' are equivalent i.e. $L(M) = L(M')$

Let $\omega \in L(M)$. Then $(q_0, \omega, z_0) \vdash^{*}_{M} (q, \in, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*$

Then $(q_0, \omega, z_0) \vdash^{*}_{M'} (q, \in, \gamma) \vdash^{*}_{M'} (q_f, \in, \gamma)$

Thus M' accepts ω

Conversely, let M' accepts ω i.e. $\omega \in L(M')$, then $(q_0, \omega, z_0) \vdash^{*}_{M'} (q, \in, \gamma) \vdash^{1}_{M'} (q_f, \in, \gamma)$ for $q \in F$ M' inherits all other moves except the last one from M. Hence $(q_0, \omega, z_0) \vdash^{*}_{M} (q, \in, \gamma)$ for some $q \in F$.

Thus M accepts ω . Informally, on any input M' simulate all the moves of M and enters in its own final state q_f whenever M enters in any one of its final status in F. Thus M' accepts a string ω iff M accepts it.

CASE II : PDA M accepts by empty stack.

We will construct M' from M in such a way that M' simulates M and detects when M empties its stack. M' enters its final state q_f when and only when M empties its stack.Thus M' will accept a string ω iff M accepts.

Let $M' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q_0', X, \{q_f\})$ where $q_0',\ q_f \notin Q$ and $X \in \Gamma$ and δ' contains all the transition of δ, as well as the following two transitions.

1. $\delta'(q_0, \in, X) = \{(q_0, z_0 X)\}$ and

2. $\delta'(q, \in, X) = \{(q_f, \in)\}, \quad \forall q \in Q$

Transitions 1 causes M' to enter the initial configuration of M except that M' will have its own bottom-of-stack marker X which is below the symbols of M's stack. From this point onward M' will simulate every move of M since all the transitions of M are also in M'

If M ever empties its stack, then M' when simulating M will empty its stack except the symbol X at the bottom. At this point, M' will enter its final state q_f by using transition rule 2, thereby (correctly) accepting the input. We will prove that M and M' are equivalent.

Let M accepts ω. Then

$(q_0, \omega, z_0) \vdash_M^{*} (q, \in, \in)$ for some $q \in Q$. But then

$(q_0, \omega, X) \vdash_{M'}^{1} (q_0, \omega, z_0 X)$ (by transition rule 1)

$\vdash_{M'}^{*} (q, \in, X)$ (Since M' includes all the moves of M)

$\vdash_{M'}^{1} (q_f, \in, \in)$ (by transition rule 2)

Hence, M' also accepts ω. Conversely, let M' accepts ω.

Then $(q_0', \omega, X) \vdash_{M'}^{1} (q_0, \omega, z_0 X) \vdash_{M'}^{*} (q, \in, X) \vdash_{M'}^{1} (q_f, \in, \in)$ for some $q \in Q$

Every move in the sequence, $(q_0, \omega, z_0 X) \vdash_{M'}^{*} (q, \in, X)$ were taken from M.

Hence, M starting with its initial configuration will eventually empty its stack and accept the input i.e.

$(q_0, \omega, z_0) \vdash_M^{*} (q, \in, \in)$

Equivalence of PDA's and CFG's:
We will now show that pushdown automata and context-free grammars are equivalent in expressive power, that is, the language accepted by PDAs are exactly the context-free languages. To show this, we have to prove each of the following:

i) Given any arbitrary CFG G there exists some PDA M that accepts exactly the same language generated by G.

ii) Given any arbitrary PDA M there exists a CFG G that generates exactly the same language accpeted by M.

(i) CFA to PDA

We will first prove that the first part i.e. we want to show to convert a given CFG to an equivalent PDA.

Let the given CFG is $G = (N, \Sigma, P, S)$. Without loss of generality we can assume that G is in Greibach Normal Form i.e. all productions of G are of the form .

$A \rightarrow cB_1B_2 \cdots B_k$ where $c \in \Sigma \cup \{\varepsilon\}$ and $k \geq 0$.

From the given CFG G we now construct an equivalent PDA M that accepts by empty stack. Note that there is only one state in M. Let

$M = (\{q\}, \Sigma, N, \delta, q, S, \phi)$, where

- q is the only state
- Σ is the input alphabet,
- N is the stack alphabet ,
- q is the start state.
- S is the start/initial stack symbol, and δ, the transition relation is defined as follows

  For each production $A \rightarrow cB_1B_2 \cdots B_k \in P$, $(q, B_1B_2 \ldots B_k) \in \delta(q, c, A)$. We now want to show that M and G are equivalent i.e. $L(G)=N(M)$. i.e. for any $w \in \Sigma^*$. $w \in L(G)$ iff $w \in N(M)$.

If $w \in L(G)$, then by definition of $L(G)$, there must be a leftmost derivation starting with S and deriving w.

i.e. $S \overset{*}{\underset{G}{\Rightarrow}} w$

Again if $w \in N(M)$, then one sysmbol. Therefore we need to show that for any $w \in \Sigma^*$.

$S \overset{*}{\underset{G}{\Rightarrow}} w$ iff $(q, w, s) \vdash (q, \in, \in)$.

But we will prove a more general result as given in the following lemma. Replacing A by S (the start symbol) and γ by $\in$ gives the required proof.

Lemma For any $x, y \in \Sigma^*$, $\gamma \in N^*$ and $A \in N$, $A \overset{n}{\underset{G}{\Rightarrow}} x\gamma$ via a leftmost derivative iff $(q, xy, A) \vdash^n_M (q, y, \gamma)$.

Proof : The proof is by induction on n.

Basis : $n = 0$

$A \overset{0}{\underset{G}{\Rightarrow}} x\gamma$ iff $A = x\gamma$ i.e. $x = \in$ and $\gamma = A$

iff $(q, xy, A) = (q, y, \gamma)$

iff $(q, xy, A) \vdash_{M}^{0} (q, y, \gamma)$

Induction Step :

First, assume that $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$ via a leftmost derivation. Let the last production applied in their derivation is $B \to c\beta$ for some $c \in \Sigma \cup \{\in\}$ and $\beta \in N^*$.

Then, for some $\omega \in \Sigma^*$, $\alpha \in N^*$

$$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha \overset{1}{\underset{G}{\Rightarrow}} \omega c\beta\alpha = x\gamma$$

where $x = \omega c$ and $\gamma = \beta\alpha$

Now by the indirection hypothesis, we get,

$(q, \omega cy, A) \vdash_{M}^{n} (q, cy, B\alpha)$ ....(1)

Again by the construction of M, we get

$$(q, \beta) \in \delta(q, c, B)$$

so, from (1), we get

$$(q, \omega cy, A) \vdash_{M}^{n} (q, cy, B\alpha) \vdash_{M}^{1} (q, y, \beta\alpha)$$

since $x = \omega c$ and $\gamma = \beta\alpha$, we get $(q, xy, A) \vdash_{M}^{n+1} (q, y, \gamma)$

That is, if $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$, then $(q, xy, A) \vdash_{M}^{n+1} (q, y, \gamma)$. Conversely, assume that $(q, xy, A) \vdash_{M}^{n+1} (q, y, \gamma)$ and let

$\delta(q,c,B)=(q,\beta)$ be the transition used in the last move. Then for some $\omega\in\Sigma^*$, $c\in\Sigma\cup\{\in\}$ and $\alpha\in\Gamma^*$

$(q,\omega cy,A) \vdash_M^n (q,cy,B\alpha) \vdash_M^1 (q,y,\beta\alpha)$ where $x=\omega c$ and $\gamma=\beta\alpha$.

Now, by the induction hypothesis, we get

$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha$ via a leftmost derivation.

Again, by the construction of M, $B\to c\beta$ must be a production of G. [Since $(q,\beta)\in\delta(q,c,B)$]. Applying the production to the sentential form $\omega B\alpha$ we get

$$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha \overset{1}{\underset{G}{\Rightarrow}} \omega c\beta\alpha = x\gamma$$

i.e. $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$

via a leftmost derivation.

Hence the proof.

Example : Consider the CFG G in GNF

S → aAB
A → a / aA
B → a / bB

The one state PDA M equivalent to G is shown below. For convenience, a production of G and the corresponding transition in M are marked by the same encircled number.

(1) $S\to aAB$

(2) $A\to a$

(3) $A\to aA$

(4) $B\to a$

(5) $B\to bB$

$M=(\{q\},\{a,b\},\{S,A,B\},\delta,q,S,\Sigma)$. We have used the same construction discussed earlier

Some Useful Explanations :

Consider the moves of M on input *aaaba* leading to acceptance of the string.

Steps

1. $(q, aaaba, s) \overset{(1)}{\underset{M}{\mapsto}} (q, aaba, AB)$

2. $\overset{(2)}{\underset{M}{\mapsto}} (q, aba, AB)$

3. $\overset{(3)}{\underset{M}{\mapsto}} (q, ba, B)$

4. $\overset{(4)}{\underset{M}{\mapsto}} (q, a, B)$

5. $\overset{(5)}{\underset{M}{\mapsto}} (q, \in, \in)$ Accept by empty stack.

Note : encircled numbers here shows the transitions rule applied at every step.
Now consider the derivation of the same string under grammar G. Once again, the production used at every step is shown with encircled number.

$$S \overset{(1)}{\underset{G}{\Rightarrow}} aAB \overset{(3)}{\underset{G}{\Rightarrow}} aaAB \overset{(2)}{\underset{G}{\Rightarrow}} aaaB \overset{(5)}{\underset{G}{\Rightarrow}} aaabB \overset{(4)}{\underset{G}{\Rightarrow}} aaaba$$

Steps → 1 → 2 → 3 → 4 → 5

Observations:

- There is an one-to-one correspondence of the sequence of moves of the PDA M and the derivation sequence under the CFG G for the same input string in the sense that - number of steps in both the cases are same and transition rule corresponding to the same production is used at every step (as shown by encircled number).
- considering the moves of the PDA and derivation under G together, it is also observed that at every step the input read so far and the stack content together is exactly identical to the corresponding sentential form i.e.
  <what is Read><stack> = <sentential form>

Say, at step 2, Read so far = a
stack = AB

Sentential form = aAB From this property we claim that $(q,x,S) \vdash_M^{*} (q,\in,\alpha)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x\alpha$. If the claim is true, then apply with $\alpha = \in$ and we get $(q,x,S) \vdash_M^{*} (q,\in,\in)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x$ or $x \in N(M)$ iff $x \in L(G)$ (by definition)

Thus $N(M) = L(G)$ as desired. Note that we have already proved a more general version of the claim

PDA and CFG:

We now want to show that for every PDA M that accpets by empty stack, there is a CFG G such that $L(G) = N(M)$

we first see whether the "reverse of the construction" that was used in part (i) can be used here to construct an equivalent CFG from any PDA M.

It can be show that this reverse construction works only for single state PDAs.

- That is, for every one-state PDA M there is CFG G such that $L(G) = N(M)$. For every move of the PDA M $(q, B_1B_2\cdots B_K) \in \delta(q, c, A)$ we introduce a production $A \rightarrow cB_1B_2\cdots B_K$ in the grammar $G = (N, \Sigma, P, S)$ where $N = T$ and $S = z_0$.

we can now apply the proof in part (i) in the reverse direction to show that $L(G) = N(M)$.

But the reverse construction does not work for PDAs with more than one state. For example, consider the PDA M produced here to accept the langauge $\{a^n ba^n \mid n \geq 1\}$

$M = (\{p, q\}, \{a, b\}, \{z_0, A\}, \delta, p, z_0, \phi)$

Now let us construct CFG $G = (N, \Sigma, P, S)$ using the "reverse" construction.

(Note $N = \{z_0, A\}$, $S = z_0$).

Transitions in M	Corresponding Production in G
$a, z_0 / A$	$z_0 \rightarrow aA$
$a, A/AA$	$A \rightarrow aAA$
$b, A/A$	$A \rightarrow bA$
$a, A/\in$	$A \rightarrow a$

We can drive strings like aabaa which is in the language.

$s = z_0 \underset{G}{\Rightarrow} aA \underset{G}{\Rightarrow} aaAA \underset{G}{\Rightarrow} aabAA \underset{G}{\Rightarrow} aabaA \underset{G}{\Rightarrow} aabaa$

But under this grammar we can also derive some strings which are not in the language. e.g

$s = z_0 \Rightarrow aA \Rightarrow abA \Rightarrow abaAA \Rightarrow abaaA \Rightarrow abaaa$

and $s = z_0 \Rightarrow aA \Rightarrow aa$. But $aa, abaa \notin L(M)$

Therefore, to complete the proof of part (ii) we need to prove the following claim also.

Claim: For every PDA M there is some one-state PDA M' such that $N(M) = N(M')$.

It is quite possible to prove the above claim. But here we will adopt a different approach. We start with any arbitrary PDA M that accepts by empty stack and directly construct an equivalent CFG G.

PDA to CFG

We want to construct a CFG G to simulate any arbitrary PDA M with one or more states. Without loss of generality we can assume that the PDA M accepts by empty stack.

The idea is to use nonterminal of the form <PAq> whenever PDA M in state P with A on top of the stack goes to state q_0. That is, for example, for a given transition of the PDA corresponding production in the grammar as shown below,

And, we would like to show, in general, that $\langle pAq\rangle \overset{*}{\underset{G}{\Rightarrow}} \omega$ iff the PDA M, when started from state P with A on the top of the stack will finish processing ω, arrive at state q and remove A from the stack.

we are now ready to give the construction of an equivalent CFG G from a given PDA M. we need to introduce two kinds of producitons in the grammar as given below. The reason for introduction of the first kind of production will be justified at a later point. Introduction of the second type of production has been justified in the above discussion.

Let $M=(Q,\Sigma,\Gamma,\ \delta,\ q_0,\ z_0,\ \phi)$ be a PDA. We construct from M a equivalent CFG $G=(N,\Sigma,\ P,\ S)$

Where

- N is the set of nonterminals of the form <PAq> for $p,\ q\in Q$ and $A\in\Gamma$ and P contains the follwoing two kind of production
  1. $S\to\langle q_0z_0q\rangle\ \ \forall\ q\in Q$
  2. If $(q_1,\ B_1B_2\cdots B_n)\in\delta(q,\ a,\ A)$, then for every choice of the sequence $q_2,q_3,\cdots,q_{n+1}$, $q_i\in Q$, $2\le i\le n+1$.

Include the follwoing production

$$\langle q_A\ q_{n+1}\rangle\to a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\cdots\langle q_nB_nq_{n+1}\rangle$$

If $n=0$, then the production is $\langle q_A\ q_1\rangle\to a$. For the whole exercise to be meaningful we want $\langle q_A\ q_{n+1}\rangle\overset{*}{\underset{G}{\Rightarrow}}\omega$ means there is a sequence of transitions (for PDA M), starting in state q, ending in q_{n+1}, during which the PDA M consumes the input string ω and removes A from the stack (and, of course, all other symbols pushed onto stack in A's place, and so on.)

That is we want to claim that

$$\langle pAq\rangle\overset{*}{\underset{G}{\Rightarrow}}\omega \text{ iff } (p,\omega,\ A)\vdash(q,\in,\in)$$

If this claim is true, then let $p=q_0,\ A=z_0$ to get $\langle q_0\ z_0\ q\rangle\overset{*}{\underset{G}{\Rightarrow}}\omega$ iff $(q_0,\ \omega,\ z_0)\vdash(q,\in,\in)$ for some $q\in Q$. But for all $q\in Q$ we have $S\to\langle q_0z_0q\rangle$ as production in G. Therefore,

$S\overset{1}{\underset{G}{\Rightarrow}}\langle q_0 z_0 q\rangle\overset{*}{\underset{G}{\Rightarrow}}$ iff $(q_0,\ \omega,\ z_0)\vdash(q,\in,\in)$ i.e. $S\overset{*}{\underset{G}{\Rightarrow}}w$ iff PDA M accepts w by empty stack or $L(G) = N(M)$

Now, to show that the above construction of CFG G from any PDA M works, we need to prove the proposed claim.

Note: At this point, the justification for introduction of the first type of production (of the form $S\rightarrow\langle q_0 z_0 q\rangle$) in the CFG G, is quite clear. This helps use deriving a string from the start symbol of the grammar.

Proof : Of the claim $\langle PAq\rangle\overset{*}{\underset{G}{\Rightarrow}}w$ iff $(P,w,\ A)\vdash(q,\in,\in)$ for some $w\in\Sigma^*$, $A\in\Gamma$ and $p,\ q\ \in Q$
The proof is by induction on the number of steps in a derivation of G (which of course is equal to the number of moves taken by M). Let the number of steps taken is n.

The proof consists of two parts: ' *if* ' part and ' *only if* ' part. First, consider the ' *if* ' part

If $(P,w,\ A)\vdash(q,\in,\in)$ then $\langle PAq\rangle\overset{*}{\underset{G}{\Rightarrow}}w$.

Basis is $n=1$
Then $(P,w,\ A)\ (q,\in,\in)$. In this case, it is clear that $w\in\Sigma\cup\{\in'\}$. Hence, by construction $\langle PAq\rangle\rightarrow w$ is a production of G.

Then

Inductive Hypothesis :

$$\forall i<n\ (P,w,\ A)\vdash(q,\in,\in)\Rightarrow\langle PAq\rangle\overset{*}{\underset{G}{\Rightarrow}}w$$

Inductive Step : $(P,w,\ A)\vdash(q,\in,\in)$

For $n>1$, let $w = ax$ for some $a\in\Sigma\cup\{\in\}$ and $x\in\Sigma^*$ consider the first move of the PDA M which uses the general transition $(q_1,\ B_1B_2\cdots B_n)\in\delta(p,a,A)(p,w,A)$ =
$(p,\ ax,\ A)\vdash(q_1,\ x,\ B_1B_2\cdots B_n)\vdash(q,\in,\in)$. Now M must remove $B_1B_2\cdots B_n$ from stack while consuming x in the remaining n-1 moves.

Let $x = x_1x_2\cdots x_n$, where $x_1x_2\cdots x_n$ is the prefix of x that M has consumed when B_{i+1} first appears at top of the stack. Then there must exist a sequence of states in M (as per construction) $q_2, q_3,\cdots q_n, q_{n+1}$ (with $q_{n+1} = p$), such that

$(p, ax, A) \vdash (q_1,\ x,\ B_1B_2\cdots B_n) = (q_1,\ x_1x_2\cdots x_n,\ B_1B_2\cdots B_n)$

$(q_2,\ x_2x_3\cdots x_n,\ B_2B_3\cdots B_n)$ [This step implies $(q_1,\ x_1,\ B_1) \vdash (q_2, \in, \in)$]

$(q_3,\ x_3x_4\cdots x_n,\ B_3B_4\cdots B_n)$ [This step implies $(q_2,\ x_2,\ B_2) \vdash (q_3, \in, \in)$]

...

$\vdash (q_n,\ x_n,\ B_n)$

$\vdash (q_{n+1}, \in, \in) = (q, \in, \in)$

[Note: Each step takes less than or equal to n -1 moves because the total number of moves required assumed to be n-1.]

That is, in general

$(q_i,\ x_i,\ B_i) \vdash (q_{i+1}, \in, \in)$, $1 \le i \le n+1$.

So, applying inductive hypothesis we get

$\langle q_i B_i q_{i+1}\rangle \overset{*}{\underset{G}{\Rightarrow}} x_i$, $1 \le i \le n+1$. But corresponding to the original move $(p, w, A) = (p, ax, A) \vdash (q_1,\ x,\ B_1B_2\cdots B_n)$ in M we have added the following production in G.

We can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.
i) Let the input be $aabb$. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

$(q_1, aabb, z) \vdash (q_2, abb, az)$ (using transition 1) , $\vdash (q_2, bb, aaz)$ (using transition 2)

$\vdash (q_3, b, az)$ (using transition 3), $(q_3, \in, z)$ (using transition 4)

$\vdash (q_4, \in, z)$ (using transition 5) , q_4 is final state. Hence, accept.

So the string $aabb$ is rightly accepted by M.

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be $aabab$.

$(q_1, aabab, z) \vdash (q_2, abab, az)$

$\vdash (q_2, bab, aaz)$

$\vdash (q_3, ab, az)$

No further move is defined at this point.

Hence the PDA gets stuck and the string $aabab$ is not accepted.

The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].

$(q, [[\ ][\ ]][\ ], z) \vdash (q, [\ ][\ ]][\ ], [z) \vdash (q,][\ ]][\ ], [[z)$

$\vdash (q, [\ ]][\ ], [z) \vdash (q,]][\ ], [[z) \vdash (q,][\ ], [z)$

$\vdash (q, [\ ], z) \vdash (q,], [z) \vdash (q, \in, z) \vdash (q, \in, \in)$

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, since each kind of machine can simulate the other.Given any arbitrary PDA M that accpets the language L by final state or empty stack, we can always construct an equivalent PDA M with a single final state that accpets exactly the same language L. The construction process of M' from M and the proof of equivalence of M & M' are given below

There are two cases to be considered.

CASE 1 : PDA M accepts by final state, Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Let q_f be a new state not in Q. Consider the PDA $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, z_0, \{q_f\})$ where δ' as well as the following transition.

$\delta'(q, \in, X)$ contains (q_f, X) $\forall\ q \in F$ and $X \in \Gamma$. It is easy to show that M and M' are equivalent i.e.

$L(M) = L(M')$.

Let $\varpi \in L(M)$. Then $(q_0, \varpi, z_0) \vdash_M^* (q, \in, \gamma)$ for some $q \in F$ and $\Gamma \in \Sigma^*$

Then $(q_0, \varpi, z_0) \vdash_{M'}^* (q, \in, \gamma) \vdash_{M'}^* (q_f, \in, \gamma)$.

Thus M' accepts ϖ.

Conversely, let M' accepts ω i.e. $\omega \in L(M')$, then $(q_0, \omega, z_0) \vdash^{*}_{M'} (q, \in, \gamma) \vdash^{1}_{M'} (q_f, \in, \gamma)$ for some $q \in F$. M' inherits all other moves except the last one from M. Hence $(q_0, \omega, z_0) \vdash^{*}_{M} (q, \in, \gamma)$ for some $q \in F$.

Thus M accepts ω. Informally, on any input M' simulate all the moves of M and enters in its own final state q_f whenever M enters in any one of its final status in F. Thus M' accepts a string ω iff M accepts it.

CASE 2 : PDA M accepts by empty stack.

we will construct M' from M in such a way that M' simulates M and detects when M empties its stack. M' enters its final state q_f when and only when M empties its stack.Thus M' will accept a string ω iff M accepts.

Let $M' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q_0', X, \{q_f\})$ where $q_0', q_f \notin Q$ and $X \in \Gamma$ and δ' contains all the transition of δ, as well as the following two transitions.

1. $\delta'(q_0, \in, X) = \{(q_0, z_0X)\}$ and

2. $\delta'(q, \in, X) = \{(q_f, \in)\}, \quad \forall q \in Q$

Transitions 1 causes M' to enter the initial configuration of M except that M' will have its own bottom-of-stack marker X which is below the symbols of M's stack. From this point onward M' will simulate every move of M since all the transitions of M are also in M'.

If M ever empties its stack, then M' when simulating M will empty its stack except the symbol X at the bottom. At this point M', will enter its final state q_f by using transition rule 2, thereby (correctly) accepting the input. we will prove that M and M' are equivalent.

Let M accepts ω.

Then

$(q_0, \omega, z_0) \vdash (q, \in, \in)$ for some $q \in Q$. But then,

$(q_0, \omega, X) \vdash^{1}_{M'} (q_0, \omega, z_0X)$ (by transition rule 1)

$\vdash^{*}_{M'} (q, \in, X)$ (since M' include all the moves of M)

$\vdash^{1}_{M'} (q_f, \in, \in)$ (by transition rule 2)

Hence, M' also accepts ω .Conversely, let M' accepts ω .

Then $(q_0', \omega, X) \vdash^{1}_{M'} (q_0, \omega, z_0X) \vdash^{*}_{M'} (q, \in, X) \vdash^{1}_{M'} (q_f, \in, \in)$ for some Q .

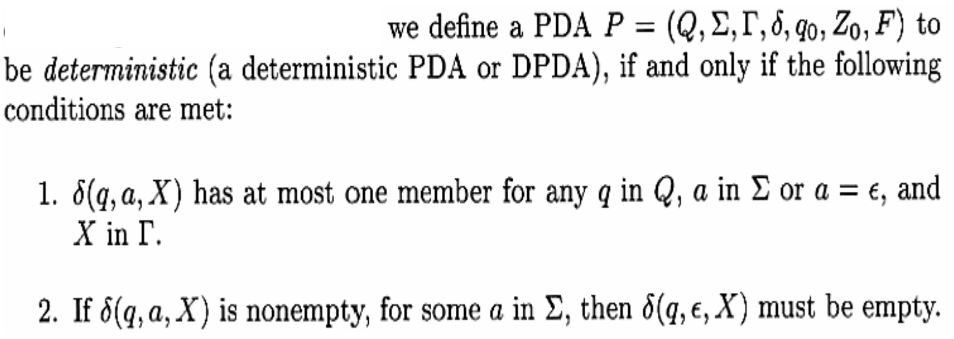
Every move in the sequence

$(q_0, \omega, z_0X) \vdash^{*}_{M'} (q, \in, X)$ were taken from M.

Hence, M starting with its initial configuration will eventually empty its stack and accept the input i.e.

$(q_0, \omega, z_0) \vdash^{*}_{M} (q, \in, \in)$.

Deterministic PDA:

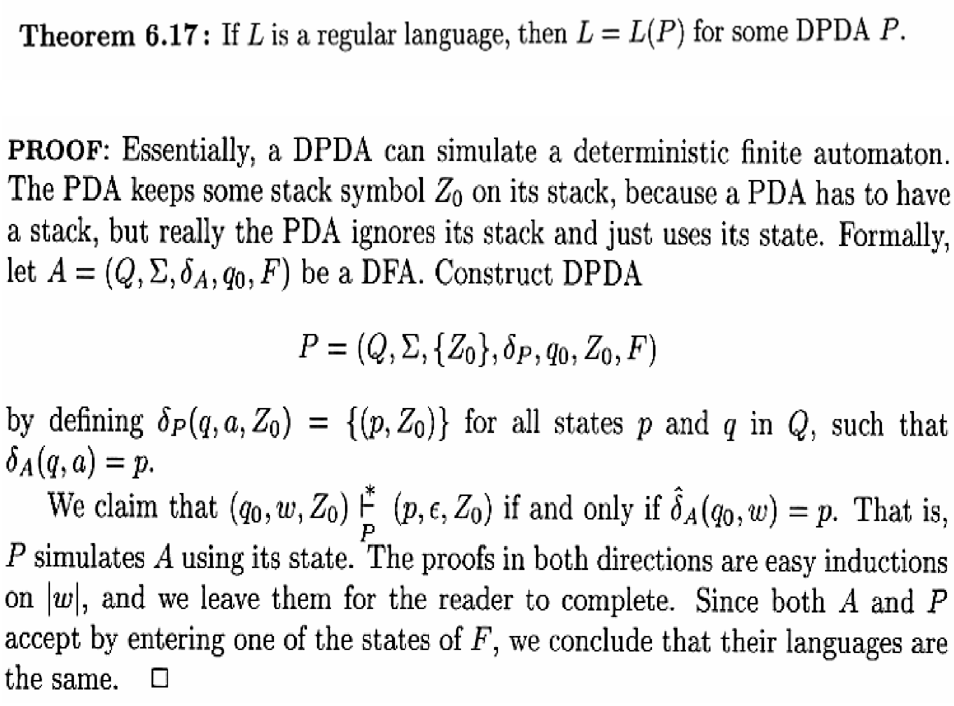


we define a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be *deterministic* (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. $\delta(q, a, X)$ has at most one member for any q in Q, a in Σ or $a = \epsilon$, and X in Γ.

2. If $\delta(q, a, X)$ is nonempty, for some a in Σ, then $\delta(q, \epsilon, X)$ must be empty.

Regular Languages and DPDA’s The DPDA’s accepts a class of languages that is in between the regular languages and CFL’s.



Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P.

PROOF: Essentially, a DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have a stack, but really the PDA ignores its stack and just uses its state. Formally, let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

by defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q, such that $\delta_A(q, a) = p$.

We claim that $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, Z_0)$ if and only if $\hat{\delta}_A(q_0, w) = p$. That is, P simulates A using its state. The proofs in both directions are easy inductions on $|w|$, and we leave them for the reader to complete. Since both A and P accept by entering one of the states of F, we conclude that their languages are the same. □

Deterministic Pushdown Automata (DPDA) and Deterministic Context-free Languages (DCFLs)

Pushdown automata that we have already defined and discussed are nondeterministic by default, that is , there may be two or more moves involving the same combinations of state, input symbol, and top of the stock, and again, for some state and top of the stock the machine may either read and input symbol or make an $\in$- transition (without consuming any input).

In deterministic *PDA* , there is never a choice of move in any situation. This is handled by preventing the above mentioned two cases as described in the definition below.

Defnition : Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a *PDA* . Then *M* is deterministic if and only if both the following conditions are satisfied.

1. $\delta(q, a, X)$ has at most one element for any $q \in Q, a \in \Sigma \cup \{\in\}$, and $X \in \Gamma$ (this condition prevents multiple choice f any combination of q, a and X)
2. If $\delta(q, \in, X) \neq \phi$ and $\delta(q, a, X) = \phi$ for every $a \in \Sigma$

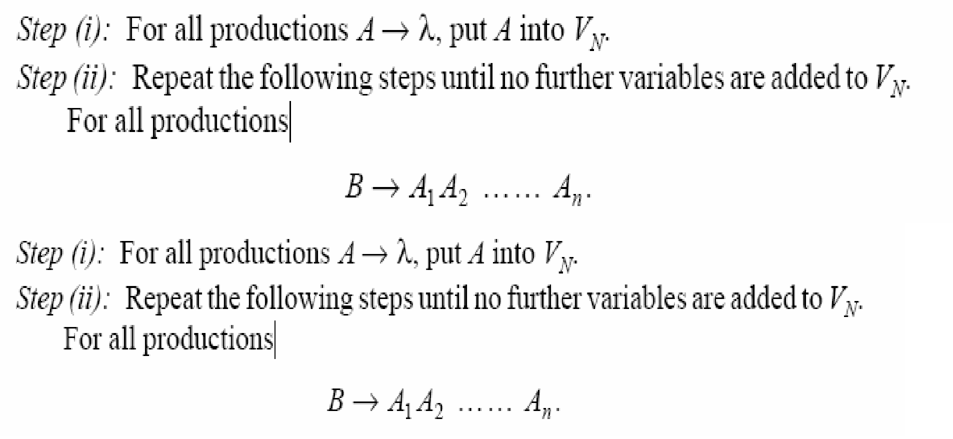
(This condition prevents the possibility of a choice between a move with or without an input symbol).

Empty Production Removal

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammars. If the empty string does not belong to a language, then there is a way to eliminate the productions of the form $A \rightarrow \lambda$ from the grammar.

If the empty string belongs to a language, then we can eliminate λ from all productions save for the single production $S \rightarrow \lambda$. In this case we can also eliminate any occurrences of S from the right-hand side of productions.

Procedure to find CFG with out empty Productions



Step (i): For all productions $A \rightarrow \lambda$, put A into V_N.

Step (ii): Repeat the following steps until no further variables are added to V_N.

For all productions

$$B \rightarrow A_1 A_2 \;\ldots\ldots\; A_n .$$

Step (i): For all productions $A \rightarrow \lambda$, put A into V_N.

Step (ii): Repeat the following steps until no further variables are added to V_N.

For all productions

$$B \rightarrow A_1 A_2 \;\ldots\ldots\; A_n .$$

where $A_1, A_2, A_3, \ldots\ldots, A_n$ are in V_N, put B into V_N.

To find $\hat{P}$, let us consider all productions in P of the form

$$A \rightarrow x_1 x_2 \ \ldots\ldots\ x_m, m \geq 1$$

for each $x_i \in V \cup T$.

Unit production removal

Any production of a CFG of the form

$$A \rightarrow B$$

where $A, B \in V$ is called a "Unit-production". Having variable one on either side of a production is sometimes undesirable.

"Substitution Rule" is made use of in removing the unit-productions.

Given $G = (V, T, S, P)$, a CFG with no λ-productions, there exists a CFG $\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$ that does not have any unit-productions and that is equivalent to G.

Let us illustrate the procedure to remove unit-production through example 2.4.6.

Procedure to remove the unit productions:

Find all variables B, for each A such that

$$A \overset{*}{\Rightarrow} B$$

This is done by sketching a "depending graph" with an edge (C, D) whenever the grammar has unit-production $C \rightarrow D$, then $A \overset{*}{\Rightarrow} B$ holds whenever there is a walk between A and B.

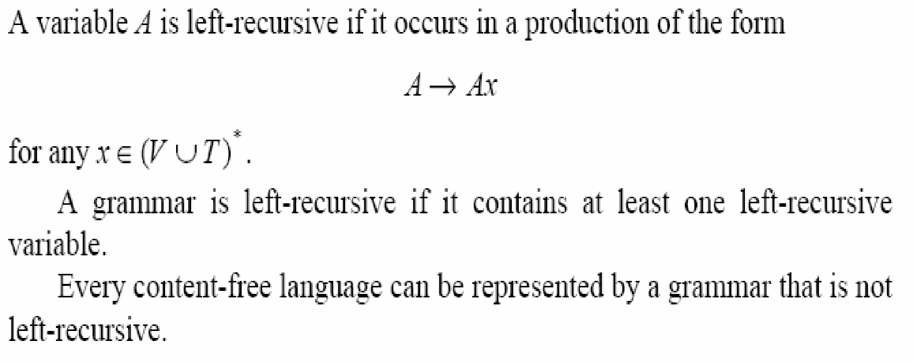
The new grammar $\hat{G}$, equivalent to G is obtained by letting into $\hat{P}$ all non-unit productions of P.

Then for all A and B satisfying $A \overset{*}{\Rightarrow} B$, we add to $\hat{P}$

$$A \rightarrow y_1 \mid y_2 \mid \ldots\ldots \mid y_n$$

where $B \rightarrow y_1 \mid y_2 \mid \ldots\ldots \mid y_n$ is the set of all rules in $\hat{P}$ with B on the left.

Left Recursion Removal



A variable A is left-recursive if it occurs in a production of the form

$$A \to Ax$$

for any $x \in (V \cup T)^*$.

A grammar is left-recursive if it contains at least one left-recursive variable.

Every content-free language can be represented by a grammar that is not left-recursive.

NORMAL FORMS

Two kinds of normal forms viz., Chomsky Normal Form and Greibach Normal Form (GNF) are considered here.

Chomsky Normal Form (CNF)

Any context-free language L without any λ-production is generated by a grammar is which productions are of the form $A \to BC$ or $A \to a$, where A, B $\in V_N$, and a $\in V_T$.

Procedure to find Equivalent Grammar in CNF

(i) Eliminate the unit productions, and λ-productions if any,

(ii) Eliminate the terminals on the right hand side of length two or more.

(iii) Restrict the number of variables on the right hand side of productions to two.

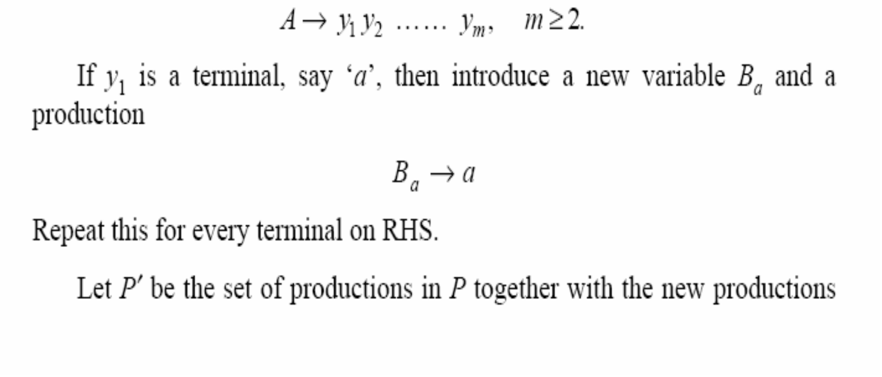
Proof:

For Step (i): Apply the following theorem: “Every context free language can be generated by a grammar with no useless symbols and no unit productions”.

At the end of this step the RHS of any production has a single terminal or two or more symbols.

Let us assume the equivalent resulting grammar as G = (V_N ,V_T ,P ,S).

For Step (ii): Consider any production of the form



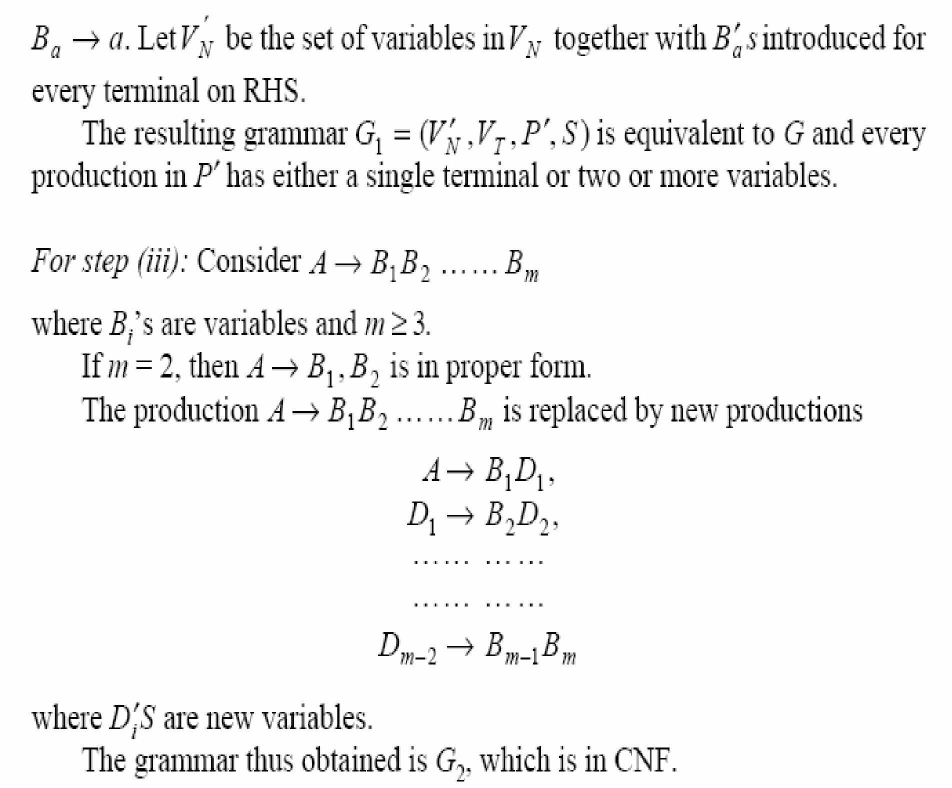
$$A \to y_1 y_2 \ldots\ldots y_m, \quad m \geq 2.$$

If y_1 is a terminal, say ‘a’, then introduce a new variable B_a and a production

$$B_a \to a$$

Repeat this for every terminal on RHS.

Let P' be the set of productions in P together with the new productions



$B_a \to a$. Let $V_N^{'}$ be the set of variables in V_N together with B_a's introduced for every terminal on RHS.

The resulting grammar $G_1 = (V_N', V_T, P', S)$ is equivalent to G and every production in P' has either a single terminal or two or more variables.

For step (iii): Consider $A \to B_1 B_2 \ldots\ldots B_m$

where B_i's are variables and $m \geq 3$.

If $m = 2$, then $A \to B_1, B_2$ is in proper form.

The production $A \to B_1 B_2 \ldots\ldots B_m$ is replaced by new productions

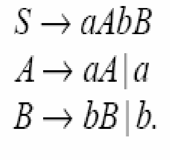
$$\begin{aligned} A &\to B_1 D_1, \\ D_1 &\to B_2 D_2, \\ \ldots\ldots & \;\; \ldots\ldots \\ \ldots\ldots & \;\; \ldots\ldots \\ D_{m-2} &\to B_{m-1} B_m \end{aligned}$$

where $D_i'S$ are new variables.

The grammar thus obtained is G_2, which is in CNF.

Example

Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar *G* with productions *P* given



$$S \to aAbB$$
$$A \to aA \mid a$$
$$B \to bB \mid b.$$

Solution

(i) There are no unit productions in the given set of P.

(ii) Amongst the given productions, we have

$$A \to a,$$
$$B \to b$$

which are in proper form.

For $S \to aAbB$, we have

$$S \to B_a A B_b B,$$
$$B_a \to a$$
$$B_b \to b.$$

For $A \to aA$, we have

$$A \to B_a A$$

For $B \to bB$, we have

$$B \to B_b B.$$

(iii) In P' above, we have only

$$S \to B_a A B_b B$$

not in proper form.

Hence we assume new variables D_1 and D_2 and the productions

$$S \to B_a D_1$$
$$D_1 \to A D_2$$
$$D_2 \to B_b B$$

Therefore the grammar in Chomsky Normal Form (CNF) is G_2 with the productions given by

$$S \to B_a D_1,$$
$$D_1 \to A D_2,$$
$$D_2 \to B_b B,$$
$$A \to B_a A,$$
$$B \to B_b B,$$
$$B_a \to a,$$
$$B_b \to b,$$
$$A \to a,$$

and $$B \to b.$$

Pumping Lemma for CFG

A "Pumping Lemma" is a theorem used to show that, if certain strings belong to a language, then certain other strings must also belong to the language. Let us discuss a Pumping Lemma for CFL. We will show that , if *L* is a context-free language, then strings of *L* that are at least '*m*' symbols long can be "pumped" to produce additional strings in *L*. The value of '*m*' depends on the particular language. Let *L* be an infinite context-free language. Then there is some positive integer '*m*' such that, if *S* is a string of *L* of Length at least '*m*', then

(i) S = uvwxy (for some u, v, w, x, y)

(ii) $| vwx| \leq m$

(iii) $| vx| \geq 1$

(iv) $uv_{i}wx_{i}y \in L$.

for all non-negative values of *i*.

It should be understood that

(i) If *S* is sufficiently long string, then there are two substrings, *v* and *x*, somewhere in *S*. There is stuff (*u*) before *v*, stuff (*w*) between *v* and *x*, and stuff (*y*), after *x*.

(ii) The stuff between *v* and *x* won't be too long, because | *vwx* | can't be larger than *m*.

(iii) Substrings *v* and *x* won't both be empty, though either one could be.

(iv) If we duplicate substring *v*, some number (i) of times, and duplicate *x* the same number of times, the resultant string will also be in *L*.

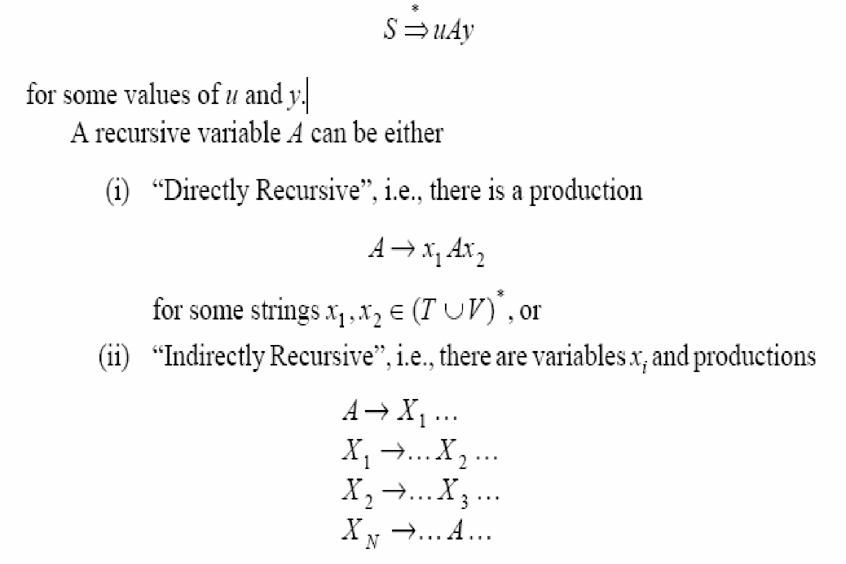
Definitions

A variable is useful if it occurs in the derivation of some string. This requires that

(a) the variable occurs in some sentential form (you can get to the variable if you start from *S*), and

(b) a string of terminals can be derived from the sentential form (the variable is not a "dead end").

A variable is "recursive" if it can generate a string containing itself. For example, variable *A* is recursive if



$$S \overset{*}{\Rightarrow} uAy$$

for some values of *u* and *y*.

A recursive variable *A* can be either

(i) "Directly Recursive", i.e., there is a production

$$A \to x_1 A x_2$$

for some strings $x_1, x_2 \in (T \cup V)^*$, or

(ii) "Indirectly Recursive", i.e., there are variables x_i and productions

$$\begin{aligned} &A \to X_1 \ldots \\ &X_1 \to \ldots X_2 \ldots \\ &X_2 \to \ldots X_3 \ldots \\ &X_N \to \ldots A \ldots \end{aligned}$$

Proof of Pumping Lemma

(a) Suppose we have a CFL given by *L*. Then there is some context-free Grammar *G* that generates *L*. Suppose

(i) *L* is infinite, hence there is no proper upper bound on the length of strings belonging to *L*.

(ii) *L* does not contain l.

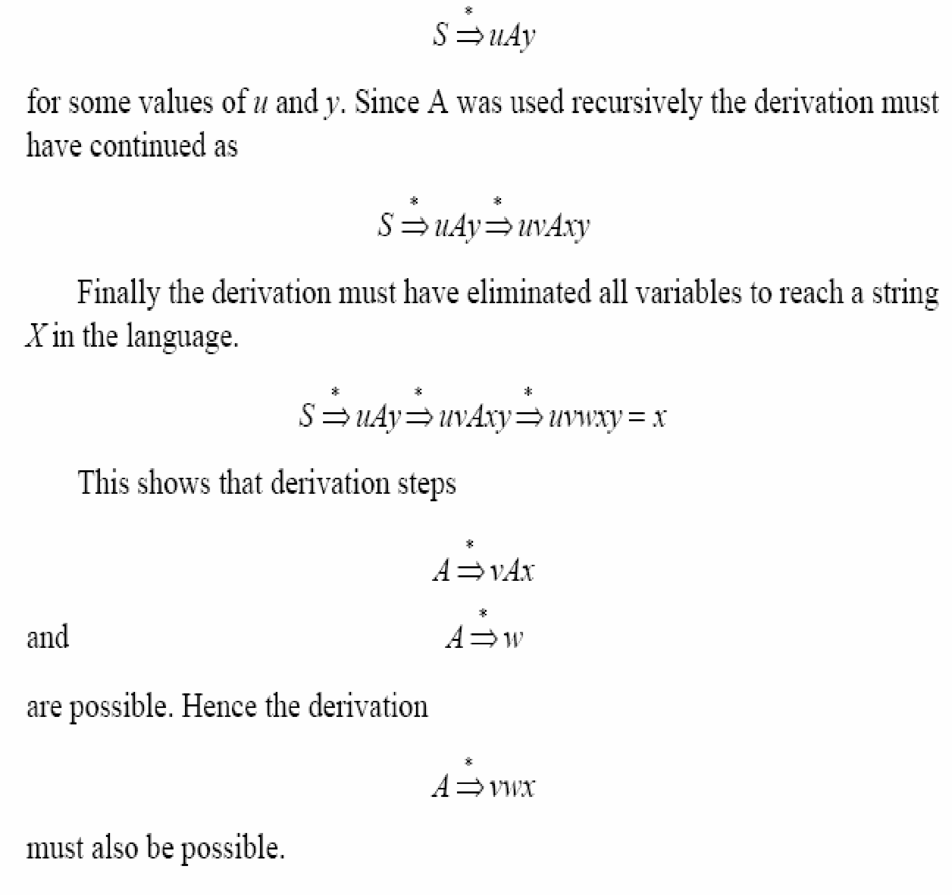
(iii) *G* has no productions or l-productions.

There are only a finite number of variables in a grammar and the productions for each variable have finite lengths. The only way that a grammar can generate arbitrarily long strings is if one or more variables is both useful and recursive. Suppose no variable is recursive. Since the start symbol is non recursive, it must be defined only in terms of terminals and other variables. Then since those variables are non recursive, they have to be defined in terms of terminals and still other variables and so on.

After a while we run out of “other variables” while the generated string is still finite. Therefore there is an upper bond on the length of the string which can be generated from the start symbol. This contradicts our statement that the language is finite.

Hence, our assumption that no variable is recursive must be incorrect.

(b) Let us consider a string *X* belonging to *L*. If *X* is sufficiently long, then the derivation of *X* must have involved recursive use of some variable *A*. Since *A* was used in the derivation, the derivation should have started as



$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y. Since A was used recursively the derivation must have continued as

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

Finally the derivation must have eliminated all variables to reach a string X in the language.

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = x$$

This shows that derivation steps

$$A \overset{*}{\Rightarrow} vAx$$

and $$A \overset{*}{\Rightarrow} w$$

are possible. Hence the derivation

$$A \overset{*}{\Rightarrow} vwx$$

must also be possible.

It should be noted here that the above does not imply that a was used recursively only once. The * of $\overset{*}{\Rightarrow}$ could cover many uses of A, as well as other recursive variables.

There has to be some "last" recursive step. Consider the longest strings that can be derived for v, w and x without the use of recursion. Then there is a number 'm' such that $|vwx| < m$.

Since the grammar does not contain any λ-productions or unit productions, every derivation step either introduces a terminal or increases the length of the sentential form. Since $A \overset{*}{\Rightarrow} vAx$, it follows that $|vx| > 0$.

Finally, since $uvAxy$ occurs in the derivation, and $A \overset{*}{\Rightarrow} vAx$ and $A \overset{*}{\Rightarrow} w$ are both possible, it follows that $uv^i wx^i y$ also belongs to L.

This completes the proof of all parts of Lemma.

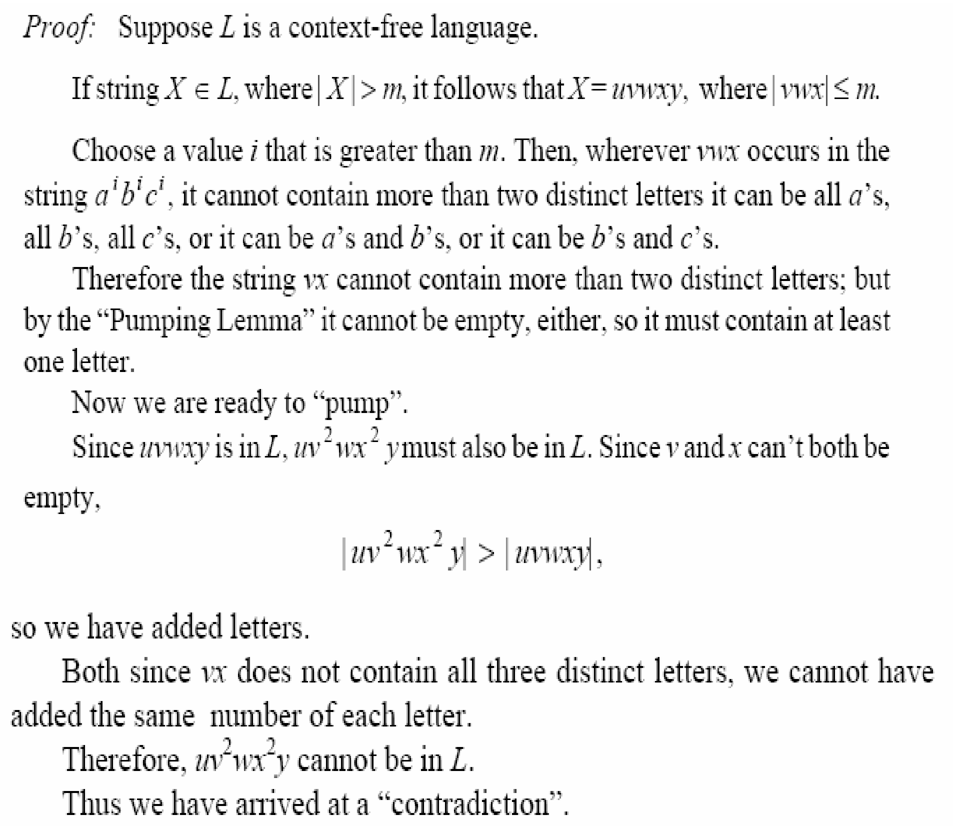
Usage of Pumping Lemma

The Pumping Lemma can be used to show that certain languages are not context free.

Let us show that the language

$$L = \{a^i b^i c^i \mid i > 0\}$$

is not context-free.



Proof: Suppose L is a context-free language.

If string $X \in L$, where $|X| > m$, it follows that $X = uvwxy$, where $|vwx| \le m$.

Choose a value i that is greater than m. Then, wherever vwx occurs in the string $a^i b^i c^i$, it cannot contain more than two distinct letters it can be all a's, all b's, all c's, or it can be a's and b's, or it can be b's and c's.

Therefore the string vx cannot contain more than two distinct letters; but by the "Pumping Lemma" it cannot be empty, either, so it must contain at least one letter.

Now we are ready to "pump".

Since $uvwxy$ is in L, uv^2wx^2y must also be in L. Since v and x can't both be empty,

$$|uv^2wx^2y| > |uvwxy|,$$

so we have added letters.

Both since vx does not contain all three distinct letters, we cannot have added the same number of each letter.

Therefore, uv^2wx^2y cannot be in L.

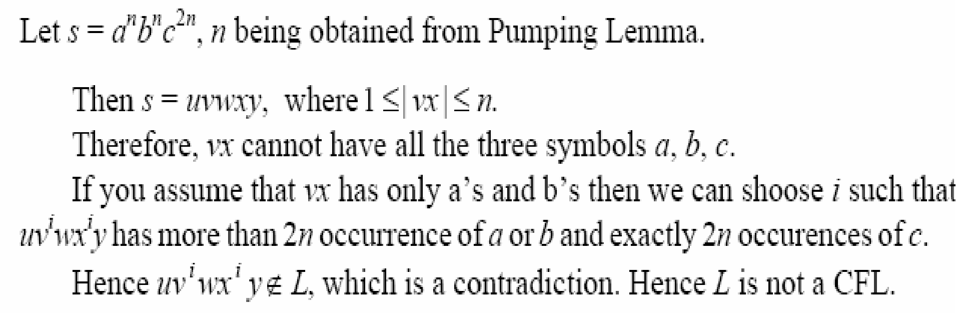
Thus we have arrived at a "contradiction".

Hence our original assumption, that L is context free should be false. Hence the language L is not con text-free.

Example

Check whether the language given by $L = \{a^{m}b^{m}C^{n} : m \le n \le 2m\}$ is a CFL or not.

Solution



Let $s = a^n b^n c^{2n}$, n being obtained from Pumping Lemma.

Then $s = uvwxy$, where $1 \le |vx| \le n.$

Therefore, vx cannot have all the three symbols a, b, c.

If you assume that vx has only a's and b's then we can shoose i such that uv^iwx^iy has more than $2n$ occurrence of a or b and exactly $2n$ occurences of c.

Hence $uv^iwx^iy \notin L$, which is a contradiction. Hence L is not a CFL.

Closure properties of CFL – Substitution

Let Σ be an alphabet, and suppose that for every symbol a in Σ, we choose a language L_a. These chosen languages can be over any alphabets, not necessarily Σ and not necessarily the same. This choice of languages defines a function s (a *substitution*) on Σ, and we shall refer to L_a as $s(a)$ for each symbol a.

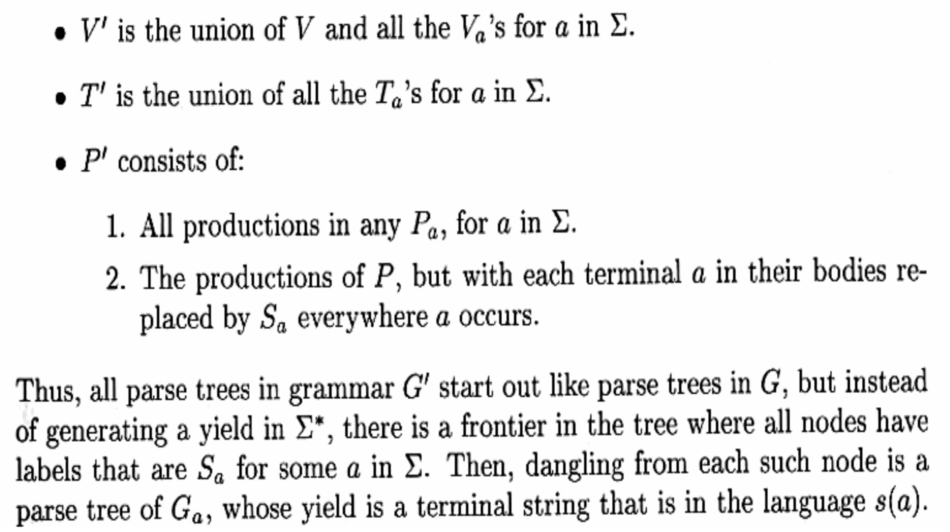
If $w = a_1a_2 \cdots a_n$ is a string in Σ^*, then $s(w)$ is the language of all strings $x_1x_2 \cdots x_n$ such that string x_i is in the language $s(a_i)$, for $i = 1, 2, \ldots, n$. Put another way, $s(w)$ is the concatenation of the languages $s(a_1)s(a_2) \cdots s(a_n)$. We can further extend the definition of s to apply to languages: $s(L)$ is the union of $s(w)$ for all strings w in L.

Theorem 7.23: If L is a context-free language over alphabet Σ, and s is a substitution on Σ such that $s(a)$ is a CFL for each a in Σ, then $s(L)$ is a CFL.

PROOF: The essential idea is that we may take a CFG for L and replace each terminal a by the start symbol of a CFG for language $s(a)$. The result is a single CFG that generates $s(L)$. However, there are a few details that must be gotten right to make this idea work.

More formally, start with grammars for each of the relevant languages, say $G = (V, \Sigma, P, S)$ for L and $G_a = (V_a, T_a, P_a, S_a)$ for each a in Σ. Since we can choose any names we wish for variables, let us make sure that the sets of variables are disjoint; that is, there is no symbol A that is in two or more of V and any of the V_a's. The purpose of this choice of names is to make sure that when we combine the productions of the various grammars into one set of productions, we cannot get accidental mixing of the productions from two grammars and thus have derivations that do not resemble the derivations in any of the given grammars.

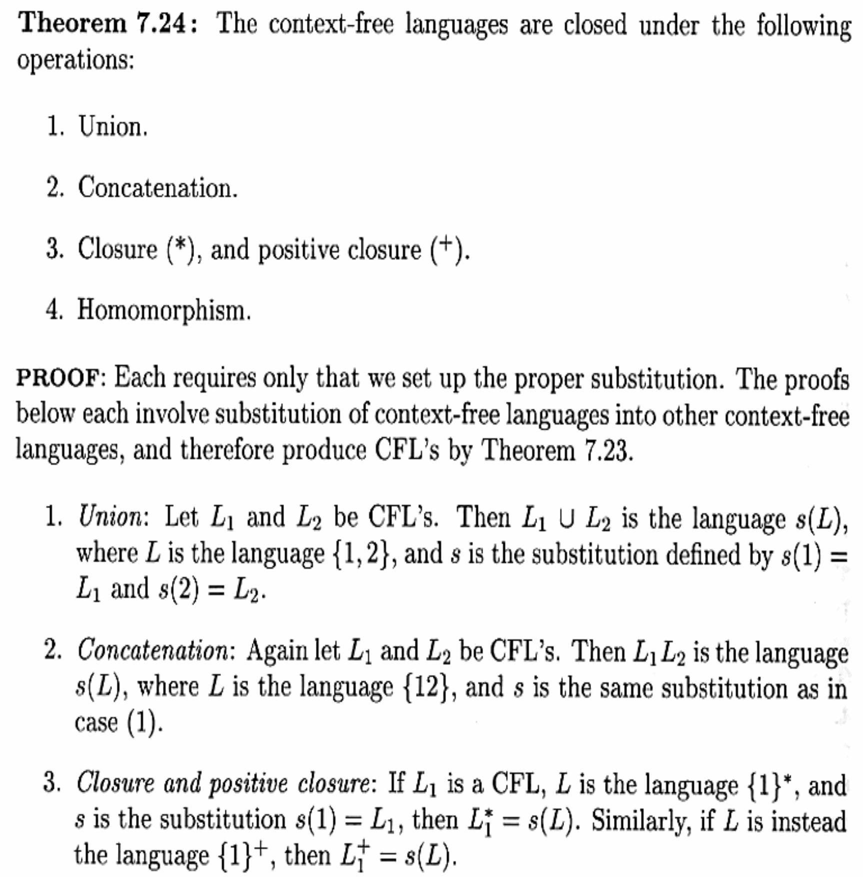
We construct a new grammar $G' = (V', T', P', S)$ for $s(L)$, as follows:



- V' is the union of V and all the V_a's for a in Σ.
- T' is the union of all the T_a's for a in Σ.
- P' consists of:
  1. All productions in any P_a, for a in Σ.
  2. The productions of P, but with each terminal a in their bodies replaced by S_a everywhere a occurs.

Thus, all parse trees in grammar G' start out like parse trees in G, but instead of generating a yield in Σ^*, there is a frontier in the tree where all nodes have labels that are S_a for some a in Σ. Then, dangling from each such node is a parse tree of G_a, whose yield is a terminal string that is in the language $s(a)$.

Applications of substitution theorem



Theorem 7.24: The context-free languages are closed under the following operations:

1. Union.

2. Concatenation.

3. Closure (*), and positive closure ($^+$).

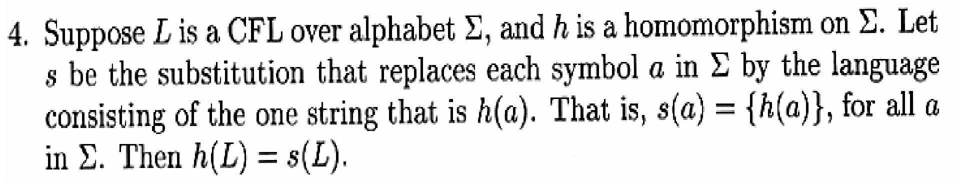
4. Homomorphism.

PROOF: Each requires only that we set up the proper substitution. The proofs below each involve substitution of context-free languages into other context-free languages, and therefore produce CFL's by Theorem 7.23.

1. *Union*: Let L_1 and L_2 be CFL's. Then $L_1 \cup L_2$ is the language $s(L)$, where L is the language $\{1,2\}$, and s is the substitution defined by $s(1) = L_1$ and $s(2) = L_2$.

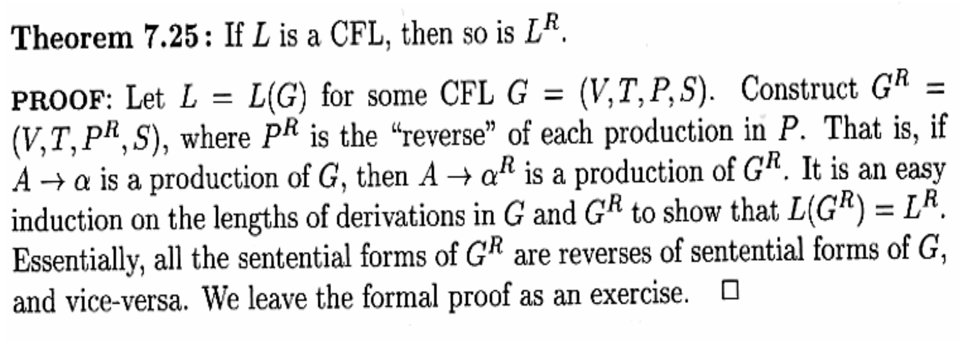
2. *Concatenation*: Again let L_1 and L_2 be CFL's. Then L_1L_2 is the language $s(L)$, where L is the language $\{12\}$, and s is the same substitution as in case (1).

3. *Closure and positive closure*: If L_1 is a CFL, L is the language $\{1\}^*$, and s is the substitution $s(1) = L_1$, then $L_1^* = s(L)$. Similarly, if L is instead the language $\{1\}^+$, then $L_1^+ = s(L)$.



4. Suppose L is a CFL over alphabet Σ, and h is a homomorphism on Σ. Let s be the substitution that replaces each symbol a in Σ by the language consisting of the one string that is $h(a)$. That is, $s(a) = \{h(a)\}$, for all a in Σ. Then $h(L) = s(L)$.

Reversal



Theorem 7.25: If L is a CFL, then so is L^R.

PROOF: Let $L = L(G)$ for some CFL $G = (V,T,P,S)$. Construct $G^R = (V,T,P^R,S)$, where P^R is the "reverse" of each production in P. That is, if $A \to \alpha$ is a production of G, then $A \to \alpha^R$ is a production of G^R. It is an easy induction on the lengths of derivations in G and G^R to show that $L(G^R) = L^R$. Essentially, all the sentential forms of G^R are reverses of sentential forms of G, and vice-versa. We leave the formal proof as an exercise. □

Inverse Homomorphism:

Theorem 7.30: Let L be a CFL and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

PROOF: Suppose h applies to symbols of alphabet Σ and produces strings in T^*. We also assume that L is a language over alphabet T. As suggested above, we start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ that accepts L by final state. We construct a new PDA

$$P' = (Q', \Sigma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \tag{7.1}$$

where:

1. Q' is the set of pairs (q, x) such that:

   (a) q is a state in Q, and

   (b) x is a suffix (not necessarily proper) of some string $h(a)$ for some input symbol a in Σ.

   That is, the first component of the state of P' is the state of P, and the second component is the buffer. We assume that the buffer will periodically be loaded with a string $h(a)$, and then allowed to shrink from the front, as we use its symbols to feed the simulated PDA P. Note that since Σ is finite, and $h(a)$ is finite for all a, there are only a finite number of states for P'.

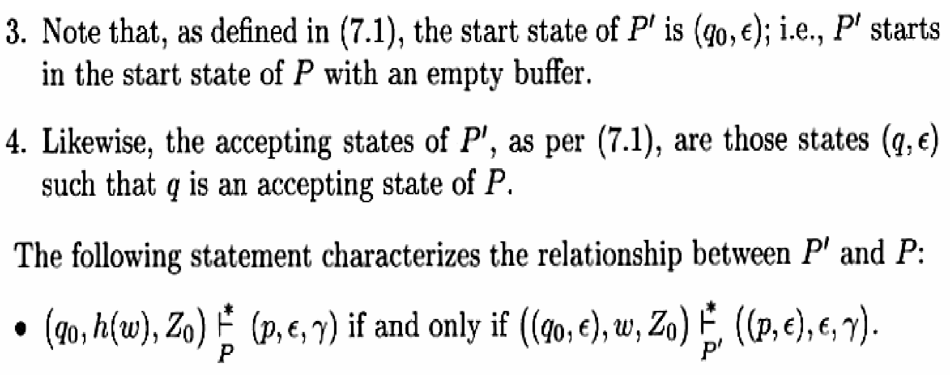
2. δ' is defined by the following rules:

   (a) $\delta'\big((q,\epsilon), a, X\big) = \Big\{\Big(\big(q, h(a)\big), X\Big)\Big\}$ for all symbols a in Σ, all states q in Q, and stack symbols X in Γ. Note that a cannot be ϵ here. When the buffer is empty, P' can consume its next input symbol a and place $h(a)$ in the buffer.

   (b) If $\delta(q, b, X)$ contains (p, γ), where b is in T or $b = \epsilon$, then

   $$\delta'\big((q, bx), \epsilon, X\big)$$

   contains $\big((p, x), \gamma\big)$. That is, P' always has the option of simulating a move of P, using the front of its buffer. If b is a symbol in T, then the buffer must not be empty, but if $b = \epsilon$, then the buffer can be empty.



3. Note that, as defined in (7.1), the start state of P' is (q_0, ϵ); i.e., P' starts in the start state of P with an empty buffer.

4. Likewise, the accepting states of P', as per (7.1), are those states (q, ϵ) such that q is an accepting state of P.

The following statement characterizes the relationship between P' and P:

- $(q_0, h(w), Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, \gamma)$ if and only if $((q_0, \epsilon), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((p, \epsilon), \epsilon, \gamma)$.

MODULE-IV

Turing machine:

Informal Definition:

We consider here a basic model of TM which is deterministic and have one-tape. There are many variations, all are equally powerfull.

The basic model of TM has a finite set of states, a semi-infinite tape that has a leftmost cell but is infinite to the right and a tape head that can move left and right over the tape, reading and writing symbols.

For any input w with |w|=n, initially it is written on the n leftmost (continguous) tape cells. The infinitely many cells to the right of the input all contain a blank symbol, B whcih is a special tape symbol that is not an input symbol. The machine starts in its start state with its head scanning the leftmost symbol of the input w. Depending upon the symbol scanned by the tape head and the current state the machine makes a move which consists of the following:

- writes a new symbol on that tape cell, • moves its head one cell either to the left or to the right and
- (possibly) enters a new state.

The action it takes in each step is determined by a transition functions. The machine continues computing (i.e. making moves) until

- it decides to "accept" its input by entering a special state called accept or final state or
- halts without accepting i.e. rejecting the input when there is no move defined.

On some inputs the TM many keep on computing forever without ever accepting or rejecting the input, in which case it is said to "loop" on that input

Formal Definition :

Formally, a deterministic turing machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite nonempty set of states.
- Γ is a finite non-empty set of tape symbols, callled the tape alphabet of M.
- $\Sigma \subseteq \Gamma$ is a finite non-empty set of input symbols, called the input alphabet of M.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L \times R\}$ is the transition function of M,

- $q_0 \in Q$ is the initial or start state.
- $B \in \Gamma \setminus \Sigma$ is the blank symbol
- $F \subseteq Q$ is the set of final state.

So, given the current state and tape symbol being read, the transition function describes the next state, symbol to be written on the tape, and the direction in which to move the tape head (L and R denote left and right, respectively).

Transition function : δ

- The heart of the TM is the transition function, δ because it tells us how the machine gets one step to the next.
- when the machine is in a certain state q$\in$Q and the head is currently scanning the tape symbol $X \in \Gamma$, and if $\delta(q,x) = (p,Y,D)$, then the machine

1. replaces the symbol X by Y on the tape
2. goes to state p, and
3. the tape head moves one cell (i.e. one tape symbol) to the left (or right) if D is L (or R).

The ID (instantaneous description) of a TM capture what is going out at any moment i.e. it contains all the information to exactly capture the "current state of the computations".

It contains the following:

- The current state, q
- The position of the tape head,
- The constants of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost.

Note that, although there is no limit on how far right the head may move and write nonblank symbols on the tape, at any finite

time, the TM has visited only a finite prefix of the infinite tape.

An ID (or configuration) of a TM M is denoted by $\alpha q \beta$ where $\alpha, \beta \in \Gamma^*$ and

- α is the tape contents to the left of the head
- q is the current state.
- β is the tape contents at or to the right of the tape head

That is, the tape head is currently scanning the leftmost tape symbol of β. (Note that if $\beta = \in$, then the tape head is scanning a blank symbol)

If q_0 is the start state and w is the input to a TM M then the starting or initial configuration of M is onviously denoted by $q_0 w$

Moves of Turing Machines

To indicate one move we use the symbol $\vdash$. Similarly, zero, one, or more moves will be represented by $\vdash$. A move of a TM

M is defined as follows.

Let $\alpha ZqX\beta$ be an ID of M where $X, Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$ and $q \in Q$.

Let there exists a transition $\delta(q, X) = (p, Y, L)$ of M.

Then we write $\alpha ZqX\beta \vdash_M \alpha qZY\beta$ meaning that ID $\alpha ZqX\beta$ yields $\alpha ZqY\beta$

- Alternatively , if $\delta(q, X) = (p, Y, R)$ is a transition of M, then we write $\alpha ZqX\beta \vdash \alpha zYp\beta$ which means that the ID $\alpha zqX\beta$ yields $\alpha zYp\beta$

- In other words, when two IDs are related by the relation $\vdash$, we say that the first one yields the second (or the second is the result of the first) by one move.
- If IDj results from IDi by zero, one or more (finite) moves then we write $\vdash$(If the TM M is understand, then the subscript M can be dropped from $\vdash$ or $\vdash$)

Special Boundary Cases

- Let $qx\alpha$ be an ID and $\delta(q, x) = (p, Y, L)$ be an transition of M. Then $\vdash$. That is, the head is not allowed to fall off the left end of the tape.
- Let $qx\alpha$ be an ID and $\delta(q, x) = (p, Y, R)$ then figure (Note that αYq is equivalent to αYqB)
- Let $qx\alpha$ be an ID and $\delta(q, x) = (p, B, R)$ then figure
- Let αzqx be an ID and $\delta(q, x) = (p, B, L)$ then figure

The language accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, denoted as $L(M)$ is

$L(M)$ = { w | $w \in \Sigma^*$ and figure for some $p \in F$ and $\alpha, \beta \in \Gamma^*$ }

In other words the TM M accepts a string $w \in \Sigma^*$ that cause M to enter a final or accepting state when started in its initial ID (i.e. $q_0 w$). That is a TM M accepts the string $w \in \Sigma^*$ if a sequence of IDs, $ID_1, ID_2, \cdots, ID_k$ exists such that

- ID_1 is the initial or starting ID of M
- $ID_i \vdash_M ID_{i+1}$; $1 \leq i < k$

- The representation of IDk contains an accepting state.

The set of strings that *M* accepts is the language of *M*, denoted *L*(*M*), as defined above

More about configuration and acceptance

- An ID $\alpha q \beta$ of *M* is called an accepting (or final) ID if $q \in F$
- An ID $\alpha q x \beta$ is called a blocking (or halting) ID if $\delta(q, x)$ is undefined i.e. the TM has no move at this point.
- ID_j is called reactable from ID_i if $ID_i \vdash_M^* ID_j$
- $q_0 w$ is the initial (or starting) ID if $w \in \Sigma^*$ is the input to the TM and $q_0 \in Q$ is the initial (or start) state of *M*.

On any input string $w \in \Sigma^*$

either

- *M* halts on *w* if there exists a blocking (configuration) ID, I' such that $q_0 w \vdash_M^* I'$

There are two cases to be considered

- *M* accepts *w* if *I* is an accepting ID. The set of all $w \in \Sigma^*$ accepted by *M* is denoted as *L*(*M*) as already defined
- *M* rejects *w* if I' is a blocking configuration. Denote by reject (*M*), the set of all $w \in \Sigma^*$ rejected by *M*.

or

- *M* loops on *w* if it does not halt on *w*.

Let *loop*(*M*) be the set of all $w \in \Sigma^*$ on which *M* loops for.

It is quite clear that

$$L(M) \cup reject(M) \cup loop(M) = \Sigma^*$$

That is, we assume that a TM *M* halts

- When it enters an accepting ID_1 or
- When it enters a blocking ID_1 i.e. when there is no next move.

However, on some input string, , $w \notin L(M)$, it is possible that the TM *M* loops for ever i.e. it never halts

The Halting Problem

The input to a Turing machine is a string. Turing machines themselves can be written as strings. Since these strings can be used as input to other Turing machines. A "Universal Turing machine" is one whose input consists of a description *M* of some arbitrary Turing machine, and some input w to which machine *M* is to be applied, we write this combined input as *M* + *w*. This produces the same output that would be produced by M. This is written as

Universal Turing Machine (*M* + *w*) = *M* (*w*).

As a Turing machine can be represented as a string, it is fully possible to supply a Turing machine as input to itself, for example *M* (*M*). This is not even a particularly bizarre thing to do for example, suppose you have written a C pretty printer in C, then used the Pretty printer on itself. Another common usage is Bootstrapping—where some convenient languages used to write a minimal compiler for some new language L, then used this minimal compiler for L to write a new, improved compiler for language L. Each time a new feature is added to language L, you can recompile and use this new feature in the next version of the compiler. Turing machines sometimes halt, and sometimes they enter an infinite loop.

A Turing machine might halt for one input string, but go into an infinite loop when given some other string. The halting problem asks: "It is possible to tell, in general, whether a given machine will halt for some given input?" If it is possible, then there is an effective procedure to look at a Turing machine and its input and determine whether the machine will halt with that input. If there is an effective procedure, then we can build a Turing machine to implement it. Suppose we have a Turing machine "WillHalt" which, given an input string *M* + *w*, will halt and accept the string if Turing machine *M* halts on input *w* and will halt and reject the string if Turing machine *M* does not halt on input *w*. When viewed as a Boolean function, "WillHalt (*M*, *w*)" halts and returns "TRUE" in the first case, and (halts and) returns "FALSE" in the second.

Theorem

Turing Machine "WillHalt (*M*, *w*)" does not exist.

Proof: This theorem is proved by contradiction. Suppose we could build a machine "WillHalt". Then we can certainly build a second machine, "LoopIfHalts", that will go into an infinite loop if and only if "WillHalt" accepts its input:

```
Function LoopIfHalts (M, w):
if WillHalt (M, w) then
while true do { }
else
return false;
```

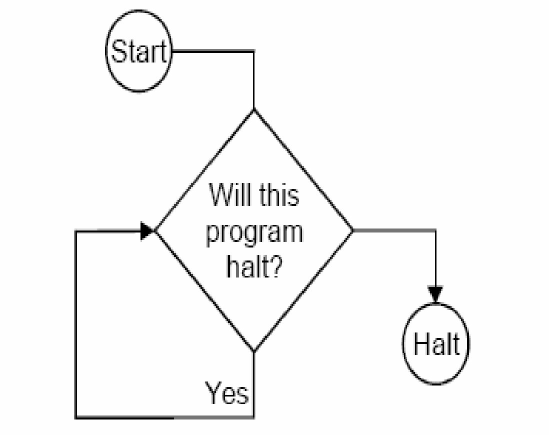
We will also define a machine "LoopIfHaltOnItSelf" that, for any given input *M*, representing a Turing machine, will determine what will happen if *M* is applied to itself, and loops if *M* will halt in this case.

```
Function LoopIfHaltsOnItself (M):
return LoopIfHalts (M, M):
```

Finally, we ask what happens if we try:

```
Func tion Impos sible:
return LoopIfHaltsOnItself (LoopIfHaltsOnItself):
```

This machine, when applied to itself, goes into an infinite loop if and only if it halts when applied to itself. This is impossible. Hence the theorem is proved.



Implications of Halting Problem

Programming

The Theorem of “Halting Problem” does not say that we can never determine whether or not a given program halts on a given input. Most of the times, for practical reasons, we could eliminate infinite loops from programs. Sometimes a “meta-program” is used to check another program for potential infinite loops, and get this meta-program to work most of the time.

The theorem says that we cannot ever write such a meta-program and have it work all of the time. This result is also used to demonstrate that certain other programs are also impossible.

The basic outline is as follows:

(i) If we could solve a problem *X*, we could solve the Halting problem

(ii) We cannot solve the Halting Problem

(iii) Therefore, we cannot solve problem *X*

A Turing machine can be "programmed," in much the same manner as a computer is programmed. When one specifies the function which we usually call δ for a Tm, he is really writing a program for the Tm.

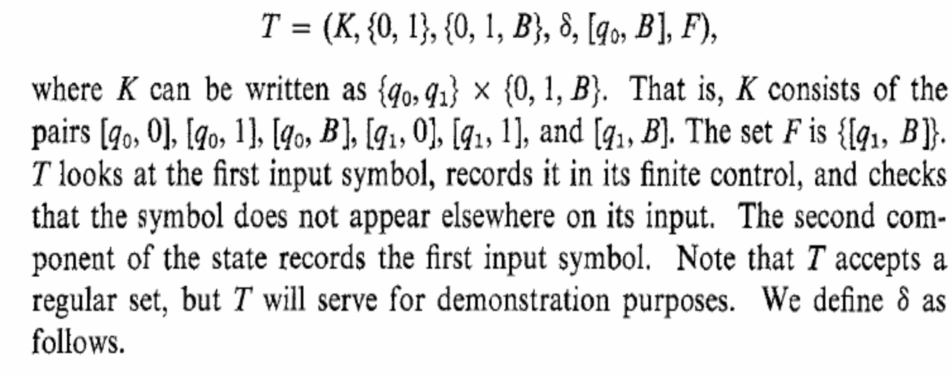
1. Storage in finite Control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing a symbol. It should be emphasized that this arrangement is for conceptual purposes only. No modification in the definition of the Turing machine has been made.

Example

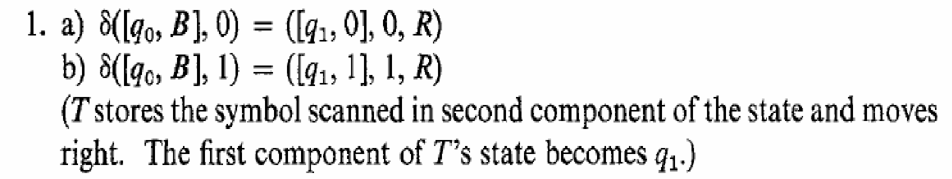
Consider the Turing machine

Solution

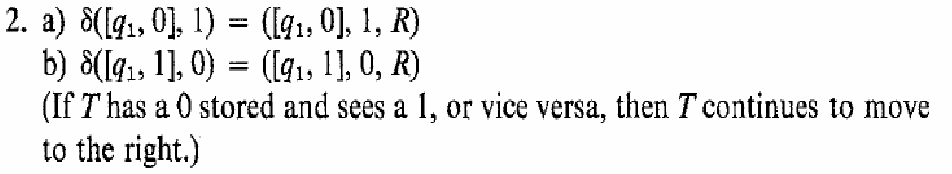


$$T = (K, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], F),$$

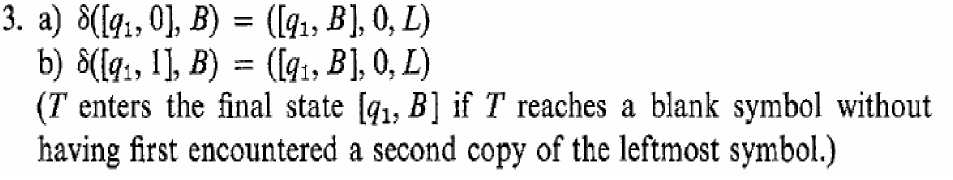
where K can be written as $\{q_0, q_1\} \times \{0, 1, B\}$. That is, K consists of the pairs $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$, and $[q_1, B]$. The set F is $\{[q_1, B]\}$. T looks at the first input symbol, records it in its finite control, and checks that the symbol does not appear elsewhere on its input. The second component of the state records the first input symbol. Note that T accepts a regular set, but T will serve for demonstration purposes. We define δ as follows.



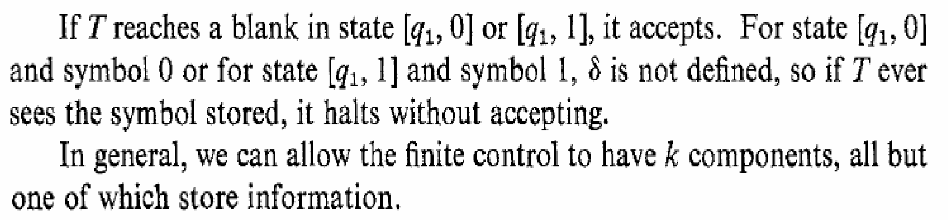
1. a) $\delta([q_0, B], 0) = ([q_1, 0], 0, R)$
   b) $\delta([q_0, B], 1) = ([q_1, 1], 1, R)$
   (T stores the symbol scanned in second component of the state and moves right. The first component of T's state becomes q_1.)



2. a) $\delta([q_1, 0], 1) = ([q_1, 0], 1, R)$
   b) $\delta([q_1, 1], 0) = ([q_1, 1], 0, R)$
   (If T has a 0 stored and sees a 1, or vice versa, then T continues to move to the right.)



3. a) $\delta([q_1, 0], B) = ([q_1, B], 0, L)$
   b) $\delta([q_1, 1], B) = ([q_1, B], 0, L)$
   (T enters the final state $[q_1, B]$ if T reaches a blank symbol without having first encountered a second copy of the leftmost symbol.)



If T reaches a blank in state $[q_1, 0]$ or $[q_1, 1]$, it accepts. For state $[q_1, 0]$ and symbol 0 or for state $[q_1, 1]$ and symbol 1, δ is not defined, so if T ever sees the symbol stored, it halts without accepting.

In general, we can allow the finite control to have k components, all but one of which store information.

2. Multiple Tracks

We can imagine that the tape of the Turing machine is divided into k tracks, for any finite k. This arrangement is shown in Fig., with k = 3. What is actually done is that the symbols on the tape are considered as k-tuples. One component for each track.

Example

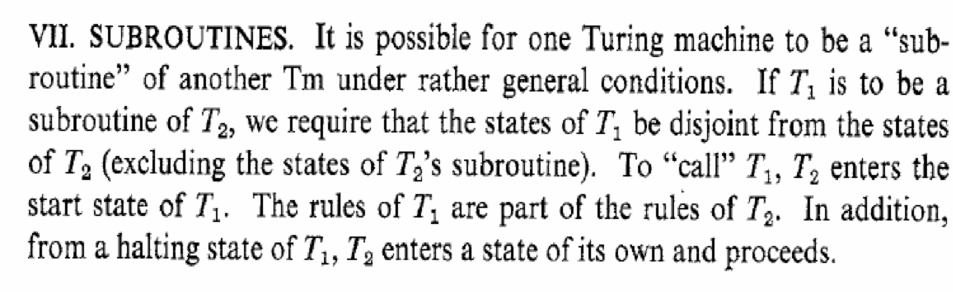
The tape in Fig. can be imagined to be that of a Turing machine which takes a binary input greater than 2, written on the first track, and determines if it is a prime. The input is surrounded by ¢ and $ on the first track.

Thus, the allowable input symbols are [¢, B, B], [0, B, B], [1, B, B], and [$, B, B]. These symbols can be identified with ¢, 0, 1, and $, respectively, when viewed as input symbols. The blank symbol can be represented by [B, B, B]

To test if its input is a prime, the Tm first writes the number two in binary on the second track and copies the first track onto the third track. Then, the second track is subtracted, as many times as possible, from the third track, effectively dividing the third track by the second and leaving the remainder. If the remainder is zero, the number on the first track is not a prime. If the remainder is nonzero, increase the number on the second track by one.

If now the second track equals the first, the number on the first track is a prime, because it cannot be divided by any number between one and itself. If the second is less than the first, the whole operation is repeated for the new number on the second track. In Fig., the Tm is testing to determine if 47 is a prime. The Tm is dividing by 5; already 5 has been subtracted twice, so 37 appears on the third track.

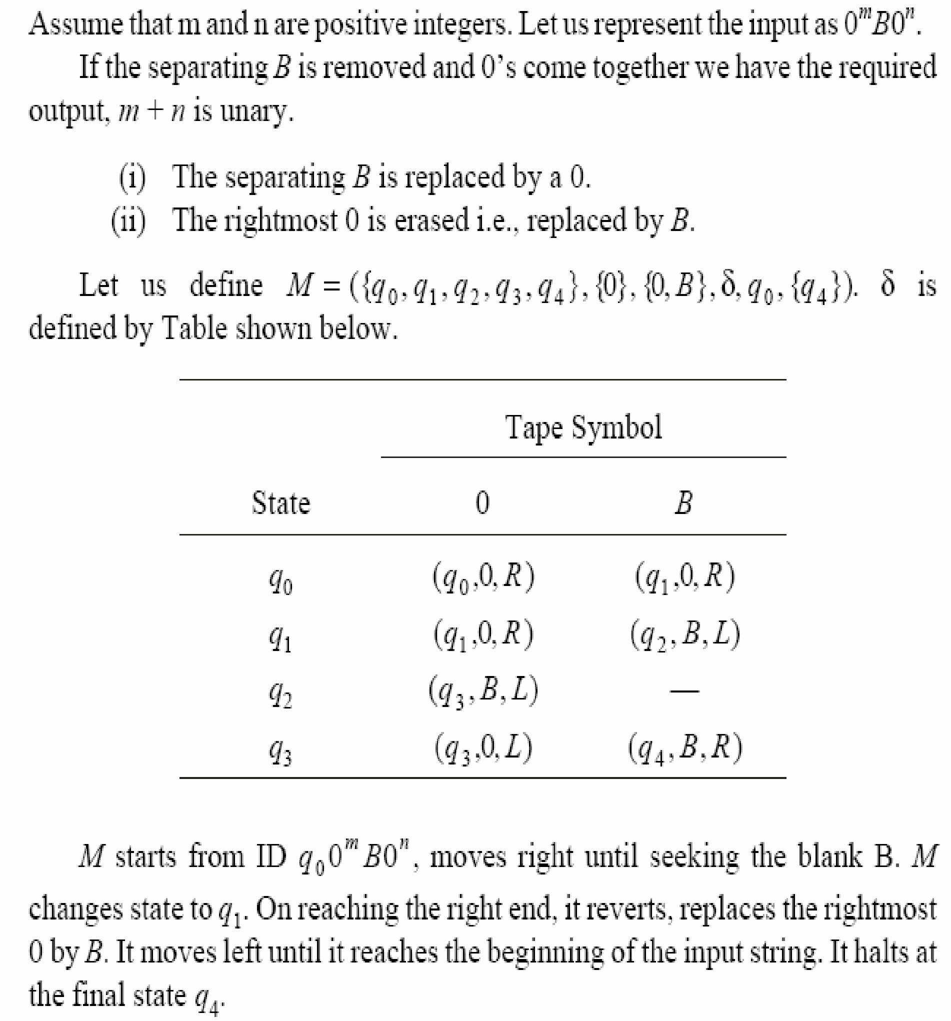
3. Subroutines



VII. SUBROUTINES. It is possible for one Turing machine to be a "subroutine" of another Tm under rather general conditions. If T_1 is to be a subroutine of T_2, we require that the states of T_1 be disjoint from the states of T_2 (excluding the states of T_2's subroutine). To "call" T_1, T_2 enters the start state of T_1. The rules of T_1 are part of the rules of T_2. In addition, from a halting state of T_1, T_2 enters a state of its own and proceeds.

UNDECIDABILITY

Design a Turing machine to add two given integers.
Solution:



Assume that m and n are positive integers. Let us represent the input as $0^m B 0^n$.

If the separating B is removed and 0's come together we have the required output, $m + n$ is unary.

(i) The separating B is replaced by a 0.
(ii) The rightmost 0 is erased i.e., replaced by B.

Let us define $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0\}, \{0, B\}, \delta, q_0, \{q_4\})$. δ is defined by Table shown below.

	Tape Symbol	
State	0	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$
q_1	$(q_1, 0, R)$	(q_2, B, L)
q_2	(q_3, B, L)	—
q_3	$(q_3, 0, L)$	(q_4, B, R)

M starts from ID $q_0 0^m B 0^n$, moves right until seeking the blank B. M changes state to q_1. On reaching the right end, it reverts, replaces the rightmost 0 by B. It moves left until it reaches the beginning of the input string. It halts at the final state q_4.

Some unsolvable Problems are as follows:
(i) Does a given Turing machine *M* halts on all input?
(ii) Does Turing machine *M* halt for any input?
(iii) Is the language *L*(*M*) finite?
(iv) Does *L*(*M*) contain a string of length *k*, for some given *k*?
(v) Do two Turing machines *M*1 and *M*2 accept the same language?
It is very obvious that if there is no algorithm that decides, for an arbitrary given Turing machine *M* and input string *w*, whether or not *M* accepts *w*. These problems for which no algorithms exist are called "UNDECIDABLE" or "UNSOLVABLE".

Code for Turing Machine:

Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet $\{0,1\}$ may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing machine, M_i." To represent a TM $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.

- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.

- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM M such that $L(M) = L_d$.

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function δ. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

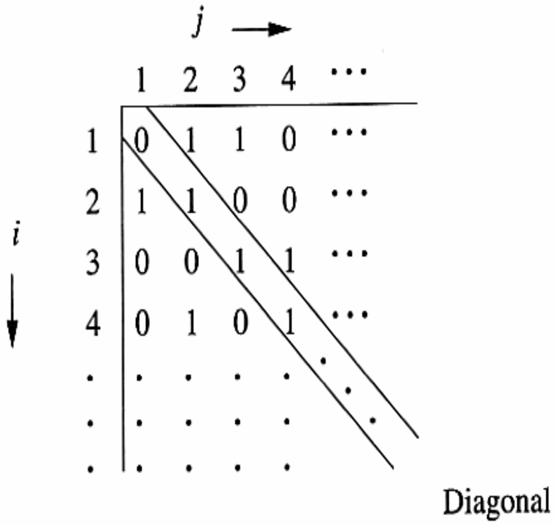
where each of the C's is the code for one transition of M.

Diagonalization language:

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not in $L(M_i)$.

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

The reason L_d is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the ith row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.



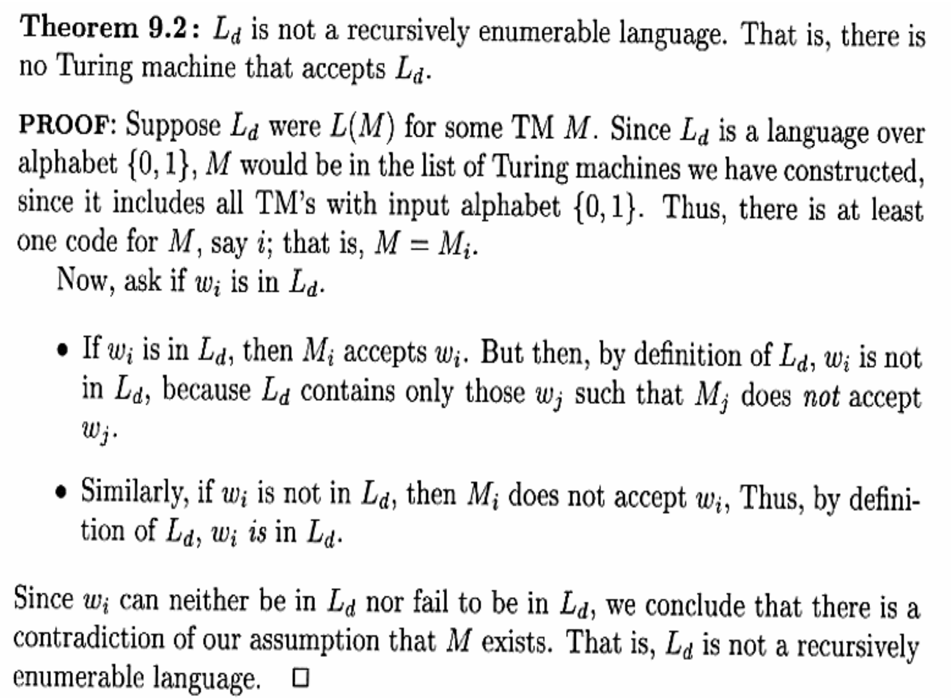
	1	2	3	4	...
1	0	1	1	0	...
2	1	1	0	0	...
3	0	0	1	1	...
4	0	1	0	1	...
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	

This table represents language acceptable by Turing machine

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, L_d would contain $w_1 = \epsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is

Proof that L_d is not recursively enumerable:



Theorem 9.2: L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

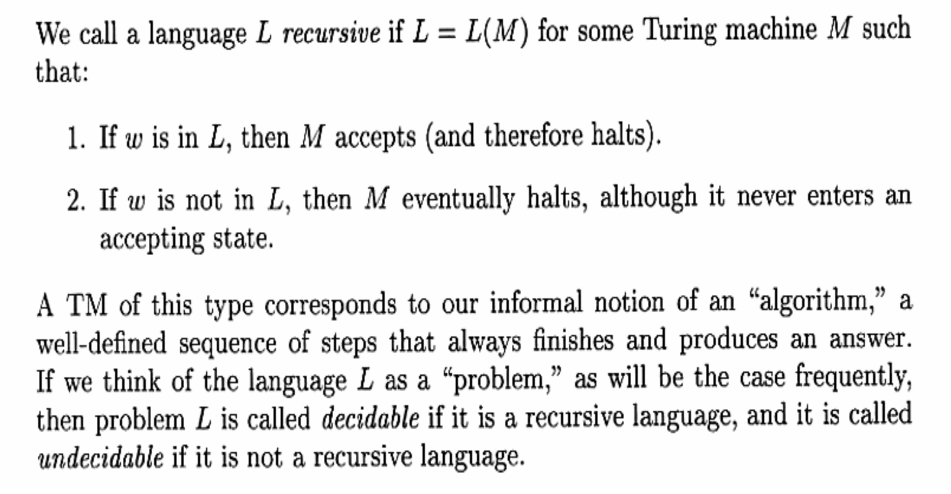
PROOF: Suppose L_d were $L(M)$ for some TM M. Since L_d is a language over alphabet $\{0, 1\}$, M would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0, 1\}$. Thus, there is at least one code for M, say i; that is, $M = M_i$.

Now, ask if w_i is in L_d.

- If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_j such that M_j does *not* accept w_j.
- Similarly, if w_i is not in L_d, then M_i does not accept w_i, Thus, by definition of L_d, w_i *is* in L_d.

Since w_i can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language. □

Recursive Languages:



We call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," as will be the case frequently, then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

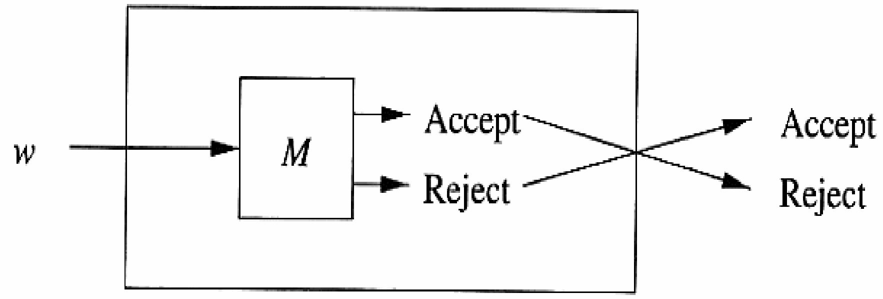
Theorem 9.3: If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.



Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

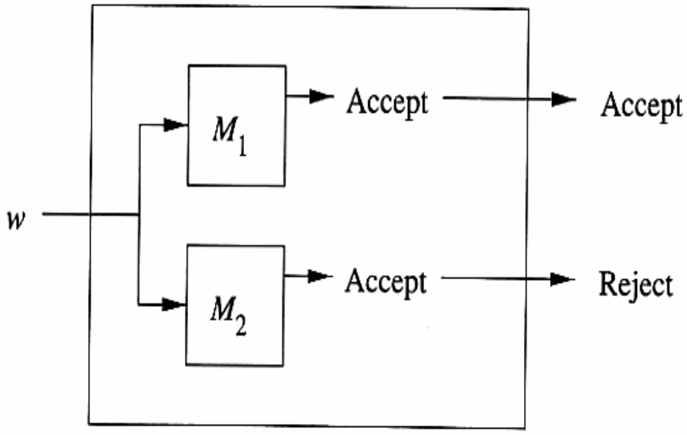


Figure 9.4: Simulation of two TM's accepting a language and its complement

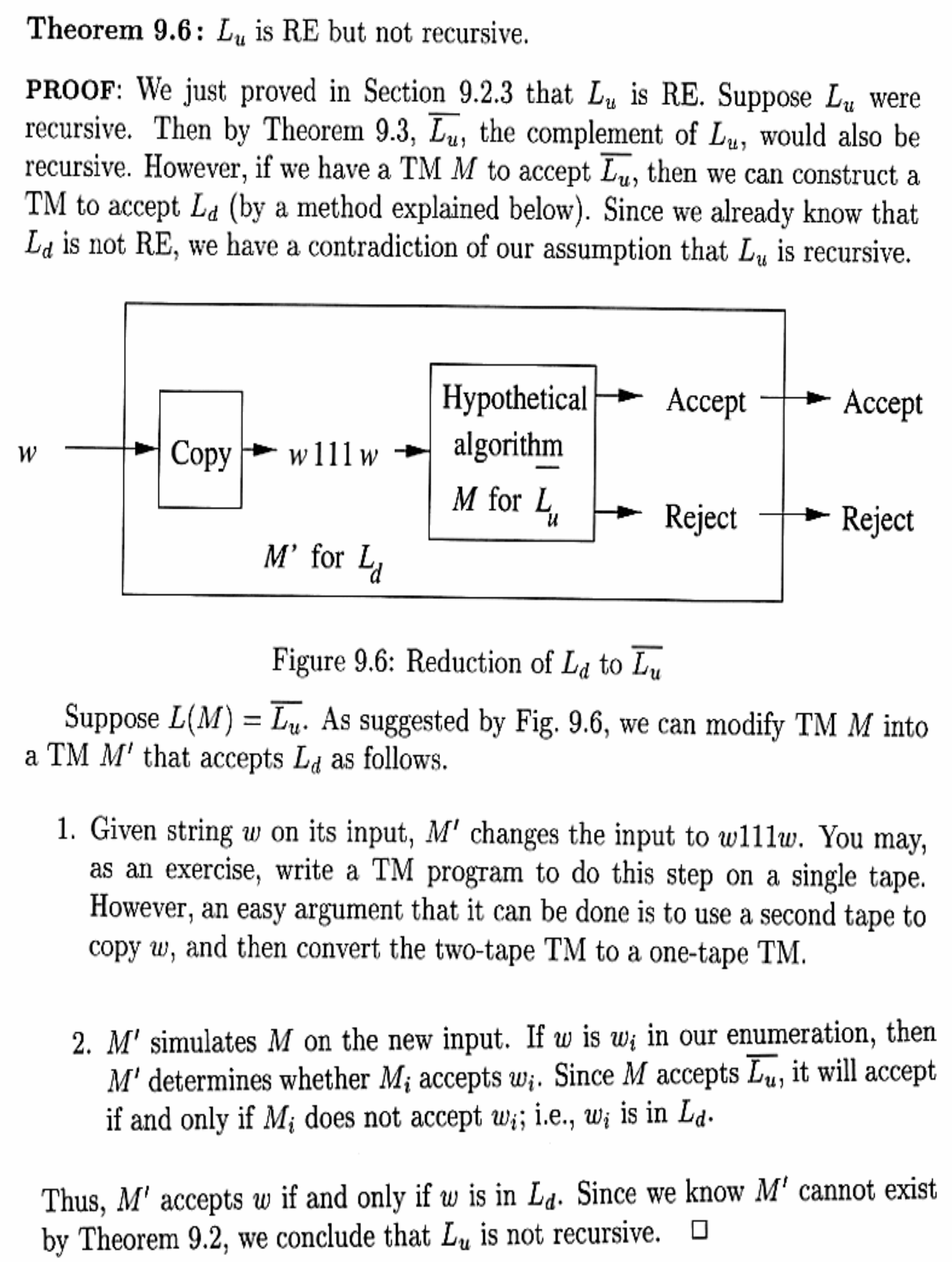
If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and

$L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

Universal
Language:

We define L_u, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(\mathbf{0}+\mathbf{1})^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_j in the list of binary-input Turing machines we developed in

Undecidability of Universal Language:



Theorem 9.6: L_u is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that L_u is RE. Suppose L_u were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of L_u, would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

Figure 9.6: Reduction of L_d to $\overline{L_u}$

Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.2, we conclude that L_u is not recursive. □

Problem -Reduction :
If P_1 reduced to P_2,
Then P_2 is at least as hard as P_1.
Theorem: If P_1 reduces to P_2 then,

- If P_1 is undecidable the so is P_2.
- If P_1 is Non-RE then so is P_2.

Post's Correspondence Problem (PCP)

A post correspondence system consists of a finite set of ordered pairs $(x_i, y_i),\ i = 1, 2, \cdots, n,$ where $x_i, y_i \in \Sigma^+$ for some alphabet Σ.

Any sequence of numbers $i_1, i_2, \cdots, i_k \quad s-t.$

is called a solution to a Post Correspondence System.

$x_{i_1}, x_{i_2}, \cdots, x_{i_k} = y_{i_1}, y_{i_2}, \cdots, y_{i_k}$ The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solutions.

Example 1 : Consider the post correspondence system

$\{(aa, aab), (bb, ba), (abb, b)\}$ The list 1,2,1,3 is a solution to it.

Because

$$x_1 x_2 x_1 x_3 = y_1 y_2 y_1 y_3$$

$$\underbrace{aa}_{x_1}\underbrace{bb}_{x_2}\underbrace{aa}_{x_1}\underbrace{abb}_{x_3} = \underbrace{aab}_{y_1}\underbrace{ba}_{y_2}\underbrace{aab}_{y_1}\underbrace{b}_{y_3}$$

$$aabbaaabb = aabbaaabb$$

i	x_i	y_i
1	aa	aab
2	bb	ba
3	abb	b

(A post correspondence system is also denoted as an instance of the PCP)

Example 2 : The following PCP instance has no solution

i	x_i	y_i
1	aab	aa
2	a	baa

This can be proved as follows. (x_2, y_2) cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol (a in LHS and 'b' in RHS). So, we must start with (x_1, y_1). The next pair must be (x_2, y_2) so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If (x_1, y_1) is selected next, then would be mismatched in the 7 th symbol

(b in LHS and a in RHS). If (x_2, y_2) is selected, instead, there will not be any choice to match the both side in the next step.

Example3 : The list 1,3,2,3 is a solution to the following PCP instance.

i	x_i	y_i
1	1	101
2	10	00
3	011	11

The following properties can easily be proved.

Proposition The Post Correspondence System

$\{(a^{i_1}, a^{j_1}), (a^{i_2}, a^{j_2}), \cdots, (a^{i_n}, a^{j_n})\}$ has solutions if and only if

$\exists k$ such that $i_k = j_k$ or

$\exists k$ and l such that $i_k > j_k$ and $i_l < j_l$

Corollary : PCP over one-letter alphabet is decidable.

Proposition Any PCP instance over an alphabet Σ with $|\Sigma| \geq 2$ is equivalent to a PCP instance over an alphabet Γ with $|\Gamma| = 2$

Proof : Let $\Sigma = \{a_1, a_2, \cdots, a_k\}, k > 2.$

Consider $\Gamma = \{0,1\}$ We can now encode every $a_i \in \Sigma, 1 \leq i \leq k$ as $10^i 1.$ any PCP instance over Σ will now have only two symbols, 0 and 1 and, hence, is equivalent to a PCP instance over Γ

Theorem : PCP is undecidable. That is, there is no algorithm that determines whether an arbitrary Post Correspondence System has a solution.

Proof: The halting problem of turning machine can be reduced to PCP to show the undecidability of PCP. Since halting problem of TM is undecidable (already proved), This reduction shows that PCP is also undecidable. The proof is little bit lengthy and left as an exercise.

Some undecidable problem in context-free languages

We can use the undecidability of PCP to show that many problem concerning the context-free languages are undecidable. To prove this we reduce the PCP to each of these problem. The following discussion makes it clear how PCP can be used to serve this purpose.

Let $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ be a Post Correspondence System over the alphabet Σ. We construct two CFG's G_x and G_y from the ordered pairs *x,y* respectively as follows.

$G_x = (N_x, \Sigma_x, P_x, S_x)$ and

$G_y = (N_y, \Sigma_y, P_y, S_y)$ where

$$G_y = (N_y, \Sigma_y, P_y, S_y)$$

$$N_x = \{S_x\} \text{ and } N_y = \{S_y\}$$

$$\Sigma_x = \Sigma_y = \Sigma \bigcup \{1, 2, \cdots, n\},$$

$$P_x = \{S_x \to x_i S_x i,\ S_x \to x_i i \mid i = 1, 2, \cdots, n\}$$

and $P_y = \{S_y \to y_i S_y i,\ S_y \to y_i i \mid i = 1, 2, \cdots n\}$

it is clear that the grammar G_x generates the strings that can appear in the LHS of a sequence while solving the PCP followed by a sequence of numbers. The sequence of number at the end records the sequence of strings from the PCP instance (in reverse order) that generates the string. Similarly, G_y generates the strings that can be obtained from the RHS of a sequence and the corresponding sequence of numbers (in reverse order).

Now, if the Post Correspondence System has a solution, then there must be a sequence

$$i_1 i_2, \cdots, i_k \ s.t$$

$$x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$$

According to the construction of G_x and G_y

$$S_x \overset{*}{\underset{G_x}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_k} i_k i_{k-1} \cdots i_2 i_1 \text{ and}$$

$$S_y \overset{*}{\underset{G_y}{\Rightarrow}} y_{i_1} y_{i_2} \cdots y_{i_k} i_k i_{k-1} \cdots i_2 i_1$$

In this case

$$x_{i_1} x_{i_2} \cdots x_{i_k} i_k \cdots i_2 i_1 = y_{i_1} y_{i_2} \cdots i_{i_k} i_k \cdots i_2 i_1 = w(\text{say})$$

Hence , $w \in L(G_x)$ and $w \in L(G_y)$ implying

$$L(G_x) \cap L(G_y) \neq \phi$$

Conversely, let $w \in L(G_x) \cap L(G_y)$

Hence, *w* must be in the form $w_1 w_2$ where $w_1 \in \Sigma^*$ and w_2 in a sequence $i_k i_{k-1} \cdots i_2 i_1$(since, only that kind of strings can be generated by each of G_x and G_y).

Now, the string $w_1 = x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$ is a solution to the Post Correspondence System.

It is interesting to note that we have here reduced PCP to the language of pairs of CFG,s whose intersection is nonempty. The following result is a direct conclusion of the above.

Theorem : Given any two CFG's G_1 and G_2 the question "Is $L(G_1) \cap L(G_2) = \phi ?$" is undecidable.

Proof: Assume for contradiction that there exists an algorithm *A* to decide this question. This would imply that PCP is decidable as shown below.

For any Post Correspondence System, *P* construct grammars G_x and G_y by using the constructions

elaborated already. We can now use the algorithm *A* to decide whether and $L(G_x) \cap L(G_y) \neq \phi$ Thus, PCP is decidable, a contradiction. So, such an algorithm does not exist.

If G_x and G_y are CFG's constructed from any arbitrary Post Correspondence System, than it is not difficult to show that $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are also context-free, even though the class of context-free languages are not closed under complementation.

$L(G_x), L(G_y)$ and their complements can be used in various ways to show that many other questions related to CFL's are undecidable. We prove here some of those.

Theorem : Foe any two arbitrary CFG's $G_1 \,\&\, G_2$, the following questions are undecidable

i. Is $L(G_1) = \Sigma^* ?$

ii. Is $L(G_1) = L(G_2) ?$

iii. Is $L(G_1) \subseteq L(G_2)$?

Proof :

i. If $L(G_1) = \Sigma^*$ then, $\overline{L(G_1)} = \phi$

Hence, it suffice to show that the question "Is $L(G_1) = \phi$?" is undecidable.

Since, $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are CFl's and CFL's are closed under union, $L = \overline{L(G_x)} \cup \overline{L(G_y)}$ is also context-free. By DeMorgan's theorem, $\overline{L} = L(G_x) \cap L(G_y)$

If there is an algorithm to decide whether $L(G_1) = \phi$ we can use it to decide whether $\overline{L} = L(G_x) \cap L(G_y) = \phi$ or not. But this problem has already been proved to be undecidable.

Hence there is no such algorithm to decide or not. $L(G_1) = \phi$

ii.

Let *P* be any arbitrary Post correspondence system and G_x and G_y are CFg's constructed from the pairs of strings.

$L_1 = \overline{L(G_x)} \cup \overline{L(G_y)}$ must be a CFL and let G_1generates L_1. That is,

$$L_1 = L(G_1) = \overline{L(G_x)} \cup \overline{L(G_y)} = \overline{L(G_x) \cap L(G_y)}$$

by De Morgan's theorem, as shown already, any string, $w \in L(G_x) \cap L(G_y)$ represents a solution to the PCP. Hence, $L(G_1)$ contains all but those strings representing the solution to the PCP.

Let $L(G_2) = (\Sigma \cup \{1, 2, \cdots, n\})^*$ for same CFG G_2.

It is now obvious that $L(G_1) = L(G_2)$ if and only if the PCP has no solutions, which is already proved to be undecidable. Hence, the question "Is $L(G_1) = L(G_2)$?" is undecidable.

iii.

Let G_1 be a CFG generating the language $\left(\Sigma \cup \{1, 2, \cdots n\}\right)^*$ and G_2 be a CFG generating $\overline{L(G_x)} \cup \overline{L(G_y)}$ where G_x and G_y are CFG.s constructed from same arbitrary instance of PCP.

$$L(G_1) \subseteq L(G_2) \text{ iff } \overline{L(G_x)} \cup \overline{L(G_y)} = \left(\Sigma \cup \{1, 2, \cdots n\}\right)^*$$

i.e. iff the PCP instance has no solutions as discussed in part (ii).

Hence the proof.

Theorem : It is undecidable whether an arbitrary CFG is ambiguous.

Proof : Consider an arbitrary instance of PCP and construct the CFG's G_x and G_y from the ordered pairs of strings.

We construct a new grammar *G* from G_x and G_y as follows.

$G = (N, \Sigma, P, S)$ where

$N = \{S, S_x, S_y\}$,

Σ is same as that of G_x and G_y.

$P = \{P_x \cup P_y \cup \{S \rightarrow S_x | S_y\}$

This constructions gives a reduction of PCP to the -------- of whether a CFG is ambiguous, thus leading to the undecidability of the given problem. That is, we will now show that the PCP has a solution if and only if *G* is ambiguous. (where *G* is constructed from an arbitrary instance of PCP).

Only if Assume that $i_1, i_2, \cdots i_k$ is a solution sequence to this instance of PCP.

Consider the following two derivation in $i_1, i_2, \cdots i_k$.

$$S \underset{G}{\overset{1}{\Rightarrow}} S_x \underset{G}{\overset{1}{\Rightarrow}} x_{i_1} S_x i_1 \underset{G}{\overset{1}{\Rightarrow}} x_{i_1} x_{i_2} S_x i_2 i_1$$
$$\underset{G}{\overset{*}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} S_x i_{k-1} \cdots i_2 i_1$$
$$\underset{G}{\overset{1}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} x_{i_k} i_k i_{k-1} \cdots i_2 i_1$$

$$S \overset{1}{\underset{G}{\Rightarrow}} S_y \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} S_y i_1 \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} S_y i_2 i_1$$
$$\overset{*}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots x y_{i_{k-1}} S_y i_{k-1} \cdots i_2 i_1$$
$$\overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots y_{i_{k-1}} y_k i_k i_{k-1} \cdots i_2 i_1$$

But ,

$$x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_k, \text{ since } \cdots i_1, i_2, \cdots i_k$$

is a solution to the PCP. Hence the same string of terminals $\left(x_{i_1} x_{i_2} \cdots x_{i_k}\right)$ has two derivations. Both these derivations are, clearly, leftmost. Hence *G* is ambiguous.

If It is important to note that any string of terminals cannot have more than one derivation in G_x and G_y Because, every terminal string which are derivable under these grammars ends with a sequence of integers $i_k i_{k-1} \cdots i_2 i_1$ This sequence uniquely determines which productions must be used at every step of the derivation.

Hence, if a terminal string, $w \in L(G)$, has two leftmost derivations, then one of them must begin with the step.

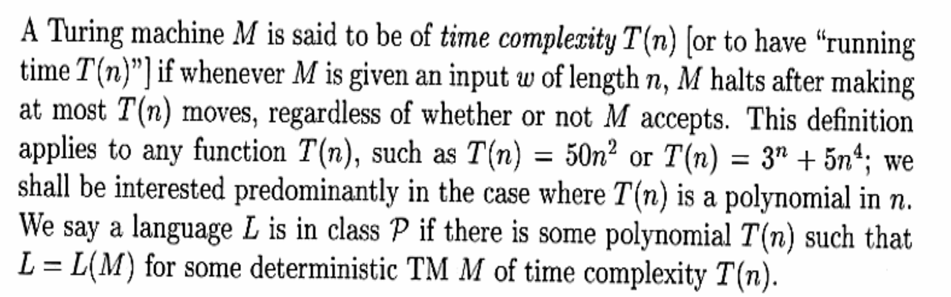
then continues with derivations under G_y

In both derivations the resulting string must end with a sequence $i_p i_{p-1} \cdots i_2 i_1$ for same $p \geq 1$ The reverse of this sequence must be a solution to the PCP, because the string that precede in one case is $x_1 x_2 \cdots x_{i_{p-1}} x_{i_p}$ and $y_1 y_2 \cdots y_{i_{p-1}} y_{i_p}$ in the other case. Since the string derived in both cases are identical, the sequence $i_1, i_2, \cdots i_{p-1}, i_p$

must be a solution to the PCP.

Hence the proof

Class p-problem solvable in polynomial time:



A Turing machine M is said to be of *time complexity* $T(n)$ [or to have "running time $T(n)$"] if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in n. We say a language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.

Non deterministic polynomial time:

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p is said to be non polynomial time NTM.

- NP is the set of languags that are accepted by polynomial time NTM's
- Many problems are in NP but appear not to be in p.
- One of the great mathematical questions of our age: is there anything in NP that is not in p?

NP-complete problems:

If We cannot resolve the "p=np question, we can at least demonstrate that certain problems in NP are the hardest , in the sense that if any one of them were in P , then P=NP.

- These are called NP-complete.
- Intellectual leverage: Each NP-complete problem's apparent difficulty reinforces the belief that they are all hard.

Methods for proving NP-Complete problems:

- Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another.
- If P1 PTR to P2 and P2 is in P1 the so is P1.
- Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.
- Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.