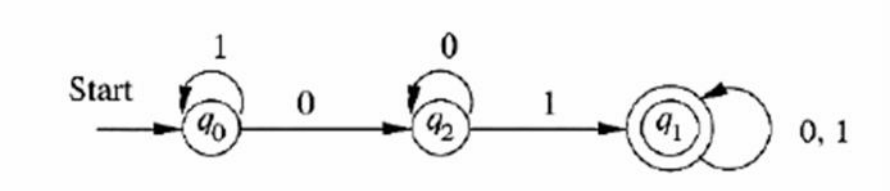
QUESTION BANK SOLUTION

Unit 1

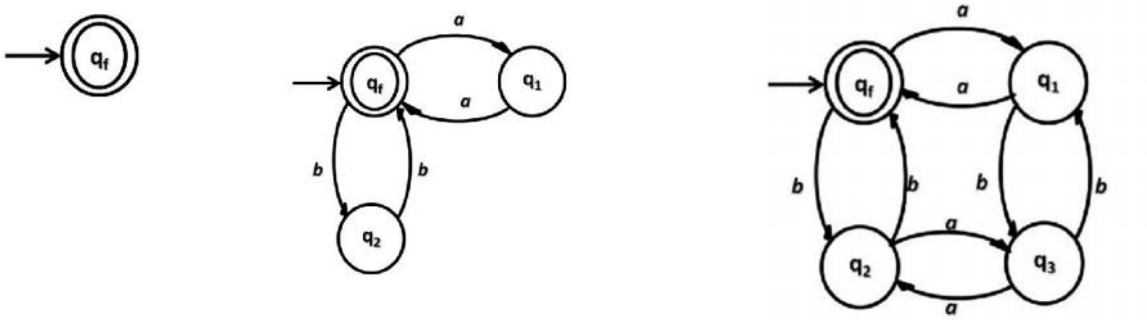
Introduction to Finite Automata

1. **Obtain DFAs to accept strings of a's and b's having exactly one a.(5m)(Jun-Jul 10)**



2. **Obtain a DFA to accept strings of a's and b's having even number of a's and b's.(5m)(Jun-Jul 10)**

L = {Œ,aabb,abab,baba,baab,bbaa,aabbaa,---------}



3. **Give Applications of Finite Automata. (5m)(Jun-Jul 10)**

String Processing

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

Finite-State Machines

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

Statecharts

Statecharts model tasks as a set of states and actions. They extend FA diagrams.

Lexical Analysis

In compiling a program, the first step is lexical analysis. This isolates keywords, identifiers etc., while eliminating irrelevant symbols. A token is a category, for example "identifier", "relation operator" or specific keyword.

4. **Define DFA, NFA & Language? (5m)(Jun-Jul 10)**

Deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. 'Deterministic' refers to the uniqueness of the computation. **Nondeterministic finite automaton (NFA)** or nondeterministic finite state machine is a finite state machine where from each state and a given input symbol the automaton may jump into several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. Although the DFA and NFA have distinct definitions, a NFA can be translated to equivalent DFA using the subset construction algorithm
A language is any subset of

Languages:

1. The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0$: $\{\epsilon, 01, 0011, 000111, \ldots\}$.
2. The set of strings of 0's and 1's with an equal number of each:

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

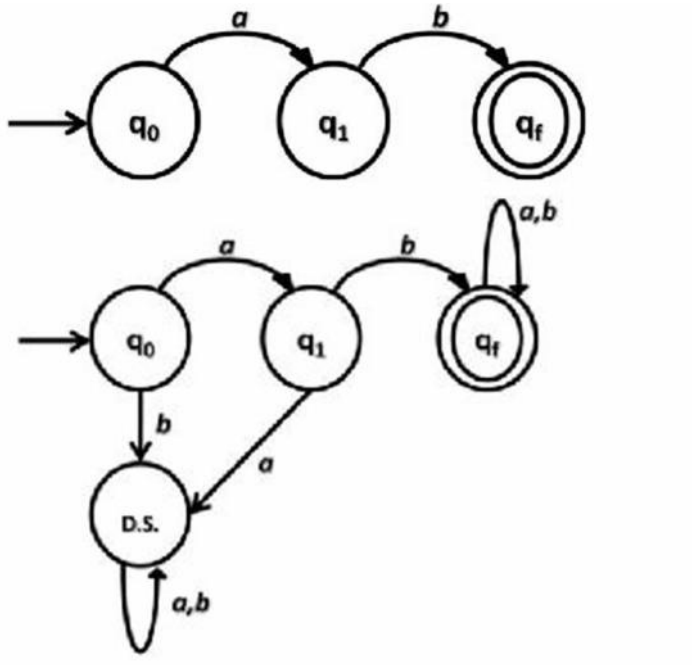
3. The set of binary numbers whose value is a prime:

$$\{10, 11, 101, 111, 1011, \ldots\}$$

4. Σ^* is a language for any alphabet Σ.
5. $\emptyset$, the empty language, is a language over any alphabet.
6. $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Notice that $\emptyset \neq \{\epsilon\}$; the former has no strings and the latter has one string.

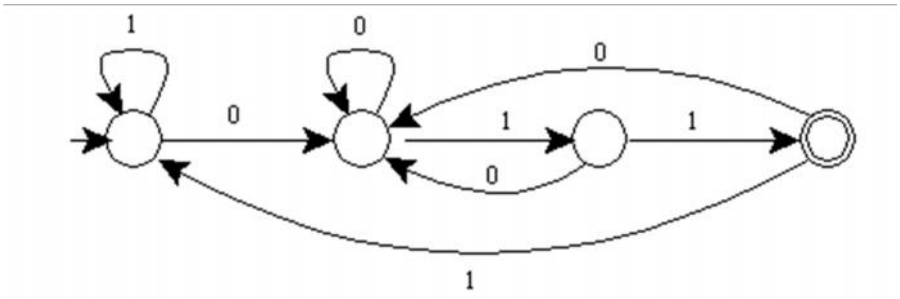
5. **Obtain a DFA to accept strings of a's and b's starting with the string ab. (6m)(Dec-Jan 10) (Jun-Jul 12)**

L = {ab,aba,abb,abab,abaa,abbb,abba ------}



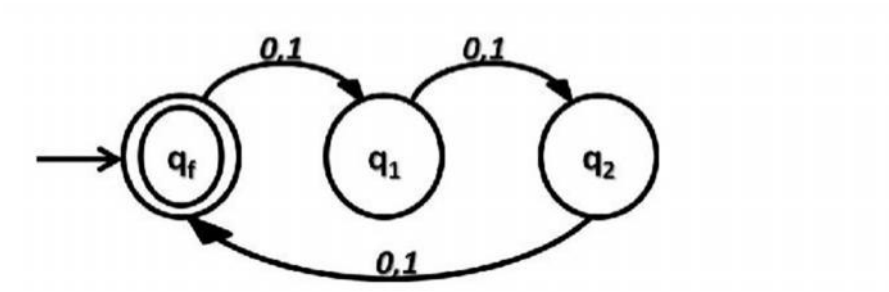
6. **Draw a DFA to accept string of 0's and 1's ending with the string 011. (4m)(Dec-Jan 10) (Jun-Jul 12)**

L= {011, 0011, 1011, 00011, 01011, 10011, 11011, ...},

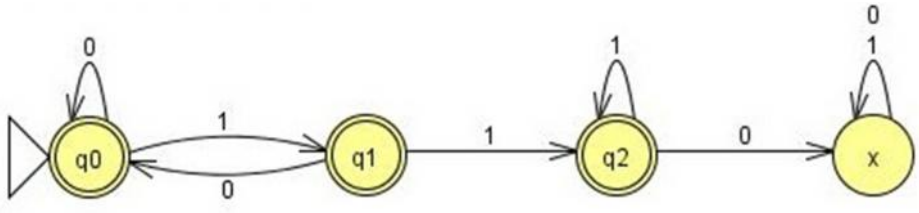


7. **Write DFA to accept strings of 0's, 1's & 2's beginning with a 0 followed by odd number of 1's and ending with a 2. (10m)(Dec-Jan 10) (Jun-Jul 12)**

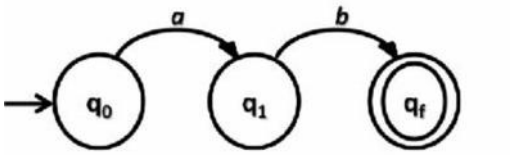
8. **Design a DFA to accept string of 0's & 1's when interpreted as binary numbers would be multiple of 3. (5m)(Jun-Jul 11)(Jun-Jul12)**

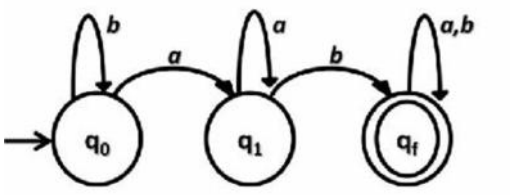


9. **Find closure of each state and give the set of all strings of length 3 or less accepted by automaton.(5m)(Jun-Jul 11) (Jun-Jul12)**



10. **Obtain a DFA to accept strings of a's and b's having a sub string aa. (5m)(Jun-Jul-11)**





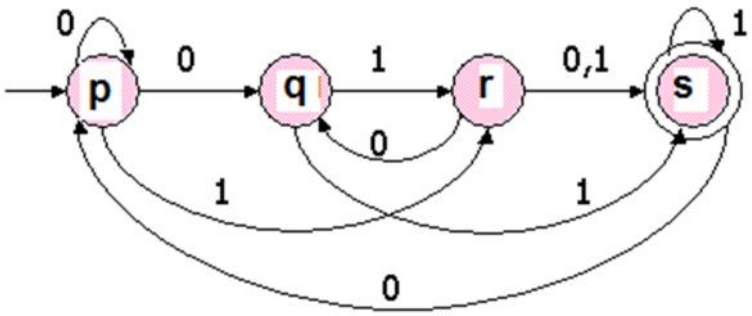
11. **Write Regular expression for the following L = { a^n b^m : m, n are even} L = { a^n,b^m : m>=2, n>=2}.(5m)(Jun-Jul 11)**

ab numberOf(a)=1 and numberOf(b)=1 > 1/2
abb numberOf(a)=1 and numberOf(b)=2 > 1/2
abbb numberOf(a)=1 and numberOf(b)=3 > 1/2
aabb numberOf(a)=2 and numberOf(b)=2 > 2/2 = 1
aaabb numberOf(a)=3 and numberOf(b)=2 > 3/2 = 1.5
aaaabb numberOf(a)=4 and numberOf(b)=2 = 4/2 = 2

12.

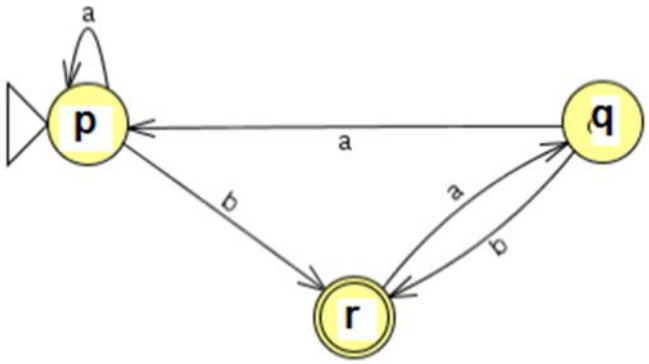
δ	0	1
p	{p,r}	{q}
q	{r,s}	{p}
*	{p,s}	{r}
	{q,r}	

Convert above automaton to a DFA.(10m)(Dec-Jan 11)

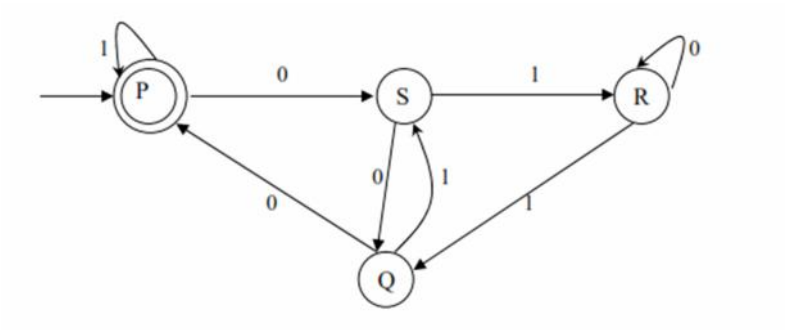


13. **Convert following NFA to DFA using subset construction method.(10m)(Dec-Jan 11)**

δ		a	b
p q	{r} ∅ {p,q}	{q} {p} {r}	{p, r} ∅ {p



14. **Convert the following DFA to Regular Expression (10m)(Dec –Jan-12)**



Here is a transition table for a DFA:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

15. **Define NFA. With example explain the extended transition function(5m)(Dec –Jan-12)**

As with a DFA, we can de¯ne the extended transition function of an NFA. If the transition function is ±, we usually denote the extended transition function by ^±. The basis is that ^±(q; a) := fqg:

For the induction step, let S be ^±(q; x). Then ^±(p; xa) := [p2S ±(p; a):

The Subset Construction

In order to show that DFAs and NFAs have the same computational power, gave the subset construction, which, given an NFA, constructs a DFA that accepts the same language.The alphabet of the new DFA is the same as that of the NFA. If Q is the set of states of the given NFA, then the set Q0 of states of the new DFA is P(Q), the power set of Q, that is, the set of all subsets of Q. In another words, a state of the new DFA is a set of states of the NFA.If q0 is the start state of the NFA, then fq0g is the start state of the new DFA. A state in the new DFA is accepting if it contains an accepting state of the NFA. If ± is the transition function of the NFA, then we de¯ne the transition function ±0 of the new DFA as follows. Where S is a subset of Q and a is a symbol: ± 0 (S; a) := [p2S ±(p; a):

16. **Explain the ground rules of finite automata.(5m)(Dec-Jan12)**

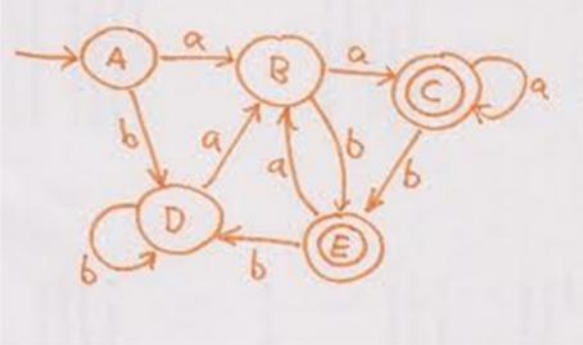
There are three participants: the customer, the store, and the bank. We assume for simplicity that there is only one "money" file in existence. The customer may decide to transfer this money file to the store, which will then redeem the file from the bank (i.e., get the bank to issue a new money file belonging to the store rather than the customer) and ship goods to the customer. In addition, the customer has the option to cancel the file. That is, the customer may ask the bank to place the money back in the customer's account, making the money

1. The customer may decide to *pay*. That is, the customer sends the money to the store.
2. The customer may decide to *cancel*. The money is sent to the bank with a message that the value of the money is to be added to the customer's bank account.
3. The store may *ship* goods to the customer.
4. The store may *redeem* the money. That is, the money is sent to the bank with a request that its value be given to the store.
5. The bank may *transfer* the money by creating a new, suitably encrypted money file and sending it to the store.

UNIT 2

Finite Automata, Regular Expressions

1. **P.T. Let R be a regular expression. Then there exists a finite automaton M = (Q, ∑, G, q0, A) which accepts L(R). (10m)(June- july 2010)**



2. **Define derivation , types of derivation , Derivation tree & ambiguous grammar. Give example for each. (4m)(June- July 2010)**

Derivation Tree

Derivation trees (also called "parse trees" in Sethi's book) are a way to represent the generation of strings in a grammar. They also give information about the structure of the strings, i.e. the way they are organized in syntactical categories.

Definition

Given a grammar G = < T , N , s , P > , a derivation tree t for G is a tree such that:

the root is labeled by s

the leaves are labeled by terminal symbols

each intermediate node is labeled by a non-terminal symbol, and, if its label is A, then its children are labeled by symbols s1 , s2 , ... , sn such that there exists a production A ::= s1 s2 ... sn in P

The labels of the leaves (fringe) represent the string generated by t. We will indicate it by string(t).

It is easy to see that a derivation tree represents a set of derivations (usually more than one) for the same string, and that for each derivation there is a derivation tree for the same string. Hence L(G) coincides with the set of strings generated by all possible derivation trees for G. More formally, if we denote by DT(G) the set of all derivation trees for G, we have the following result:

Proposition

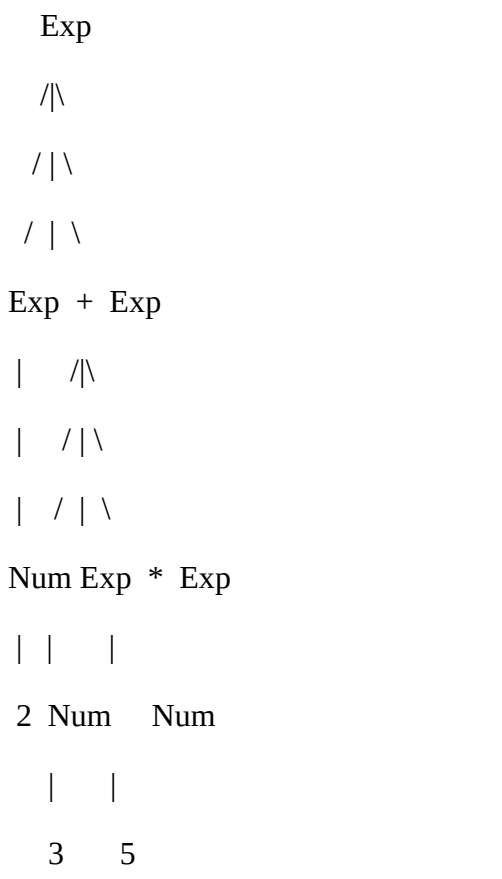
L(G) = { alpha in T* | alpha = string(t) for some t in DT(G) }

Example

Let us consider again the language of numerical expressions, with productions

Exp ::= Num | Exp + Exp | Exp * Exp

We have that a possible derivation tree for the string 2 + 3 * 5 is the following



This tree corresponds to several derivations for the same string, which differ only for the choice of the non-terminal to expand at each derivation step.

Ambiguity

The structure of an expression is usually essential to interpret its meaning. The expression 2 + 3 * 5 for example has two different values depending on its intended structure: If we assume it to be 2 + (3 * 5) (i.e. 3 and 5 grouped together by *) then the result is 17. If, on the other hand, we assume it to be (2 + 3) * 5, then the result is 25. In order to avoid this kind of ambiguity, it is essential that the grammar generates

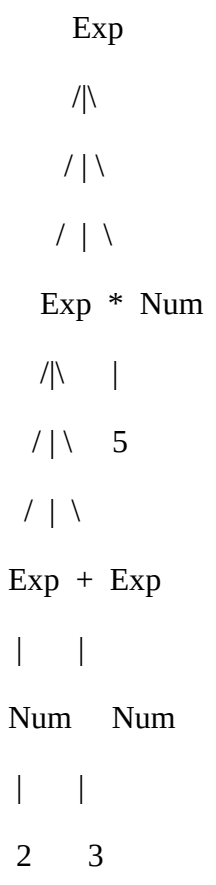
only one possible structure for each string in the language. Since the structure is represented by the derivation tree, we have the following definition:

Definition

A grammar G is ambiguous if there exist a string in L(G) which can be derived by two (or more) different derivation trees.

Example

The grammar in the example above is ambiguous, in fact the string 2 + 3 * 5 can be generated also by the following tree:



This tree corresponds to the grouping (2 + 3) * 5, while the tree in the example above corresponds to 2 + (3 * 5).

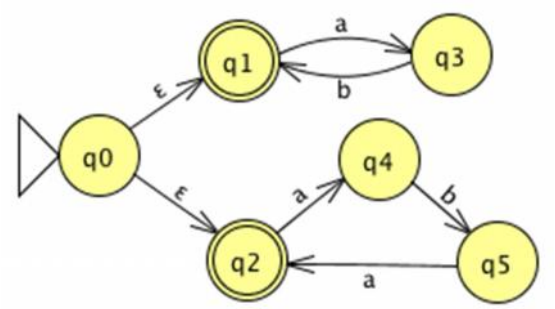
There are languages which are intrinsically ambiguous, i.e. it is not possible to eliminate their ambiguities without changing the language.

Definition

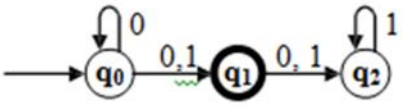
A language L is intrinsically ambiguous if can be generated only by ambiguous grammars, i.e. for every grammar G such that L=L(G), we have that G is ambiguous.

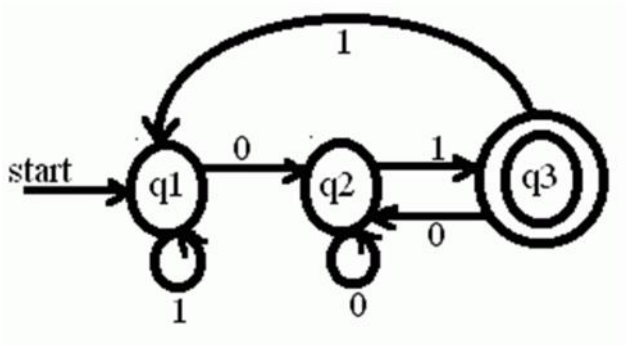
Luckily, languages which are interesting from the point of view of programming usually are not intrinsically ambiguous, and therefore we can find non-ambiguous grammars which generates them. When a (non-intrinsically ambiguous) language L is presented by an ambiguous grammar G, "to eliminate the ambiguities of G" means to find another grammar G', which is non ambiguous, and which generates the same language L.

3. **Obtain an NFA to accept the following language L = {w | w abab^n or aba^n where n t 0} (6m)(June- July- 2010)**

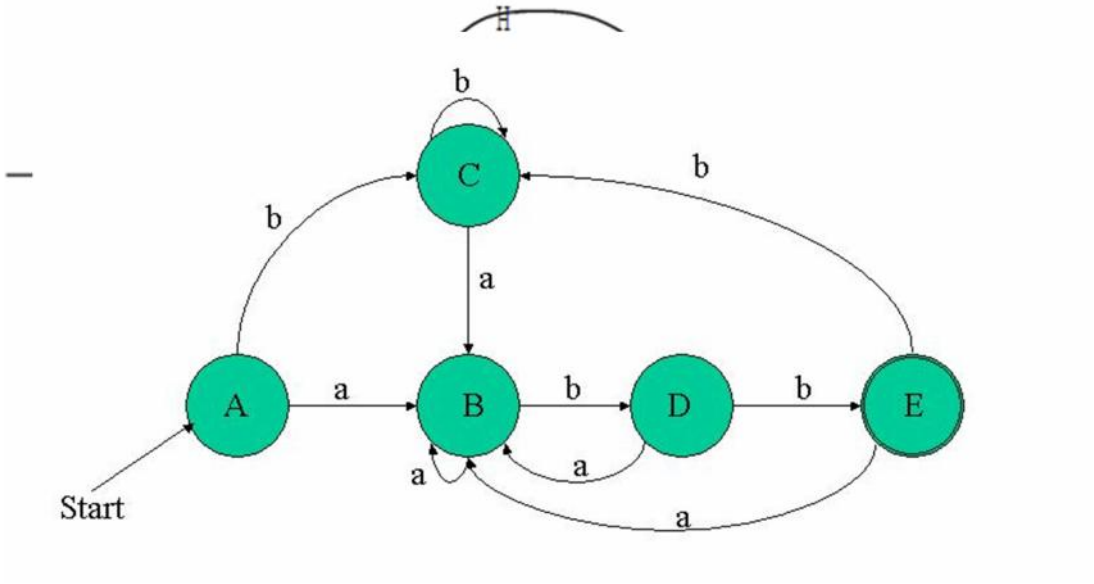


17. **Convert the following NFA into an equivalent DFA. (10m)(Dec- Jan 2010) (Jun-Jul 12)**

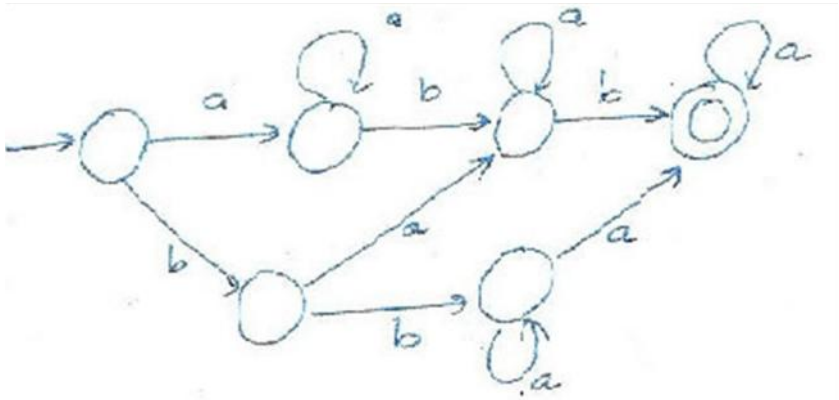




18. **Convert the following NFA to its equivalent DFA(10m)(Dec- Jan 2010) (Jun-Jul 12)**



4. **Obtain an NFA which accepts strings of a's and b's starting with the string ab.). (7m)(June- july 2011)**



5. **Define grammar? Explain Chomsky Hierarchy? Give an example (6m)(June- July 2011)**

A formal grammar of this type consists of:
a finite set of production rules (left-hand side right-hand side) where each side consists of a sequence of these symbols
a finite set of nonterminal symbols (indicating that some production rule can yet be applied)
a finite set of terminal symbols (indicating that no production rule can be applied)
a start symbol (a distinguished nonterminal symbol)

For example, the grammar with terminals $\{a, b\}$, nonterminals $\{S, A, B\}$, production rules

$S \to ABS$

$S \to \varepsilon$ (where ε is the empty string)

$BA \to AB$

$BS \to b$

$Bb \to bb$

$Ab \to ab$

$Aa \to aa$

The Chomsky hierarchy consists of the following levels:

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form with a nonterminal and , and strings of terminals and nonterminals. The strings and may be empty, but must be nonempty. The rule is allowed if does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form with a nonterminal and a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages – or rather the subset of deterministic context-free language – are the theoretical basis for the phrase structure of most programming languages, though their syntax also includes context-sensitive name resolution due to declarations and scope. Often a subset of grammars are used to make parsing easier, such as by an LL parser.

Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal (right regular). Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal (left regular); these generate the same languages – however, if left-regular rules and right-regular rules are combined, the language need no longer be regular. The rule is also allowed here if does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

6. **Is the following grammar ambiguous (7m)(June- July 2011)**

S -> aB | bA

A -> aS | bAA |a

B -> bS | aBB | b

It is ambiguous because there are two di®erent leftmost derivations for

the string aaa:

$$
\begin{aligned}
S &\Rightarrow aA \\
&\Rightarrow aaB \\
&\Rightarrow aaaS \\
&\Rightarrow aaa
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow aA \\
&\Rightarrow aaC \\
&\Rightarrow aaaS \\
&\Rightarrow aaa
\end{aligned}
$$

7. **Obtain grammar to generate string consisting of any number of a's and b's with at least one b. (5m)(Dec- Jan 2011) (Jun-Jul12)**

S	S→bA
bA	A→bAA
bbAA	A→aS
bbaSA	S→aB
bbaaBA	B→b
bbaabA	A→a
bbaaba	

8. **Obtain a grammar to generate the following language L ={WWR Where W {a, b}*}. (5m)(Dec- Jan 2011) (Jun-Jul12)**

- S -> SA
  S -> A
  A -> bSe
  A -> be

S
A
bSe
bSAe
bAAe
bbeAe
bbebee

Example: b and e matched as parentheses

9. **Obtain a grammar to generate the following language: L = { $0^m 1^m 2^n$ | m>= 1 and n>=0}. (5m)(Dec- Jan 2011)**

$$S \rightarrow TC \mid AR$$
$$T \rightarrow aTb \mid \lambda$$
$$C \rightarrow Cc \mid \lambda$$
$$R \rightarrow bRc \mid C$$
$$A \rightarrow Aa \mid \lambda$$

10. **Obtain a grammar to generate the following language: (5m)(Dec- Jan 2011)**
L = { w | $n_a(w) > n_b(w)$ }

$$S \rightarrow UT$$
$$U \rightarrow aa \mid ab \mid ba \mid bb$$
$$T \rightarrow aTa \mid bTb \mid V$$
$$V \rightarrow aa \mid ab \mid ba \mid bb$$

L = { $a^n b^m c^k$ | n+2m = k for n>=0, m>=0}

$$R \rightarrow XRX \mid S$$
$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}T\mathtt{a}$$
$$T \rightarrow XTX \mid X \mid \varepsilon$$
$$X \rightarrow \mathtt{a} \mid \mathtt{b}$$

11. **Define PDA. Obtain PDA to accept the language L = {$a^n b^n$ | n>=1} by a final state. (5m)(Dec- Jan 2011) (Jun-Jul12)**

$$S \rightarrow aaaA \mid aA \mid aaA \mid \lambda$$
$$A \rightarrow aAb \mid B$$
$$B \rightarrow Bb \mid \lambda$$

12. **Write a short note on application of context free grammar. (7m)(Dec- Jan 2012)**

Well-formed parentheses

The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

S → SS
S → (S)
S → ()

The first rule allows Ss to multiply; the second rule allows Ss to become enclosed by matching parentheses; and the third rule terminates the recursion.

Well-formed nested parentheses and square brackets

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

S → SS
S → ()
S → (S)
S → []
S → [S]

with terminal symbols [ ] ( ) and nonterminal S.

The following sequence can be derived in that grammar:

([[[()() [ ][ ]]]([ ])])

A regular grammar

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

S → a
S → aS

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

Matching pairs

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:
S → aSb
S → ab
This grammar generates the language , which is not regular (according to the Pumping Lemma for regular languages).
The special character ε stands for the empty string. By changing the above grammar to
S → aSb | ε
we obtain a grammar generating the language instead. This differs only in that it contains the empty string while the original grammar did not.

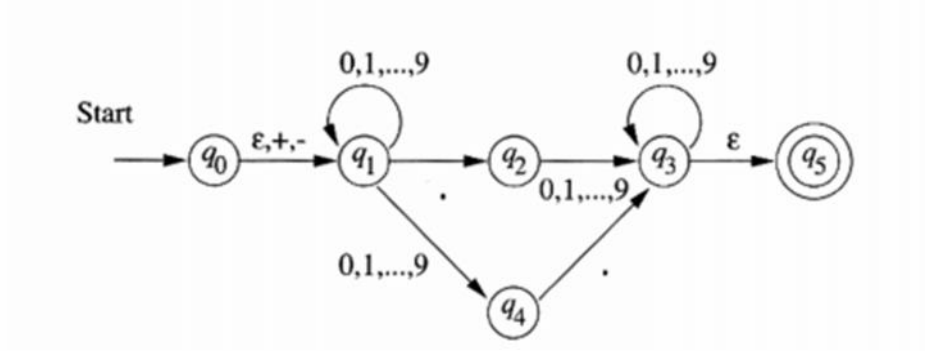
Algebraic expressions

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:
S → x
S → y
S → z
S → S + S
S → S - S
S → S * S
S → S / S
S → (S)
This grammar can, for example, generate the string
(x + y) * x - z * y / (x + x)
as follows:
S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ (S) * S - S / S (by rule 8, applied to the leftmost S)
→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)
→ (S + S) * S - S / (S) (etc.)
→ (S + S) * S - S * S / (S)
→ (S + S) * S - S * S / (S + S)
→ (x + S) * S - S * S / (S + S)
→ (x + y) * S - S * S / (S + S)
→ (x + y) * x - S * y / (S + S)
→ (x + y) * x - S * y / (x + S)
→ (x + y) * x - z * y / (x + S)
→ (x + y) * x - z * y / (x + x)
Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)

could be done in the opposite order:
→ S - S / S (by rule 7, applied to the rightmost S)
→ S * S - S / S (by rule 6, applied to the leftmost S)

13. **Explain finite automata with epsilon transition. (7m)(Dec-Jan 12)**

An informal treatment of €-NFA's, using transition diagrams with f allowed as a label. In the examples to follow, think of the automaton as accepting those sequences of labels along paths from the start state to an accepting state. However, each e along a path is "invisible" j i.e., it contributes nothing to the string along the path.
In Fig. is an €-NFAthat a.ccepts decimal numbers con- sisting of:2. A string of digits,
1. An optional + or - sign,
3. A decimal point, and
4. Another string of digits. Either this string of digits, or the string (2) can
be empty, but at least one of the two strings of digits must be nonempty.



14. **Explain the application of regular expression (6m)(Dec-Jan 12) (Jun-Jul12)**

• Search commands such as the UNIX grep or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems. These systems use a regular-expression-like notation for describing pat-terns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton on the file being searched.
• Lexical-analyzer generators, such as Lex or Flex. Recall that a lexical analyzer is the component of a compiler that breaks the source program into logical units (called tokens) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g., while),identifiers (e.g., any letter followed by zero or more letters and/or digits),and Sig,TIS,such as + or <=. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, andproduces a DFA that recognizes which token appears next on the input.

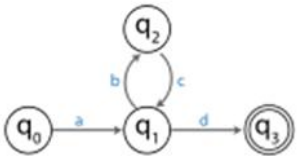
Unit 3

Regular Languages, Properties of Regular Languages

1. **Prove pumping lemma? (4m)(June-July 2010)**

For every regular language there is a finite state automaton (FSA) that accepts the language. The number of states in such an FSA are counted and that count is used as the pumping length *p*. For a string of length at least *p*, let s_0 be the start state and let s_1, ..., s_p be the sequence of the next *p* states visited as the string is emitted. Because the FSA has only *p* states, within this sequence of $p + 1$ visited states there must be at least one state that is repeated. Write *S* for such a state. The transitions that take the machine from the first encounter of state *S* to the second encounter of state *S* match some string. This string is called *y* in the lemma, and since the machine will match a string without the *y* portion, or the string *y* can be repeated any number of times, the conditions of the lemma are satisfied.

For example, the following image shows an FSA.



The FSA accepts the string: abcd. Since this string has a length which is at least as large as the number of states, which is four, the pigeonhole principle indicates that there must be at least one repeated state among the start state and the next four visited states. In this example, only q_1 is a repeated state. Since the substringbc takes the machine through transitions that start at state q_1 and end at state q_1, that portion could be repeated and the FSA would still accept, giving the stringabcbcd. Alternatively, the bc portion could be removed and the FSA would still accept giving the string ad. In terms of the pumping lemma, the string abcd is broken into an *x* portion a, a *y* portion bc and a *z* portion d.

2. **Prove that L={w|w is a palindrome on {a,b}*} is not regular. i.e., L={aabaa, aba, abbbba,…} (8m)(June-July 2010)**

The case n = 0 just means that u = ε (so ε always matches r∗); and the case n = 1 just means that u matches r (so any string matching r also matches r∗). For example, if Σ = {a, b, c}
and r = ab, then the strings matching r∗ are ε, ab, abab, ababab, etc.
Note that we didn't include a regular expression for the '∗' occurring in the UNIX examples on Slide 1. However, once we know which alphabet we are referring to, Σ = {a1, a2, . . . , an} say, we can get the effect of ∗ using the regular expression
(a1|a2| . . . |an)∗ which is indeed matched by any string in Σ∗ (because a1|a2| . . . |an is matched by any symbol in Σ)

3. **Prove that L={ all strings of 1's whose length is prime} is not regular. i.e., L={1^2,1^3 ,1^5 ,1^7 ,1^{11} ,----} (8m)(June-July 2010)**

Suppose the statement is true, and this langauge is regular. Then there exists a FSA(finite state automaton) that recognizes this language, which we call M. The pumping lemma says that there exists a natural

number p such that for every string s in L(M) of length at least p, there is a decompositon of s=xyz such that:

|y| > 0
|xy| <= p

Now, we can assume that there is a string w in L(M) such that |w|=k is the first prime number greater than p since there are infinitely many prime numbers. Because w is in L(M) and |w| > p, w can be decomposed as w=xyz that satisfies the above conditions.

Now consider the string . By the condition 3 above, v is in L(M). Thus, the length of v must be a prime number. But . Clearly, k | k(1+|y|) and k > 1. Hence |v| is not prime. This contradiction implies that the supposition is false, and the given langauge is not regular**.**

4.**Let M = (Q, ¦, G, q0, A) be an FA recognizing the language L. Then there exists an equivalent regular expression R for the regular language L such that L = L(R). (8m)(Dec- Jan 2010) (Jun-Jul12)**

Let n be the pumping-lemma constant. Test all strings of length between n and 2n-1 for membership in L. If we find even one such string, then L is infinite. The reason is that the pumping lemma applies to such a string, and it can be ``pumped" to show an infinite sequence of strings are in L.

Suppose, however, that there are no strings in L whose length is in the range n to 2n-1. We claim there are no strings in L of length 2n or more, and thus there are only a finite number of strings in L. In proof, suppose w is a string in L of length at least 2n, and w is as short as any string in L that has length at least 2n. Then the pumping lemma applies to w, and we can write w = xyz, where xz is also in L. How long could xz be? It can't be as long as 2n, because it is shorter than w, and w is as short as any string in L of length 2n or more. n, because xz is at most n shorter than w. Thus, xz is of length between n and 2n-1, which is a contradiction, since we assumed there were no strings in L with a length in that range.

5.**What is the language accepted by the following FA. (6m)(Dec-Jan 2010)**

{w ∈ {a, b}* : each ‘a’ in w is immediately preceded and followed by a ‘b’}
{w ∈ {a, b}* : w has abab as a substring}
{w ∈ {a, b}* : w has neither aa nor bb as a substring}
{w ∈ {a, b}* : w has an odd number of a's and an even number of b's}
{w ∈ {a, b}* : w has both ab and ba as substrings}

19. **Write short note on Applications of Regular Expressions (6m)(Dec-Jan 2010) (Jun-Jul 12)**

The first enhancement to the regular-expression notation concerns the fact that most real applications deal with the ASCII character set. Our examples have typicaUy used a small alphabet, such as {O, I}. The existence of only two symbols allowed us to write succinct expressions like 0 +1 for "any character." However, if there were 128 characters, say, the same expression would involve listing them all, and would be highly inconvenient to write. Thus, UNIX regular expressions allow us to write character classes to represent large sets of characters as succinctly as possible. The rules for character classes are:

• The symbol. (dot) stands for "any character."

• The sequence [at a2 ... ad stands for the regular expression

This notation saves about half the characters, since we don't have to write the +-signs. For example, we could express the four characters used in C comparison operators by [<>=! J.

20. **Show that following languages are not regular (10m)(June-July 2011) (Jun-Jul 12)**

L={$a^n b^m$ | n, m ŧ0 and n<m }

To prove that L is not a regular language, we will use a proof by contradiction. Assume that L is regular. Then by the Pumping Lemma for Regular Languages, there exists a pumping length, p for L such that for any string s 2 L where jsj p, s = xyz subject to the following conditions:
(a) jyj > 0
(b) jxyj p, and
(c) 8i > 0; xyi
z 2 L.
Choose s = 0p10p
Clearly, jsj p and s 2 L. By condition (b) above, it follows that x and y are composed only of zeros. By condition (a), it follows that y = 0kfor some
k > 0. Per (c), we can take i = 0 and the resulting string will still be in L. Thus,
xy0
z should be in L. xy0
z = xz = 0(pk)10p
. But, this is clearly not in L. This is a contradiction with the pumping lemma. Therefore our assumption that L is regular is incorrect, and L is not a regular language.

L={$a^n b^m$ | n, m ŧ0 and n>m }

To prove that L is not a regular language, we will use a proof by contradiction. Assume that L is a regular language. Then by the Pumping Lemma for Regular Languages,there exists a pumping length p for L such that for any sring s 2 L where jsj p,
s = xyz subject to the following conditions:
(a) jyj > 0
(b) jxyj p, and
(c) 8i > 0; xyi

L={$a^n b^m c^m d^n$ | n, m ŧ1 }

To prove that L is not a regular language, we will use a proof by contradiction. Assume that L is a

regular language. Then by the Pumping Lemma for Regular Languages, there exists a pumping length p for L such that for any sring s 2 L where jsj p,
s = xyz subject to the following conditions:
(a) jyj > 0
(b) jxyj p, and
(c) 8i > 0; xyi

L={a^n | n is a perfect square }

Solution: One way of showing that L is not regular is through the following argument: If L is regular, then so is L^c; however, we showed in class that L^c is not regular and hence L cannot be regular. Since you were expressly forbidden from using the complement properties of regular languages, let us proceed to prove the non-regularity of L, using first principles.

Assume that L is regular and let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ denote the unique minimal state DFA for L, where the symbols in the tuple have their usual meaning. Let p denote the number associated by the Pumping Lemma with L; we know that $p = |Q|$. Let $G = \{a^{p^2+i} : i = 1, 2, \ldots 2p\}$; we can think of the strings in G as being indexed by their length after subtracting the offset p^2. Using w_i to refer to the string a^{p^2+i}, we have that $|w_i| + 1 = |w_{i+1}|$.
Let $R = \{\delta(q_0, w_i) : w_i \in G\}$; we use r_i to denote $\delta(q_0, w_i),\ w_i \in G$.

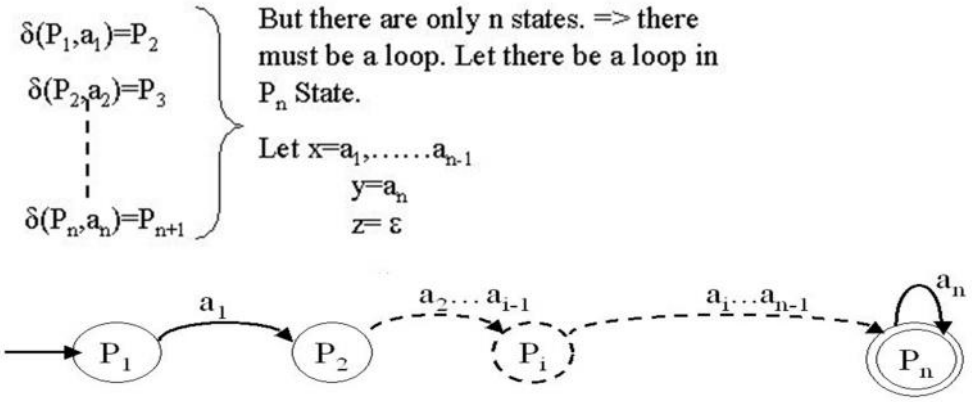
L={a^n | n is a perfect cube }

L is infinite. Suppose L is also regular. Then according to pumping lemma there exists an integer n such that for every string w in where |w| >= n, we can break w into three strings w = xyz such that:
|y| > 0, |xy| <= n and for all k>=0 , the string xyk
z is also in L.
Choose w to be w = 0s where s = n3 that is it is a perfect square.
Let w= 00000000000000000……00000 = xyz . (The length of w = s = n3 in this case.)
Let |xy| <= n and |y| = k.
That is w = 0000 0k
000…
X y z
So, |xyz| = |xz| + |y| = (n3
- k) + (k)
From pumping lemma, I can pump y any number of times and the new string should also belong to the language. Suppose I pump y twice then, the new string should belong to the language that
is it should have length that is a perfect cube

6. **Apply pumping lemma to following languages and understand why we cannot complete proof (10m)(June-July 2011)**

L={anaba| n t0 }

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



L={anbm | n, mt0 }

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

Let y = m.by PL xykz ∈L

| xykz | = | xz | + | yk |

Let k = p-m

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

= (p-m)

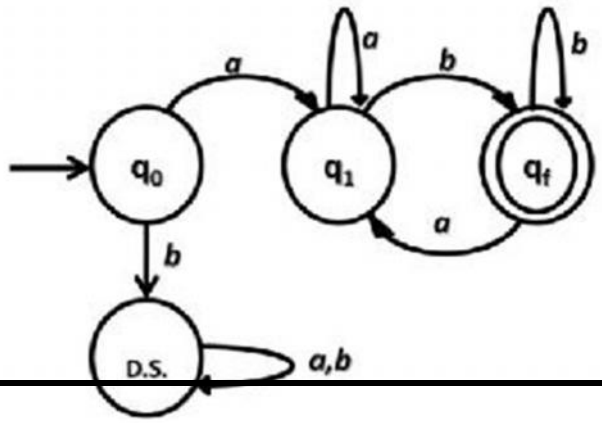
(1+m) -----

this can not be prime if p-m ≥ 2 or 1+m ≥ 2

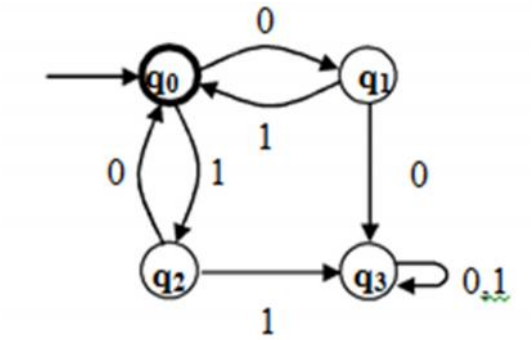
(1+m) ≥ 2 because m ≥ 1

Limiting case p=n+2 (p-m) ≥ 2 since m ≤n

7. **Obtain a DFA to accept strings of a's and b's starting with the string ab (10m)(Dec-Jan 2011)**



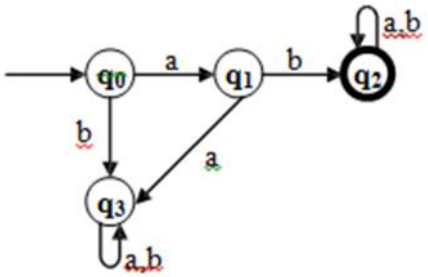
8. **Obtain a regular expression for the FA shown below: (10m)(Dec-Jan 2011) (Jun-Jul12)**



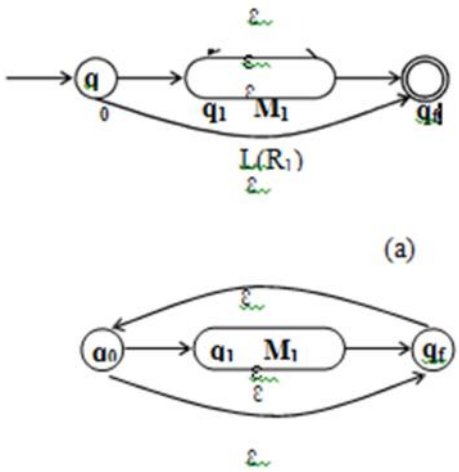
$R = R_1 . R_2$. We can construct an NFA which accepts $L(R_1)$ followed by $L(R_2)$ which can be represented as $L(R_1 . R_2)$

It is clear from figure that the machine after accepting L(R1) moves from state q1 to f1. Since there is a ε-transition, without any input there will be a transition from state f1 to state q2. In state q2, upon accepting L(R2), the machine moves to f2 which is the final
state. Thus, q1 which is the start state of machine M1 becomes the start state of thE combined machine M and f2 which is the final state of machine M2, becomes the final.

9. **Solve: (10m)(Dec-Jan12) (Jun-Jul12)**



R = (R1)*. We can construct an NFA which accepts either L(R1)*) as shown in figure. It can also be represented as shown. It is clear from figure 3.5 that the machine can either accept ε or any number of L(R1)s thus accepting the language L(R1)*. Here, q0 is the start state qf is the final state.
Obtain an NFA which accepts strings of a's and b's starting with the string ab.



10. **Explain Closure properties with an example. (10m)(Dec-Jan12)**

Closure Under Union

Theorem 4.4: If L and M are regular languages, then so is L U M.

PROOF: This proof is simple. Since L and M are regular, they have regular expressions; say L = £(R) and M = £(S). Then L U 1\11= L(R + S) by the definition of the + operator for regular expressions. 0

Closure Under Complementation

The theorem for union was made very easy by the use of the regular-expressio representation for the languages. However, let us next consider complementation. Do you see how to take a regular expression and change it into one that defines the complement language? Well neither do we. However, it can be done, because as we shall see in Theorem 4.5, it is easy to start with a OFA and construct a DFA that accepts the complement. Thus, starting with a regular expression, we could find a regular expression for its complement as follows:

1. Convert the regular expression to an f.-NFA.

2. Convert that f.-::'JFAto a OFA by the subset construction.

UNIT 4

Context-Free Grammars And Languages

1. P.T. If L and M are regular languages, then so is LĊM. (10m)June-July 2010)

There are two purely algebraic approaches to define regular languages. If:

- Σ is a finite alphabet,
- Σ* denotes the free monoid over Σ consisting of all strings over Σ,
- $f : \Sigma^* \to M$ is a monoid homomorphism where M is a *finite* monoid,
- S is a subset of M

then the set $\{w \in \Sigma^* \mid f(w) \in S\}$ is regular. Every regular language arises in this fashion.

If L is any subset of Σ*, one defines an equivalence relation ~ (called the syntactic relation) on Σ* as follows: $u \sim v$ is defined to mean

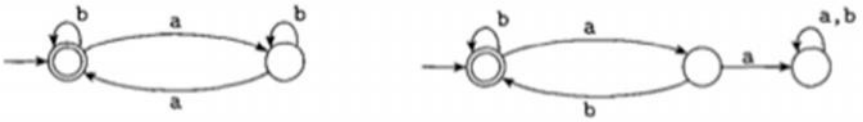
$uw \in L$ if and only if $vw \in L$ for all $w \in \Sigma^*$

The language L is regular if and only if the number of equivalence classes of ~ is finite (A proof of this is provided in the article on the syntactic monoid). When a language is regular, then the number of equivalence classes is equal to the number of states of the minimal deterministic finite automaton accepting L.

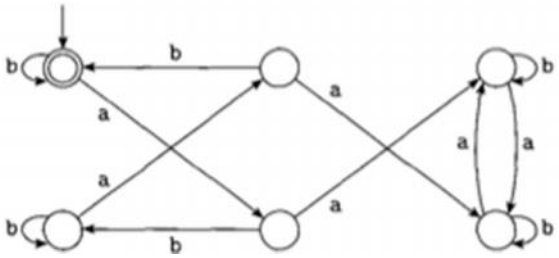
A similar set of statements can be formulated for a monoid $M \subset \Sigma^*$. In this case, equivalence over M leads to the concept of a recognizable language.

2. Write a DFA to accept the intersection of L1=(a+b)*a and L2=(a+b)*b that is for L1 ˆL2. (10m)(June-July 2010) (Jun-Jul12)

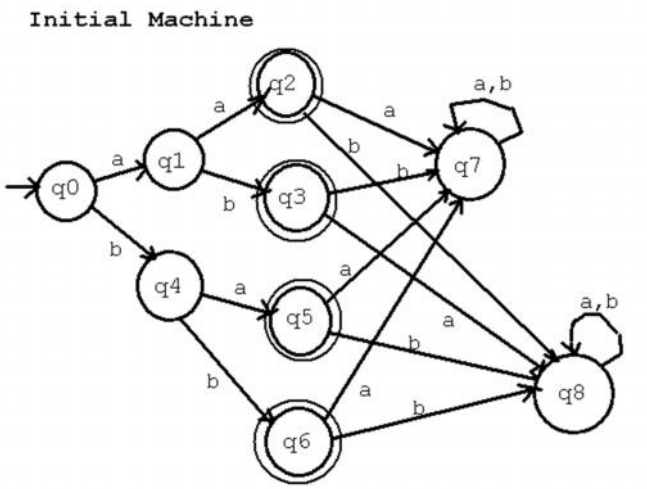
(d) These are DFAs for the two languages $\{w|\ w$ has an even number of a's$\}$ and $\{w|$ each a is followed by at least one b$\}$:



Combining them using the intersection construction gives the DFA:



3. Find the DFA to accept the intersection of L1=(a+b)*ab (a+b)* and L2=(a+b)*ba (a+b)* and that is for L1 ˆ L2 (10m)(Dec-Jan 2010) (Jun-Jul12)

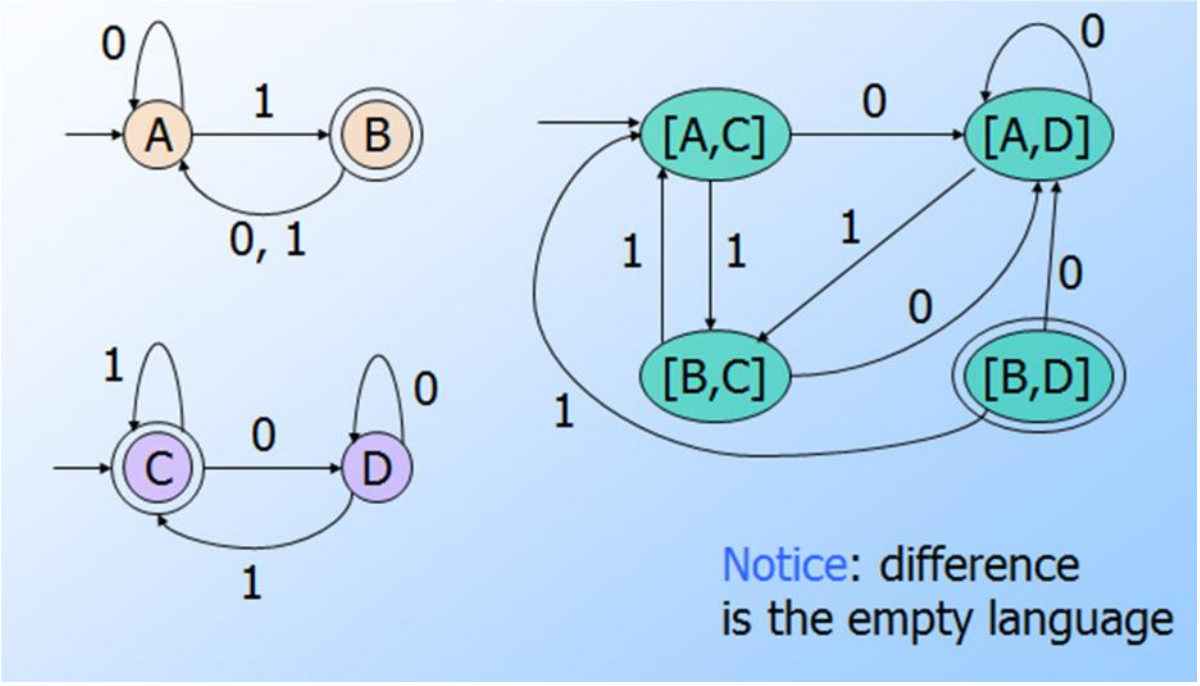


4. P.T. If L and M are regular languages, then so is L – M. (10m)(Dec-Jan 2010)

Proof: Let A and B be DFA's whose languages are L and M, respectively.

Construct C, the product automaton of A and B.

Make the final states of C be the pairs where A-state is final but B-state is not.



5. Design context-free grammar for the following cases (10m)(June-July 2011)

L={ 0n1n | n≥l }

E→I,
E→E+E,
E→E*E,
E→(E),
I→a|b|c

L={aibjck| i≠j or j≠k}

6. Generate grammar for RE 0*1(0+1)* (10m)(June-July 2011)

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

7. P.T. If L is a regular language over alphabet S, then L = 6* - L is also a regular language. (8m)(Dec-Jan 2011) (Jun-Jul12)

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

δ(q, ε, S) = {(q, 0S1), (q, A)}

δ(q, ε, A) = {(q, 1A0), (q, S), (q, ε)}

δ(q, 0, 0)

δ(q, 1, 1)

= {(q, ε)}

= {(q, ε)}

8. P.T. - If L is a regular language over alphabet 6, then, L = 6* - L is also a regular language. (8m)(Dec-Jan 2011)

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

δ(q, ε, S) = {(q, 0S1), (q, A)}

δ(q, ε, A) = {(q, 1A0), (q, S), (q, ε)}

δ(q, 0, 0)

δ(q, 1, 1)

= {(q, ε)}

= {(q, ε)}

9. P.T. If L is a regular language, so is L^R (6m)(Dec-Jan 2011) (Jun-Jul12)

Assume L is defined by a regular expression E.

We show that there is another regular expression ER such that

L(E

R) = (L(E))

R

that is, the language of ER is the reversal of the language of E.

Basis: If E is ǫ, ∅ of a, then ER = E.

Induction: There are three cases, depending on the form of E

10. . If L is a regular language over alphabet 6, and h is a homomorphism on 6, then h (L) is also regular. (10m)(Dec-Jan 2012).

Let *L = L(R)* for some regular expression *R.* In general, if *E* is a regular expression with symbols in E, let *h(E)* be the expression we obtain by replacing each symbol *a* of E in *E* by *h(a).* We claim that *heR)* defines the language *h(L).*

The proof is an easy structural induction that says whenever we take a subexpression *E* of *R* and apply *h* to it to get *h(E),* the language of *h(E)* is the same language we get if we apply *h* to the language *L(E).* Formally, *L(h(E») = h(L(E»).*

BASIS: If *E* is € or 0, then *h(E)* is the same as *E,* since *h* docs Hot affect the string € or the language 0. Thus, *L(h(E)) = L(E).* However, if *E* is 0 or 10, then *L(E)* contains either no strings or a string with no symbols, respectively. Thus *h(L(E») = L(E)* in either case. We conclude *L(hCE)) = L(E) = h(L(E)).*

The only other basis case is if *E* = a for some symbol *a* in !.:. In this case, *L(E) = {a},* so *h(L(E)) = {h(a)}.* Also, *h(E)* is the regular expression that is the string of symbols *h(a).* Thus, *L(h(E»)* is also *{h(a)},* and we conclude *L(h(E») = h(L(E»).*

11. Explain CGF with an example. . (5m)(Dec-Jan 2012) (Jun-Jul12)

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

Languages generated by context-free grammars are known as context-free languages.

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

12. Explain decision properties of regular language. . (5m)(Dec-Jan 2012) (Jun-Jul12)

To locate the regular languages in the Chomsky hierarchy, one notices that every regular language is context-free. The converse is not true: for example the language consisting of all strings having the same number of *a*'s as *b*'s is context-free but not regular. To prove that a language such as this is not regular, one often uses the Myhill–Nerode theorem or the pumping lemma among other methods.[5]

There are two purely algebraic approaches to define regular languages. If:

- Σ is a finite alphabet,
- Σ* denotes the free monoid over Σ consisting of all strings over Σ,
- $f : \Sigma^* \to M$ is a monoid homomorphism where M is a *finite* monoid,
- S is a subset of M

then the set $\{w \in \Sigma^* \mid f(w) \in S\}$ is regular. Every regular language arises in this fashion.

If L is any subset of Σ*, one defines an equivalence relation ~ (called the syntactic relation) on Σ* as follows: $u \sim v$ is defined to mean

$uw \in L$ if and only if $vw \in L$ for all $w \in \Sigma^*$

The language L is regular if and only if the number of equivalence classes of ~ is finite (A proof of this is provided in the article on the syntactic monoid). When a language is regular, then the number of equivalence classes is equal to the number of states of the minimal deterministic finite automaton accepting L.

A similar set of statements can be formulated for a monoid $M \subset \Sigma^*$. In this case, equivalence over M leads to the concept of a recognizable language.

UNIT 5
Pushdown Automata

1. . Give leftmost and rightmost derivations of the following strings

a) 00101

Leftmost	Rightmost
S	S
$A1B$	$A1B$
$0A1B$	$A10B$
$00A1B$	$A101B$
$00\varepsilon 1B$	$A101\varepsilon$
$0010B$	$0A101$
$00101B$	$00A101$
00101ε	$00\varepsilon 101$
00101	00101

b) 1001

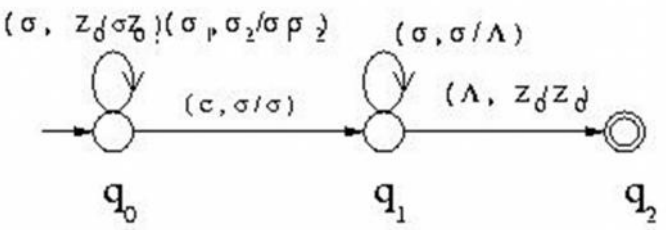
Leftmost	Rightmost
S	S
$A1B$	$A1B$
$\varepsilon 1B$	$A10B$
$10B$	$A100B$
$100B$	$A1001B$
$1001B$	$A1001\varepsilon$
1001ε	$\varepsilon 1001$
1001	1001

c) 00011 (4m)(June-July 2010) (Jun-Jul12)

Leftmost	Rightmost
S	S
$A1B$	$A1B$
$0A1B$	$A11B$
$00A1B$	$A11\varepsilon$
$000A1B$	$0A11$
$000\varepsilon 1B$	$00A11$
$00011B$	$000A11$
00011ε	$000\varepsilon 11$
00011	00011

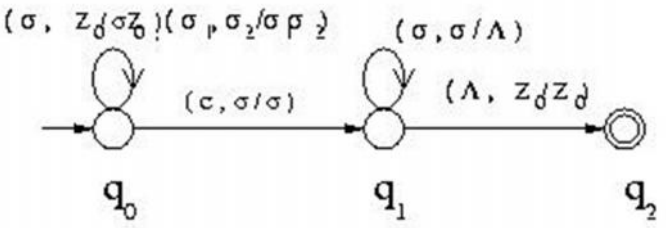
2. Construct PDA: For the language (4m)(June-July2010)

$L = \{wcw^R \mid w \quad \{a, b\}^*, c \quad \Sigma \}$



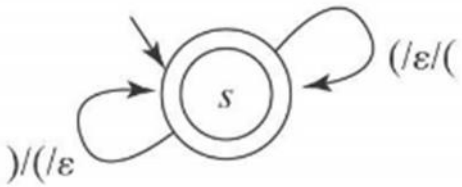
PDA accepting wcw^r

3. Construct DPDA which accepts the language L = {wcw^R | w {a, b}*, c Σ}. (4m)(June-July 2010)



PDA accepting wcw^r

4. Construct DPDA for the following: (8m)(June-July 2010) (Jun-Jul12)
Accepting the language of balanced parentheses. (Consider any type of parentheses)



$M = (K, \Sigma, \Gamma, \Delta, s, A)$, where:

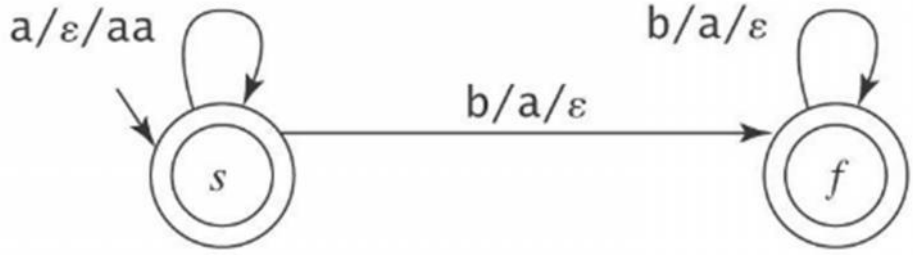
$K = \{s\}$	the states
$\Sigma = \{(,)\}$	the input alphabet
$\Gamma = \{(\}$	the stack alphabet
$A = \{s\}$	

Δ contains:

$((s, (, \varepsilon), (s, ( ))$

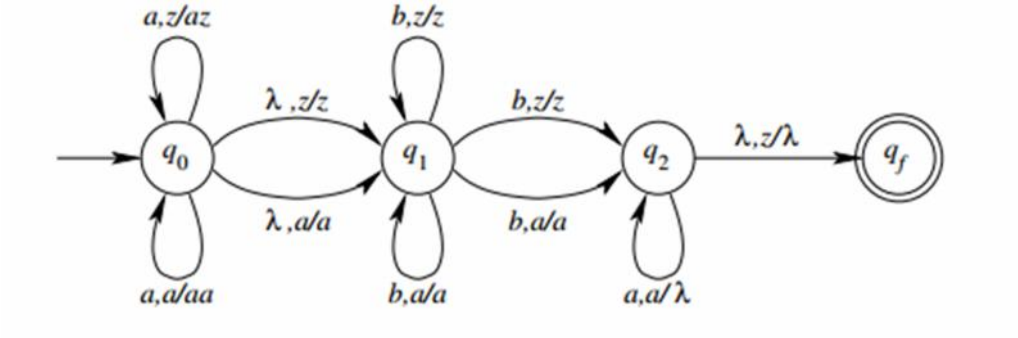
$((s,), ( ), (s, \varepsilon))$

Accepting strings with number of a's is more than number of b's
Accepting {0n1m| n t m}

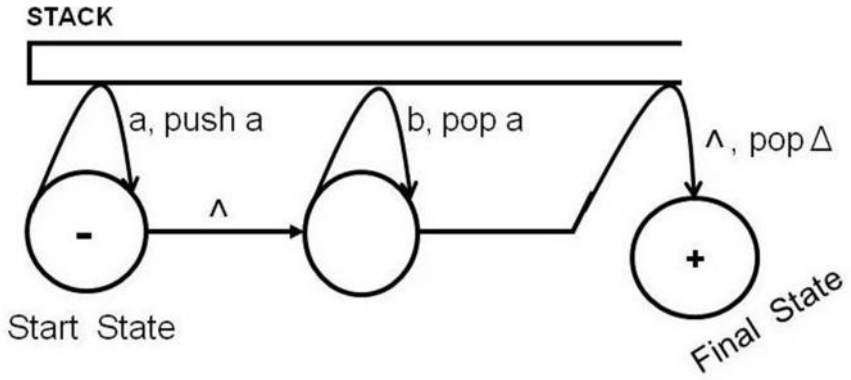


5. Design nPDA to accept the language: (10m)(Dec-Jan 2010)

$\{a^i b^j c^k \mid i, j, k \geq 0$ and $i = j$ or $i = k\}$
$\{a^i b^j c^{i+j} \mid i, j \geq 0\}$
$\{a^i b^{i+j} c^j \mid i \geq 0, j \geq 1\}$



6. Construct PDA: For the language L = {anb2n | a, b Σ, n t 0}(5m)(Dec-Jan 2010)

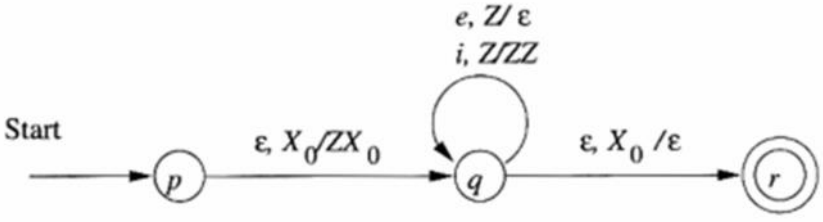


7. Construct PDA to accept if-else of a C program and convert it to CFG. (This does not accept if –if –else-else statements) (5m)(Dec-Jan 2010)

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

where δ_N is defined by:

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. This rule pushes a Z when we see an i.
2. $\delta_N(q, e, Z) = \{(q, \epsilon)\}$. This rule pops a Z when we see an e.



8. Show that deviation for the string aab is ambiguous. (5m)(June-July 2011)

Let P = (Q, Σ, Γ, δ, q0, Z0) be a PDA. An equivalent CFG is G = (V, Σ, R, S), where

V = {S, [pXq]}, where p, q ∈ Q and X ∈ Γ, productions of R consists of

1. For all states p, G has productions S → [q0Z0 p]
2. Let δ(q,a,X) = {(r, Y1Y2…Yk)} where

a ∈ Σ or

a = ε, k can be 0 or any number and r1r2 …rk are list of states. G has productions

9. Suppose h is the homomorphism from the alphabet {0,1,2} to the alphabet { a,b} defined by h(0) = a; h(1) = ab & h(2) = ba

a) What is h(0120) ?

b) What is h(21120) ?

c) If L is the language L(01*2), what is h(L) ?

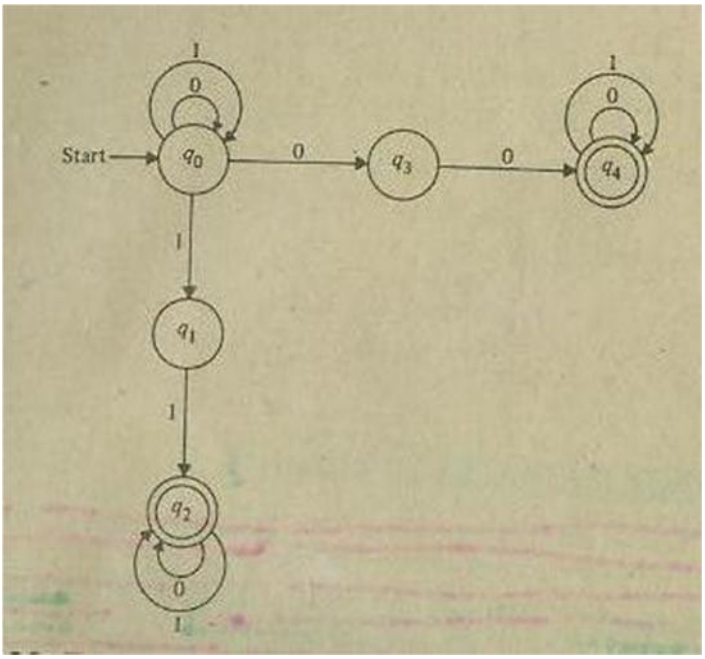
d) If L is the language L(0+12), what is h(L) ?

If L is the language L(a(ba)*) , what is h-1(L) ? (5m)(June-July 2011)

Formally, if h is a homomorphism on alphabet L:, and w = a) a2 ... an is a string of symbols in 2:, then hew) = h(al)h(a2)'" h(an). That is, we apply h to each symbol of wand concatenate the results, in order. For in- stance, if h is the homomorphism in Example 4.13, and 'W = 0011, then hew) = h(O)h(O)h(l)h(l) = (ab)(ab)(€)(e) = abab, as we claimed in that example.

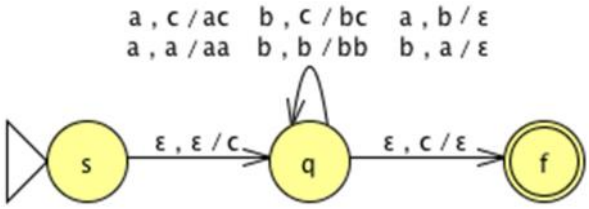
Further, we can apply a homomorphism to a language by applying it to each of the strings in the language. That is, if L is a language over alphabet 2:, and h is a homomorphism on E, then h(L) = {hew) I w is in L}. For instance, if L is the language of regular expression 10*1, i.e., any number of a's surrounded by single l's, then h(L) is the language (ab)". The reason is that h of Example 4.13 effectively drops the 1's, since they are replaced by €, and turns each a into abo The same idea, applying the homomorphism directly to the regular expression, can be used to prove that the regular languages are closed under homomorphisms.

10. Design a PDA to accept the set of all strings of 0's and 1's such that no prefix has more 1's than 0's. (5m)(June-July2011)



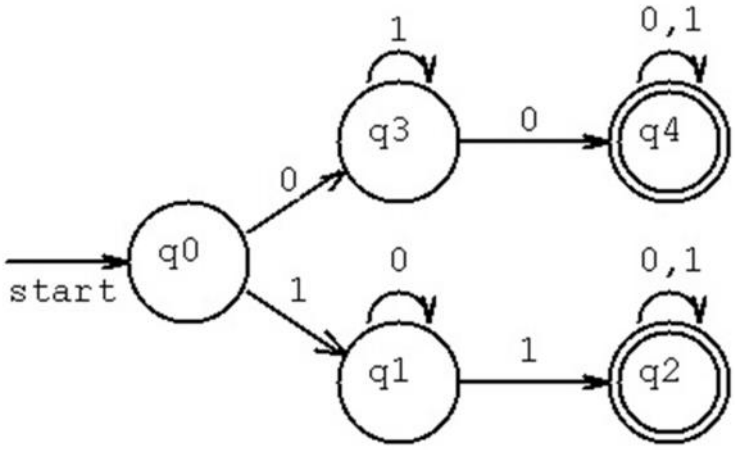
11. Construct PDA: Accepting the set of all strings over {a, b} with equal number of a's and b's. Show the moves for abbaba. (5m)(June-July2011)

The language is L = {w ∈ {a,b}*: #a(w) = #b(w) }. Here is the PDA:



12. Construct PDA: Accepting the language of balanced parentheses, (consider any type of parentheses). (5m)(Dec-Jan 2011) (Jun-Jul12)

13. Construct PDA to accept by final state the language of all strings of 0's and 1's such that number of 1's is less than number of 0's. Also convert the PDA to accept by empty stack. (5m)(Dec-Jan 2011)



14. How do you convert From PDA to CFG. (5m)(Dec-Jan 2011)

Let P = (Q, Σ, Γ, δ, q0, Z0) be a PDA. An equivalent CFG is G = (V, Σ, R, S), where V = {S, [pXq]}, where p, q ∈ Q and X ∈ Γ, productions of R consists of

1. For all states p, G has productions S → [q0Z0 p]

2. Let δ(q,a,X) = {(r, Y1Y2…Yk)} where

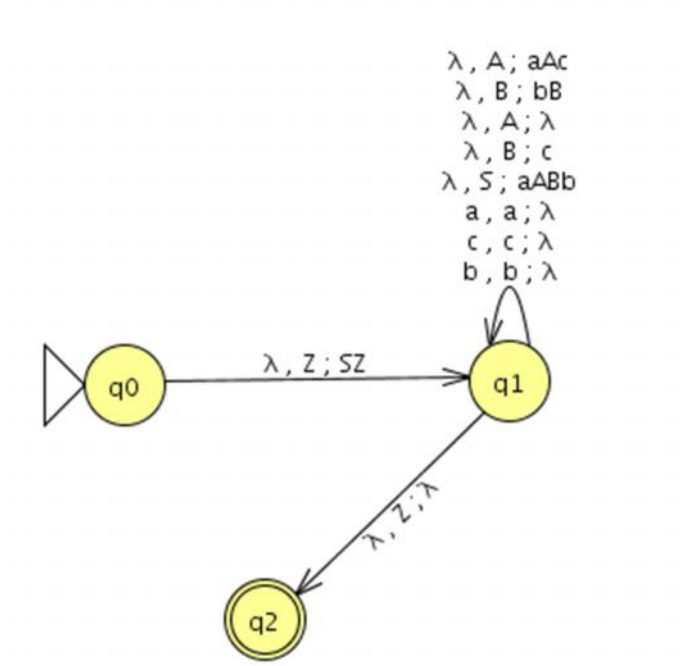
a ∈ Σ or

a = ε, k can be 0 or any number

and r1r2 …rk are list of states. G has productions

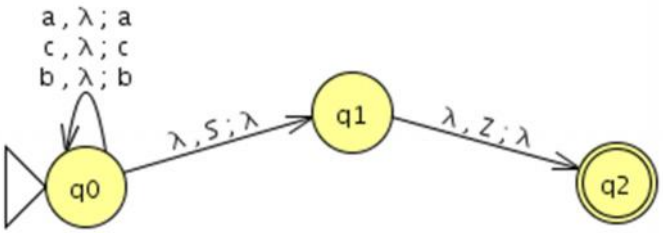
15. Convert PDA to CFG. PDA is given by P = ({p,q}, {0,1}, {X,Z}, δ, q, Z)), Transition function δ is defined by (5m)(Dec-Jan 2011)

δ(q, 1, Z) = {(q, XZ)}
δ(q, 1, X) = {(q, XX)}
δ(q, H, X) = {(q, H)}
δ(q, 0, X) = {(p, X)}
δ(p, 1, X) = {(p, H)}



16. Convert to PDA, CFG with productions (10m)(Dec-Jan 2012) (Jun-Jul12)

A o aAA, A -> aS | bS | a
S -> SS | (S) | H
S -> aAS | bAB | aB,
A -> bBB | aS | a,
B -> bA | a



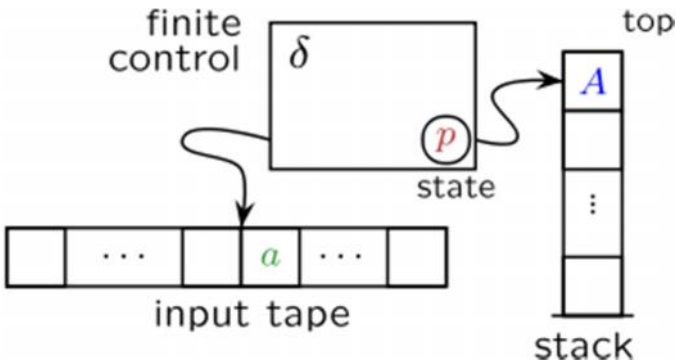
17. Explain push down automata with an example(10m)(Dec-Jan 2012)

Pushdown automata differ from finite state machines in two ways:

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

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

Pushdown automata can also manipulate the stack, as part of performing a transition. Finite state machines choose a new state, the result of following the transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack, and leave it as it is. The choice of manipulation (or no manipulation) is determined by the transition table.



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

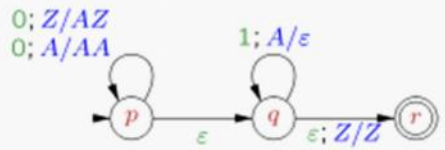
In general, pushdown automata may have several computations on a given input string, some of which may be halting in accepting configurations. If only one computation exists for all accepted strings, the result is a deterministic pushdown automaton (DPDA) and the language of these strings is a deterministic

context-free language. Not all context-free languages are deterministic. As a consequence of the above the DPDA is a strictly weaker variant of the PDA and there exists no algorithm for converting a PDA to an equivalent DPDA, if such a DPDA exists.

If we allow a finite automaton access to two stacks instead of just one, we obtain a more powerful device,

equivalent in power to a Turing machine. A linear bounded automaton is a device which is more powerful than a pushdown automaton but less so than a Turing machine.

The following is the formal description of the PDA which recognizes the language $\{0^n1^n \mid n \geq 0\}$ by final state:



PDA for $\{0^n1^n \mid n \geq 0\}$ (by final state)

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

$Q = \{p, q, r\}$

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

$\Gamma = \{A, Z\}$

$q_0 = p$

$F = \{r\}$

δ consists of the following six instructions:

$(p, 0, Z, p, AZ)$, $(p, 0, A, p, AA)$, (p, ϵ, Z, q, Z), (p, ϵ, A, q, A), $(q, 1, A, q, \epsilon)$, and (q, ϵ, Z, r, Z).

In words, in state p for each symbol 0 read, one A is pushed onto the stack. Pushing symbol A on top of another A is formalized as replacing top A by AA. In state q for each symbol 1 read one A is popped. At any moment the automaton may move from state p to state q, while it may move from state q to accepting state r only when the stack consists of a single Z.

There seems to be no generally used representation for PDA. Here we have depicted the instruction (p, a, A, q, α) by an edge from state p to state q labelled by $a; A/\alpha$ (read a; replace A by α).

UNIT 6

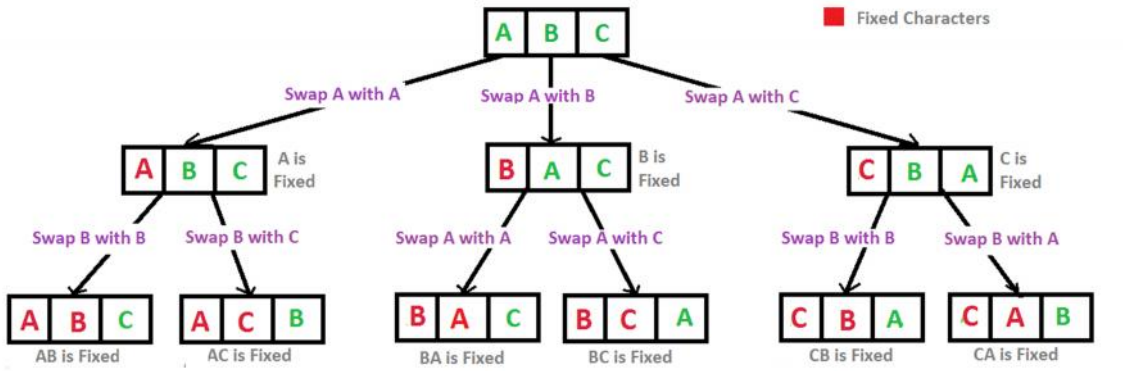
Properties of Context-Free Languages

1. Eliminate the n-on-generating symbols fr-om S -> aS | A | C, A ->a, B -> aa, C -> aCb. (8m)(June-July 2010) (Jun-Jul12)

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation. Source: Mathword(http://mathworld.wolfram.com/Permutation.html)

Below are the permutations of string ABC.

ABC, ACB, BAC, BCA, CAB, CBA



Recursion Tree for Permutations of String "ABC"

Here is a solution using backtracking.

2. Draw the dependency graph as given above. A is non-reachable from S. After eliminating A, G1= ({S}, {a}, {S -> a}, S). (6m)(June-July 2010)

We will consider CFL without. It would be easy to add to any grammar by adding a new start symbol S0, S0 ! S j Denition: A production of the form A ! Ax, A2V, x2(V [T)————————————— is left recursive. Example Previous expression grammar was left recursive. E ! E+T j T

T ! T————————————F j F

F ! I j (E)

I ! a j b A top-down parser would want to derive the leftmost terminal as soon as possible. But in the left recursive grammar above, in order to derive a sentential form that has the leftmost terminal, we have to derive a sentential form that has other terminals in it. Derivation of a+b+a+a is: E) E+T) E+T+T) E+T+T+T ————————————) a+T+T+T We will eliminate the left recursion so that we can derive a sentential form with the leftmost terminal and no other terminals.

3. Find out the grammar without H – Productions G = ({S, A, B, D}, {a}, {S o aS | AB, A -> H, B-> H, D ->b}, S). (6m)(June-July 2010)

Consider all possible four-step derivations. Toss out duplicates at any intermediate point. Also remove from consideration strings such as AAAA which cannot be instantiated (replaced with terminals)

in x (in this case three) or fewer steps. The number of nonterminals at each step cannot exceed the number of steps left in the derivation.

a) S -> AA -> aA -> aa
S -> AA -> aA -> abA -> aba
S -> AA -> aA -> aAb -> aab
S -> AA -> Aa -> aa (delete)
S -> AA -> Aa -> bAa -> baa
S -> AA -> Aa -> Aba -> aba (delete)
S -> AA -> bAA -> baA -> baa (delete)
S -> AA -> bAA -> bAa (delete)
S -> AA -> AbA -> abA (delete)
S -> AA -> AbA -> Aba (delete)
S -> AA -> AAb -> aAb (delete)
S -> AA -> AAb -> Aab -> aab (delete)

{aa, aba, aab, baa}

b) S -> AA -> bAA -> baA -> babA -> babbA -> babbAb -> babbab
S -> AA -> bAA -> baA -> baAb -> babAb -> babbAb -> babbab
S -> AA -> AAb -> bAAb -> baAb -> babAb -> babbAb -> babbab
S -> AA -> AAb -> bAAb -> bAab -> bAbab -> bAbbab -> babba

4. Eliminate non-reachable symbols from G= ({S, A}, {a}, {S -> a, A ->a}, S) (10m)(Dec-Jan

- A symbol X is *useful* for a grammar $G = (V, T, P, S)$, if there is a derivation

$$S \overset{*}{\underset{G}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G}{\Rightarrow}} w$$

for a teminal string w. Symbols that are not useful are called *useless*.

- A symbol X is *generating* if $X \overset{*}{\underset{G}{\Rightarrow}} w$, for some $w \in T^*$

- A symbol X is *reachable* if $S \overset{*}{\underset{G}{\Rightarrow}} \alpha X \beta$, for some $\{\alpha, \beta\} \subseteq (V \cup T)^*$

It turns out that if we eliminate non-generating symbols first, and then non-reachable ones, we will be left with only useful symbols.

10)

5. Eliminate non-reachable symbols from S -> aS | A, A -> a, B -> aa. (10m)(Dec-Jan 10)

Mark a variable X as "generating" if it has a production X -> w where w is a string of only terminals and/or variables previously marked "generating".
Repeat the step above until no further variables get marked "generating".
All variables not marked "generating" are non-generating (by a simple induction on the length of derivations).
Call a variable reachable if the start symbol derives a string containing that variable. Here is an algorithm to find the reachable variables in a CFG:
Mark the start variable as "reachable".
Mark a variable Y as "reachable" if there is a production X -> w where X is a variable previously marked as "reachable" and w is a string containing Y.
Repeat the step above until no further variables get marked "reachable".
All variables not marked "reachable" are non-reachable (by a simple induction on the length of derivations).
Observe that a CFG has no useless variables if and only if all its variables are reachable and generating.
Therefore it is possible to eliminate useless variables from a grammar as follows:
Find the non-generating variables and delete them, along with all productions involving non-generating variables.
Find the non-reachable variables in the resulting grammar and delete them, along with all productions involving non-reachable variables.
Note that step 1 does not make other variables non-generating, and step 2 does not make other variables non-reachable or non-generating. Therefore the end result is a grammar in which all variables are reachable and generating, and hence useful.
Reversing step 1 and 2 in the above algorithm would not work, as eliminating non-generating variables and their productions may make other variables unreachable. Example:
S -> AB | a
A -> aA
B -> b

Here A is non-generating, and after deleting A (along with the production S -> AB) the variable B becomes unreachable. So it must be a useless variable. However, if we would first test for reachability, all variables would be reachable, and subsequently eliminating non-generating variables would leave us with B.

6. Eliminate useless symbols from the grammar with productions S -> AB | CA, B ->BC | AB, A ->a, C -> AB | b. (5m)(June-July 2011) (Jun-Jul12)

7. Eliminate useless symbols from the grammar (5m)(June-July 2011)

P= {S o aAa, A ->Sb | bCC, C ->abb, E -> aC}

P= {S -> aBa | BC, A -> aC | BCC, C ->a, B -> bcc, D -> E, E ->d}

P= {S -> aAa, A -> bBB, B -> ab, C -> aB}
P= {S -> aS | AB, A -> bA, B -> AA}.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a levelof "binding strength." Specifically:
1. A *[actor* is an expression that cannot be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
(a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
(b) Any parenthesised expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent wha.t is inside from becoming the operand of any operator outside the parentheses.
2. A *term* is an expression that cannot be broken by the + operator. In our example, where + and * are the *only* operators, a term is a product of one or more factors. For instance, the term *a * b* can be "broken" if we use left associativity and place ah to its left. That is, *a1 *a * b* is grouped *(al * a) * b,* which breaks apart the *a * b.* However, placing an additive term, such as al+, to its left or *+al* to its right cannot break *a * b.* The proper grouping of *al + a * b* is *al + (a * b),* and the proper grouping of a * *b* +al is (a * *b)* +al.
3. An *expression* will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression for our example is a sum of one or more terms.
I -+
F -+
T -+
E -+
a I *b* I *I a* I *[b* I [0 I Il
[I (E)
*FIT*F*
TI *E+T*

8. Write Algorithm to find nullable variables. (5m)(June-July 2011)

If we have a production like A ! BCDE, we can introduce some new variables that allow the variables of the body to be introduced one at a time.
A body of length k requires k 2 new variables.
Example: Introduce F and G; replace A ! BCDE by A ! BF ; F ! CG; G ! DE.
Summary Theorem
If L is any CFL, there is a grammar G that generates L fg, for which each production is of the form A ! BC or A ! a, and there are no useless symbols. CFL Pumping Lemma
Similar to regular-language PL, but you have to pump two strings in the middle of the string, in tandem (i.e., the same number of copies of each). Formally:
8 CFL L 9 integer n
8 z in L, with jzj n
9 uvwxy = z such that jvwxj n and jvxj > 0
8 i 0, uviwxiy is in L.
Outline of Proof of PL
Let there be a Chomsky-normal-form CFG for L with m variables. Pick n = 2m
.
Because CNF grammars have bodies of no more than 2 symbols, a string z of length n must have some path with at least m + 1 variables.

Thus, some variable must appear twice on the path.

✦ Compare with the DFA argument about a path longer than the number of states.

9. Eliminate H - productions from the grammar. (5m)(June-July 2011)

S -> a |Xb | aYa, X -> Y| H, Y -> b | X

S -> Xa, X -> aX | bX | H

S -> XY, X ->Zb, Y -> bW, Z ->AB, W ->Z, A -> aA | bB | H, B -> Ba | Bb| H

S -> ASB | H, A -> aAS | a, B -> SbS | A| bb

Suppose h applies to symbols of alphabet :E 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, I', 6,qo, Zo, F) that accepts L by final state.

We construct a new PDA where:

P' = (Q',:E,6',(qo,€),Zo,F X {€})

1. Q' is the set of pairs (q, z) such that:

(a) q is a state in Q, and PDA I---If---I- state Stack Accept!reject

(b) x is a suffix (not necessarily proper) of some string h(a) for some input symbol a in :E.

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 period- ically 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 'E is finite, and h(a) is finite for all a, there are only a finite number of states for P'.

2. 6' is defined by the followingrules: (a) c5' (q, e), a, X) = {((q, h(a)), X)} for all symbols a in 'E, all states q in Q, and stack symbols X in r. Note that a cannot be e here. When the buffer is empty, P' can consume its next input symbol a and place h(a) in the buffer.

(b) If 6(q, b, X) contains (p, 1'), where b is in T or b = e, then contains (P, x), 1'). 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 = E, then the buffer can be empty.

3. Note that, as defined in (7.1), the start state of P' is (qO,e)j 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, €) such that q is an accepting state of P. The followingstatement characterizes the relationship between P' and P:

• (qo,h(w),Zo) ~ (P,E,'Y) if and only if (qO'€)'w,Zo)~, (p,E),e,'Y).

10. Eliminate H - pr->ductions and useless symbols from the grammar S ->a |aA|B|C, A ->aB| H, B ->aA, C ->aCD, D ->dd. (10m)(Dec-Jan-2011)

A string y is said to be a permutation of the string x if the symbols of y can be reordered to make x. For instance, the permutations of string x = 011 are 110, WI, and 011. If L is a language, then perm(L) is the set of strings that are permutations of strings in L. For example, if

L = {onl'l I n ~ OJ, then perm(L) is the set of strings with equal numbers of

O's and L's,

a) Give an example of a regular language L over alphabet {O,I} such that perm(L) is not regular. Justify your answer. Hint: Tty to find a regular language whose permutations are all strings with an equal number of O's and l's.

b) Give an example of a regular language L over alphabet {O,1,2} such that perm(L) is not context-free.

c) Prove that for every regular language L over a two-symbol alphabet, perm(L) is context-free

11. Show that L = {$a^i b^i c^i$ | i ŧ1} is not CFL. (10m)(Dec-Jan 2011)

BASIS: We compute the first row as follows. Since the string beginning and ending at position i is just the terminal ~, and the grammar is in CNF, the only way to derive the string ai is to use a production of the form A ~ ai. Thus, Xii is the set of variables A such that A -t ai is a production of G.

INDUCTION: Suppose we want to compute Xij, which is in row j - i +1, and we have computed all the X's in the rows below. That is, we know about all strings shorter than aiaj,+l... aj, and in particular we know about all proper prefixes and proper suffixesof that string. As j - i > 0 may be assumed (since the case i = j is the basis), weknowthat any derivation A ~ aiai+1 . ,. aj must start out with some step A ::::}BC. Then, B derives some prefix of aiai+l ... aj, say B ~ a'iaHl'" ak, for some k < j. Also, C must then derive the remainder of aiai+l ... aj, that is, C ~ ak+lak+2'" aj. We conclude that in order for A to be in Xij, we must find variables B and C, and integer k such that:

1. i 5: k < j.
2. B is in x.;
3. C is in Xk+l,j'
4. A -7 BO is a production of G**.**

12. Show that L = {ww |w {0, 1}*} is not CFL. (10m)(Dec-Jan 2012)

13. Using pumping lemma for CFL prove that below languages are not context free

{p | p is a prime}. . (10m)(Dec-Jan 2012)

To construct the first (lowest) rowI we use the basis rule. We have only to consider which variables have a production body a (those variables are A and C) and which variables have body b (only B does). Thus, above those positions holding a we see the entry {A, C}, and above the positions holding b we sec {B}. That is,Xll = X44 = {B}, and X22 = X33 = Xss = {A, C}.

In the second row we see the values of X12, X23, X34, and X4S' For instance, let us see how Xl2 is computed. There is only one way to break the string from positions 1 to 2, which is ba, into two nonempty substrings. The first must be position 1and the second must be position 2. In order for a variable to generate ba, it must have a body whose first variable is in Xll = {B} (i.e., it generates the b) and whose second variable is in X22 = {A, C} (i.e., it generates the a). This body can only be BA or BC. ITwe inspect the grammar, we find that the productions A ~ BA and S ~ BC are the only ones with these bodies. Thus, the two heads, A and S, constitute X12.

For a more complex example, consider the computation of X24. We can break the string aab that occupies positions 2 through 4 by ending the first string after position 2 or position 3. That is, we may choose k = 2 or k = 3 in the definition of X24. Thus, we must consider all bodies in X22X34 UX23X44. This set of strings is {A, C}{S,C} U {B}{B} = {AS, AC, CS,CC, BB}.

UNIT 7

Introduction To Turing Machine

1. Explain with example problems that Computers cannot solve.(6m)(June-July2010)

The purpose of this section is to provide an informal, C-programming-based introduction to the proof of a specific problem that computers cannot solve.

The particular problem we discuss is whether the first thing a C program prints is hello, world. Although we might imagine that simulation of the program would allow us to tell what the program does, we must in reality contend with programs that take an unimaginably long time before making any output at all.

This problem - not knowing when, if ever, something will occur - is the ultimate cause of our inability to tell what a program does. However, proving formally that there is no program to do a stated task is quite tricky, and we need to develop some formal mechanics. In this section, we give the intuition behind the formal proofs.

2. Explain briefly the following Halting problem. . (4m)(June-July2010)

One often hears of the halting problem for Turing machines as a problem similar to Lu - one that is RE but not recursive. In fact, the original Turing machine of A. M. Turing accepted by halting, not by final state.

We could define H(M) for TM M to be the set of inputs w such that M halts given input w, regardless of whether or not M accepts w. Then, the halting problem is the set of pairs (M, w) such that tv is in H(M).

This problem/language is another example of one that is RE but not recursive

3. Explain Programming techniques for Turning Machines(10m)(Dec-Jan-2010) (Jun-Jul12)

The Turing machine mathematically models a machine that mechanically operates on a tape. On this tape are symbols, which the machine can read and write, one at a time, using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On computable numbers, with an application to the Entscheidungsproblem", see also references below), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").

The head is always over a particular square of the tape; only a finite stretch of squares is shown. The

instruction to be performed (q4) is shown over the scanned square. (Drawing after Kleene (1952) p.375.)

Here, the internal state (q1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its complete configuration) consists of the internal state, any non-blank symbols on the tape (in this illustration "11B"), and the position of the head relative to those symbols including blanks, i.e. "011B". (Drawing after Minsky (1967) p. 121).

More precisely, a Turing machine consists of:

A tape divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special blank symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.

A head that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.

A state register that stores the state of the Turing machine, one of finitely many. There is one special start state with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.

A finite table (occasionally called an action table or transition function) of instructions (usually quintuples [5-tuples] : qiaj→qi1aj1dk, but sometimes quadruples [4-tuples]) that, given the state(qi) the machine is currently in and the symbol(aj) it is reading on the tape (symbol currently under the head) tells the machine to do the following in sequence (for the 5-tuple models):

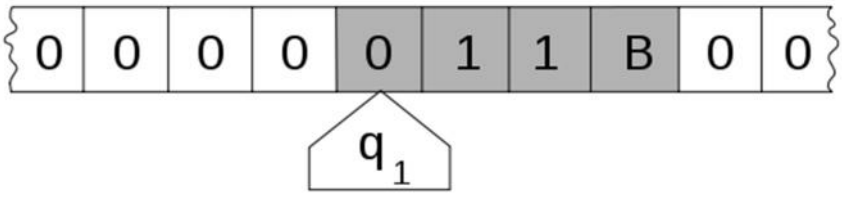
Either erase or write a symbol (replacing aj with aj1), and then

Move the head (which is described by dk and can have values: 'L' for one step left or 'R' for one step right or 'N' for staying in the same place), and then

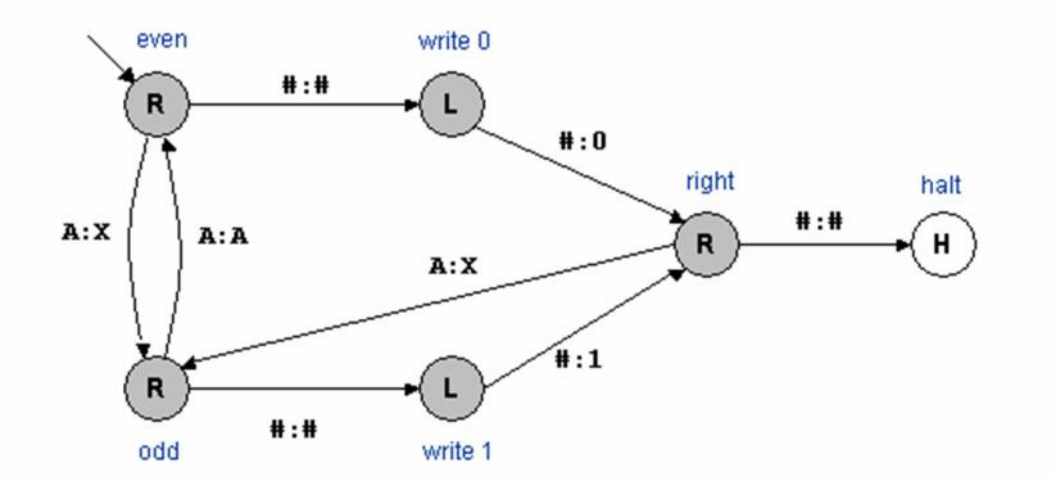
Assume the same or a new state as prescribed (go to state qi1).

In the 4-tuple models, erasing or writing a symbol (aj1) and moving the head left or right (dk) are specified as separate instructions. Specifically, the table tells the machine to (ia) erase or write a symbol or (ib) move the head left or right, and then (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

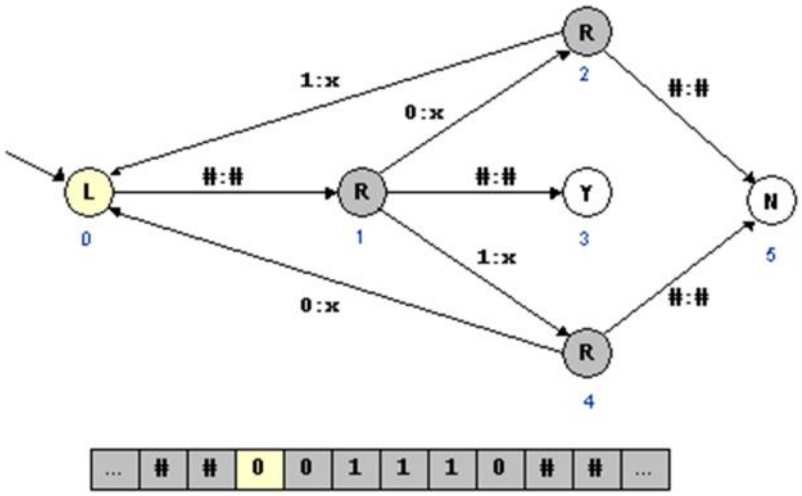
Note that every part of the machine (i.e. its state and symbol-collections) and its actions (such as printing, erasing and tape motion) is finite, discrete and distinguishable; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.



4. Design a Turing machine to accept a Palindrome. (10m)(Dec-Jan-2011)



5. Design a TM to recognize a string of the form $a^n b^{2n}$. (10m)(June-July2010)



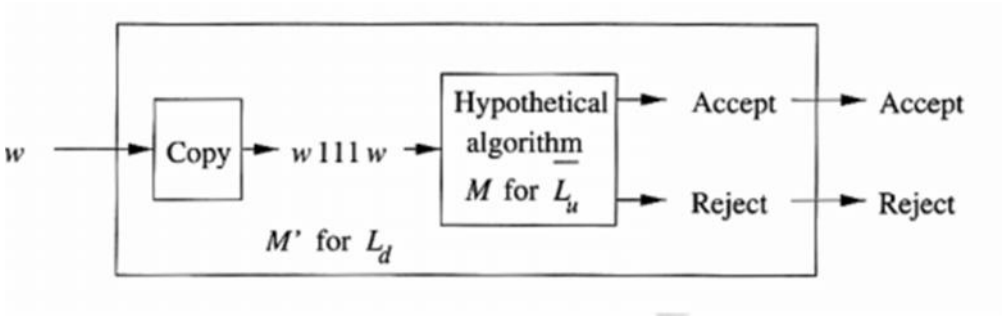
6. Design a Turing machine to accept a Palindrome. (10m)(Dec-Jan-2012)

7. Define undesirability, decidability. (10m)(June-July 2011)

We can now exhibit a problem that is RE but not recursive; it is the language Lu' Knowing that Lu is undecidable (i.e., not a recursive language) is in many ways more valuable than our previous discovery that Ld is not RE. The reason is that the reduction of L" to another problem P can be used to show there is no algorithm to solve P, regardless of whether or not P is RE. However, reduction of La to P is only possible if P is not RE, so La cannot be used to show undecidability for those problems that are RE but not recursive. On the other hand, if we want to show a problem not to be RE, then only La can be used; L11.is useless since it is RE.

Theorem 9.6: L11.is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that Lu is RE. Suppose Lu were recursive. Then by Theorem 9.3, Lu, the complement of Lu, would also be recursive. However, if we have a TM M to accept Lu, then we can construct a TM to accept La (by a method explained below). Since we already know that La is not RE, we have a contradiction of our assumption that Lu is recursive.



8. Post's Correspondence problem Design a TM to recognize a string of 0s and 1s such that the number of 0s is not twice as that of 1s. (10m)(Dec-Jan 2012)

An instance of Post's Correspondence Problem (PCP) consists of two lists of strings over some alphabet :E; the two lists must be of equal length. We generally refer to the A and B lists, and write A = WI, W2, ... ,Wk and B = Xl, X2, ... ,Xk, for some integer k. For each i, the pair (Wi, Xi) is said to be a corresponding pair.

We say this instance of PCP has a solution, if there is a sequence of one 01' more integers iI, i2, ... ,im that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, Wil Wi2 ... Wim = XiI Xi2 ... X'im . We say the sequence il, iz, ... ,im is a solution to this instance of PCP, if so.

The Post's correspondence problem is:

• Given an instance of PCP, tell whether this instance has a solution.

Example 9.13: Let :E = {O,I}, and let the A and B lists be as defined in Fig. In this case, PCP has a solution. For instance, let m = 4, il = 2, i2 = 1, i3 = 1, and i4 = 3; i.e., the solution is the list 2,1,1,3. We verify that this list is a solution by concatenating the corresponding strings in order for the two lists. That is, 'WZWIWIW3 = XZXIXjX3 = 101111110.Note this solution is not unique. For instance, 2,1,1,3,2,1,1,3 is another solution

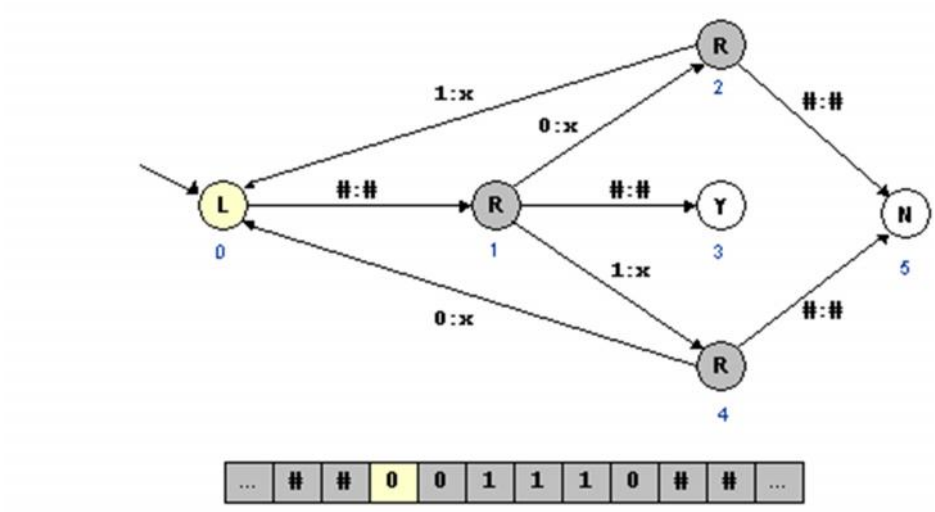
	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Figure 9.12: An instance of PCP

UNIT 8

Undecidability

1. **Design a TM to recognize a string of the form $a^n b^{2n}$. (10m)(June-July2010)**



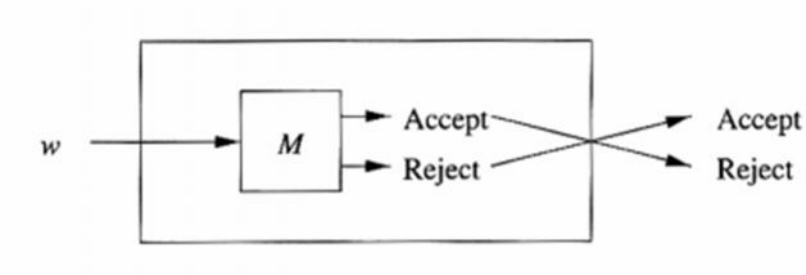
2. **P.t If L is a recursive language, L- is also recursive. (10m)(June-July2010)**

PROOF: Let L = L(M) for some TM M that always halts. We construct a TM M such that I = L(M) by the construction suggested in Fig. 9.3. That is, M behaves just like M. However, M is modified as follows to create M

1. The accepting states of M are made nonaccepting states of M with no transitions; i.e., in these states M will halt without accepting.

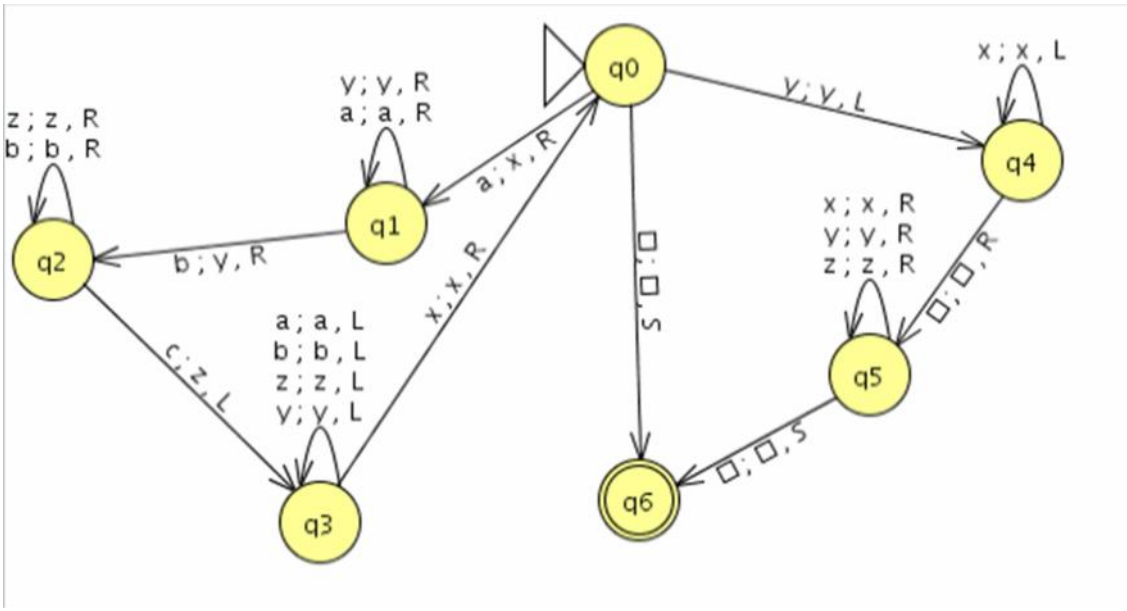
2. M has a new accepting state r; there are no transitions from r, 3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.

Since M is guaranteed to halt, we know that M is also guaranteed to halt.



Moreover, M accepts exactly those strings that M does not accept. Thus M accepts L

3. **Design a Turing Machine to recognize 0n1n2n. (10m)(Dec-Jan 2010)**



4. **Explain briefly the following Halting problem(6m)(Dec-Jan 2010)**

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e. all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program:

while (true) continue;

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

print "Hello, world!"

halts very quickly.

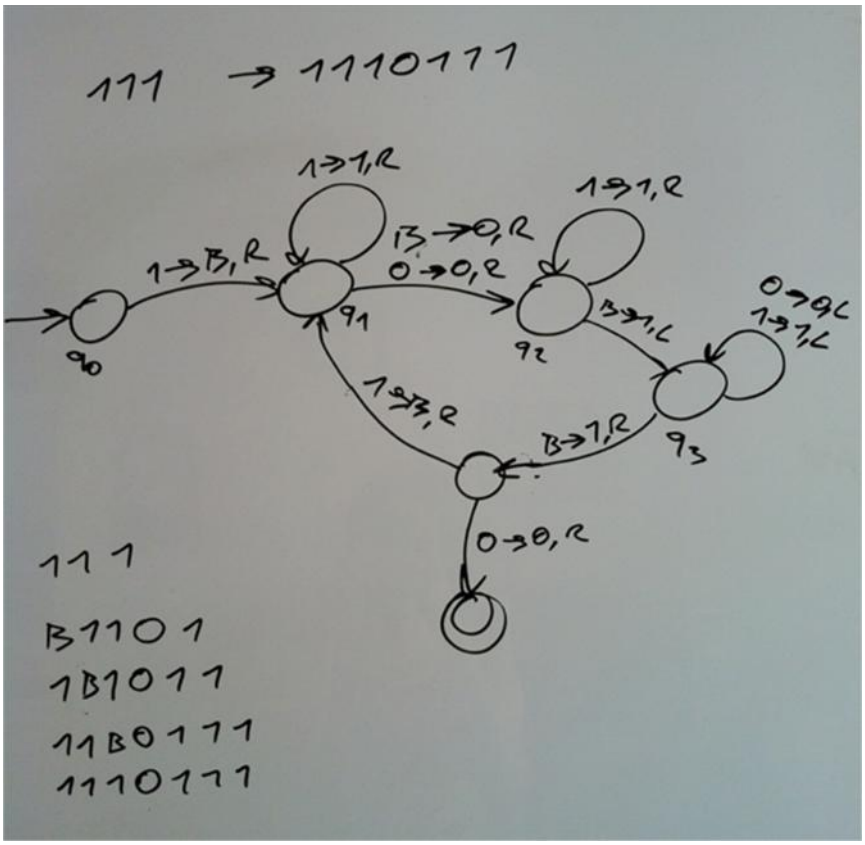
While deciding whether these programs halt is simple, more complex programs prove problematic.

One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever.

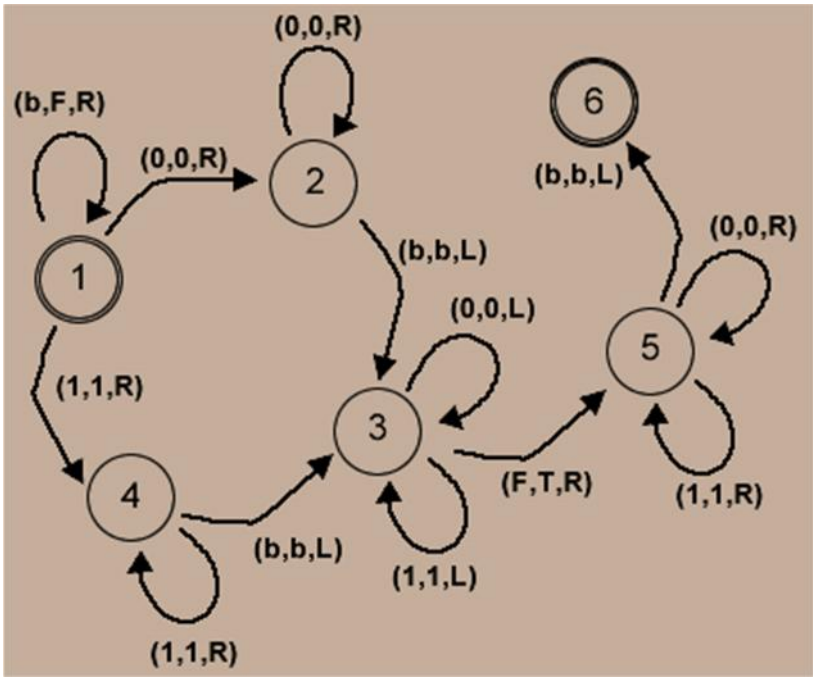
Turing proved there cannot exist an algorithm which will always correctly decide whether, for a given arbitrary program and its input, the program halts when run with that input; the essence of Turing's proof is that any such algorithm can be made to contradict itself, and therefore cannot be correct.

5. **Define undesirability, decidability. (8m)(June-July 2011)(repeated)**

6. **Post's Correspondence problem Design a TM to recognize a string of 0s and 1s such that the number of 0s is not twice as that of 1s. (10m)(Dec-Jan 2012)**



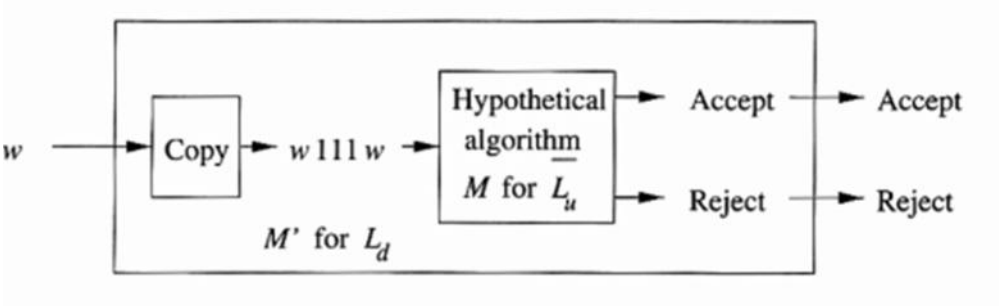
7. **Design a Turing machine to accept a Palindrome. (7m)(Dec-Jan-2011)**



8. **Write a short note on: (20m) Dec-Jan 2012)(repeated)**

a. Undesirability, decidability

We can now exhibit a problem that is RE but not recursive; it is the language Lu' Knowing that Lu is



undecidable (i.e., not a recursive language) is in many ways more valuable than our previous discovery that Ld is not RE. The reason is that the reduction of L" to another problem P can be used to show there is no algorithm to solve P, regardless of whether or not P is RE. However, reduction of La to P is only possible if P is not RE, so La cannot be used to show undecidability for those problems that are RE but not recursive. On the other hand, if we want to show a problem not to be RE, then only La can be used; L11.is useless since it is RE.

b. Theorem 9.6: L11.is RE but not recursive.

c. PROOF: We just proved in Section 9.2.3 that Lu is RE. Suppose Lu were recursive. Then by Theorem 9.3, Lu, the complement of Lu, would also be recursive. However, if we have a TM M to accept Lu,

then we can construct a TM to accept La (by a method explained below). Since we already know that La is not RE, we have a contradiction of our assumption that Lu is recursive.

b. Halting problem

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e. all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program:

while (true) continue;

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

print "Hello, world!"

halts very quickly.

While deciding whether these programs halt is simple, more complex programs prove problematic.

One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever.

Turing proved there cannot exist an algorithm which will always correctly decide whether, for a given arbitrary program and its input, the program halts when run with that input; the essence of Turing's proof is that any such algorithm can be made to contradict itself, and therefore cannot be correct.

.