

Regular Expressions and Finite State Automata

❖ Themes

- Finite State Automata (FSA)
 - Describing patterns with graphs
 - Programs that keep track of state
- Regular Expressions (RE)
 - Describing patterns with regular expressions
 - Converting regular expressions to programs

❖ Theorems

- The languages (Regular Languages) recognized by FSA and generated by RE are the same
- There are languages generated by grammars that are not Regular

Regular Expressions

❖ Describe (generate) *Regular Languages*

❖ A pattern:

➤ ϵ – the empty string

➤ a – a literal character, stands for itself

❖ Operations

➤ Concatenation, RS

➤ Alternation, R|S

➤ Closure (Kleene Star) – R^* , the set of all strings that can be made by concatenating zero or more strings in R

Regular Expressions

- ❖ In the algebra of regular expressions, an atomic operand is one of the following:
 - A character : $L(x) = \{x\}$
 - The symbol ε : $L(\varepsilon) = \{\varepsilon\}$
 - The symbol \emptyset : $L(\emptyset) = \{\}$
 - A variable whose value can be any pattern defined by a regular expression

Regular Expressions

❖ There are three operators used to build regular expressions:

➤ Union

- $R|S - L(R|S) = L(R) \cup L(S)$

➤ Concatenation

- $RS - L(RS) = \{rs, r \in R \text{ and } s \in S\}$

➤ Closure

- $R^* - L(R^*) = \{\varepsilon, R, RR, RRR, \dots\}$

RE – examples

- ❖ $a|b^*$ denotes $\{\varepsilon, a, b, bb, bbb, \dots\}$
- ❖ $(a|b)^*$ denotes the set of all strings consisting of any number of a and b symbols, including the empty string
- ❖ $b^*(ab^*)^*$ the same
- ❖ $ab^*(c|\varepsilon)$ denotes the set of strings starting with a , then zero or more b s and finally optionally a c .

Regular Expressions

❖ $a|(ab)$

❖ $(a|(ab))|(c|(bc))$

❖ a^*

❖ a^*b^*

❖ $(ab)^*$

❖ $a|bc^*d$

❖ $\text{letter} = a|b|c|\dots|z|A|B|C|\dots|Z|_$

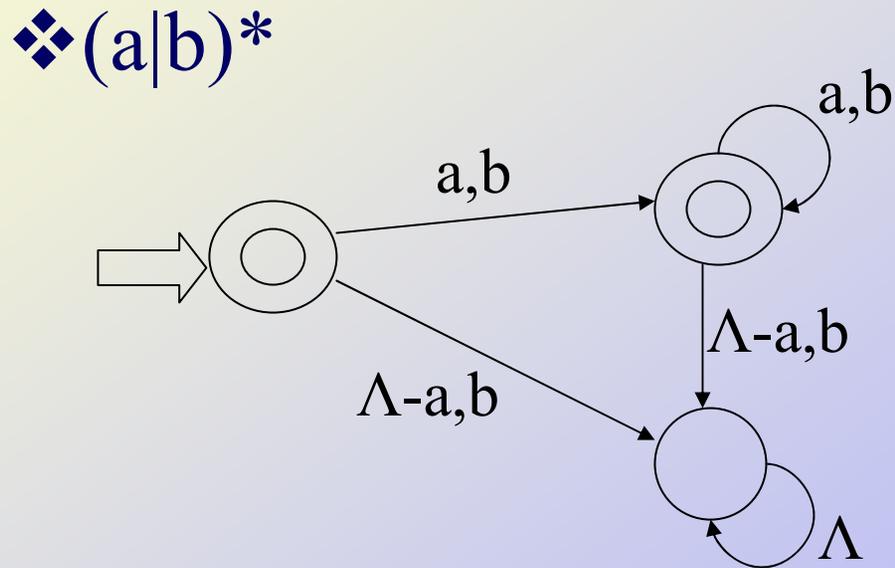
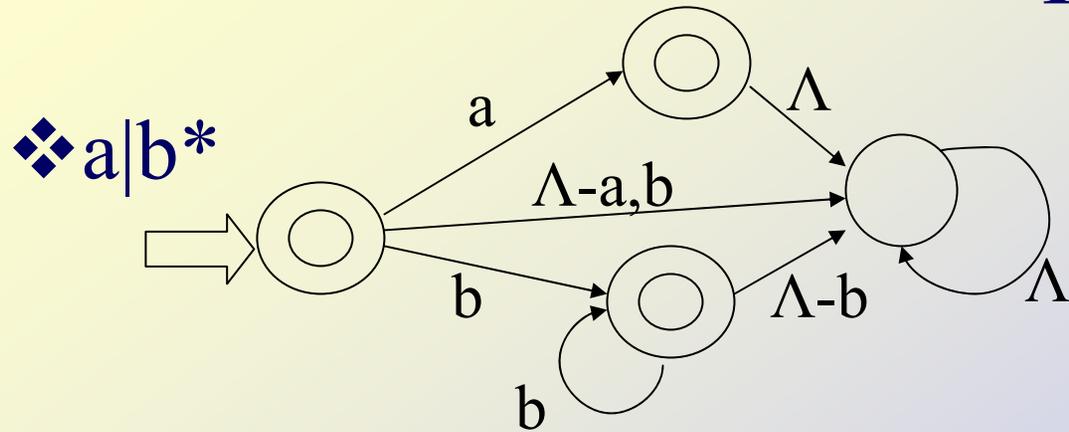
❖ $\text{digit} = 0|1|2|3|4|5|6|7|8|9$

❖ $\text{letter}(\text{letter}|\text{digit})^*$

Finite State Machine

- ❖ Accepts or rejects a string
- ❖ A finite collection of states
- ❖ Has a single *start state*
- ❖ Transition from one state to another on a given input
- ❖ Machine accepts if in an accepting state at end of input (whatever that means)

FSM - Example



Problem

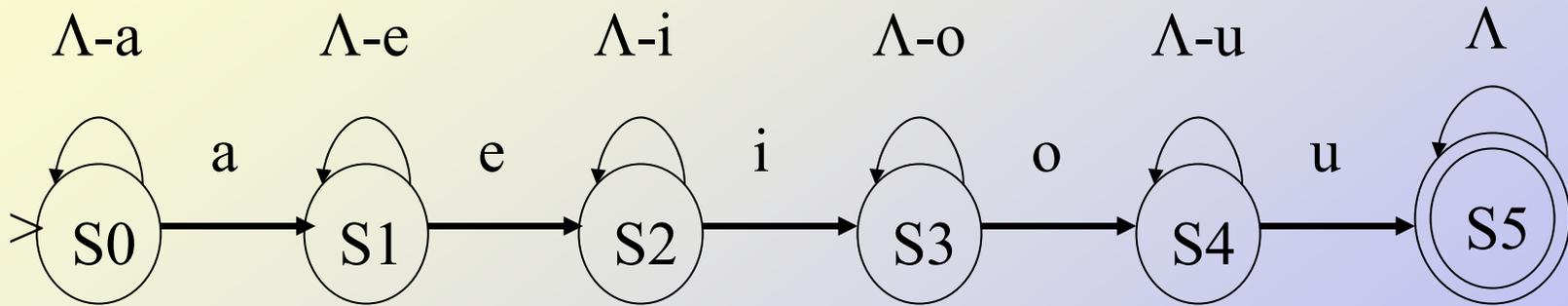
❖ Find all words that contain all of the vowels in alphabetical order*

- **ab•ste•mi•ous** *adj* : sparing in use of food or drink : temperate — **ab•ste•mi•ous•ly** *adv* — **ab•ste•mi•ous•ness** *n*

- (c)2000 Zane Publishing, Inc. and Merriam-Webster, Incorporated. All rights reserved.

*Note: this problem statement is a little ambiguous. Does **aieiaoeua** also pass?

Problem 4 Solution 1



$\Lambda = \text{set of all letters}$

Problem 4 Solution 2

```
$ grep '.*a.*e.*i.*o.*u.*' <  
  /usr/dict/words
```

adventitious

facetious

sacrilegious

Problem 5

- ❖ Partial Anagram: Find all words that can be made from the letters in Washington/



- ❖ a, ago, ah, an, angst, ...

Problem 5 Grammar

$\langle \text{Washington} \rangle \rightarrow$

$\langle w \rangle \langle a \rangle \langle s \rangle \langle h \rangle \langle i \rangle \langle n \rangle \langle g \rangle \langle t \rangle \langle o \rangle \langle n \rangle$

$\langle w \rangle \rightarrow w | \epsilon$

$\langle a \rangle \rightarrow a | \epsilon$

$\langle s \rangle \rightarrow s | \epsilon$

...

$\langle o \rangle \rightarrow o | \epsilon$

❖ Note, this only finds partial anagrams where the characters maintain their relative order

Generating Subsets

❖ Let $S = \{a,b,c\}$

➤ Review basic notions of set theory (Sec. 7.2 & 7.3)

❖ The power set of S , $P(S)$ is the set of all subsets of S – including S and the empty set

$$P(S) = \{ \{b,c\}, \{b\}, \{c\}, \{\} \\ \{a,b,c\}, \{a,b\}, \{a,c\}, \{a\},$$

Recursive Program to Generate $P(S)$

PowerSet(S)

if $S = \{\}$ return $\{\{\}\}$;

else

$S' = \text{PowerSet}(S \setminus \text{First}(S))$;

$S = S'$;

for s in S' do

$S = S \cup (\text{First}(S) \text{ union } s)$;

return S ;

Generating Permutations

- ❖ Let $S = [a,b,c]$
- ❖ The permutations of S are
 - $[a,b,c]$
 - $[a,c,b]$
 - $[b,a,c]$
 - $[b,c,a]$
 - $[c,a,b]$
 - $[c,b,a]$

Recursive Program to Generate

$\text{Perm}(L), L = [a_1, \dots, a_n]$

$S = \text{Perm}(L)$

if $\text{Length}(L) = 1$ return $\{L\}$;

else

for a in L do

$S' = \text{Perm}(L/a)$; // delete a from L

for s in S' do

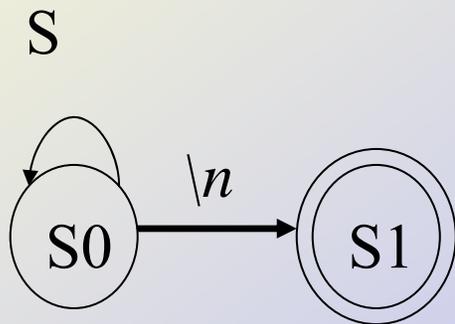
$S = S \cup [a,s]$;

Alternate Approach

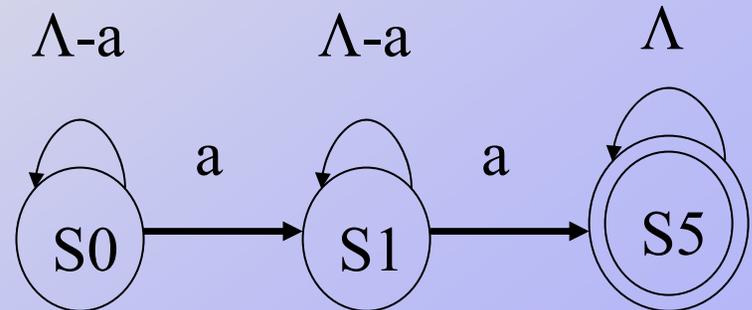
- ❖ Instead of generating all possibilities and checking the result to see if it is a word, check each word to see if it is a partial anagram.
- ❖ To check a word
 - see if it has the right letters
 - make sure each letter occurs an allowable number of times

Problem 5 - Solution 1

$$\diamond S = \{a, g, h, i, n, o, s, t, w\}$$

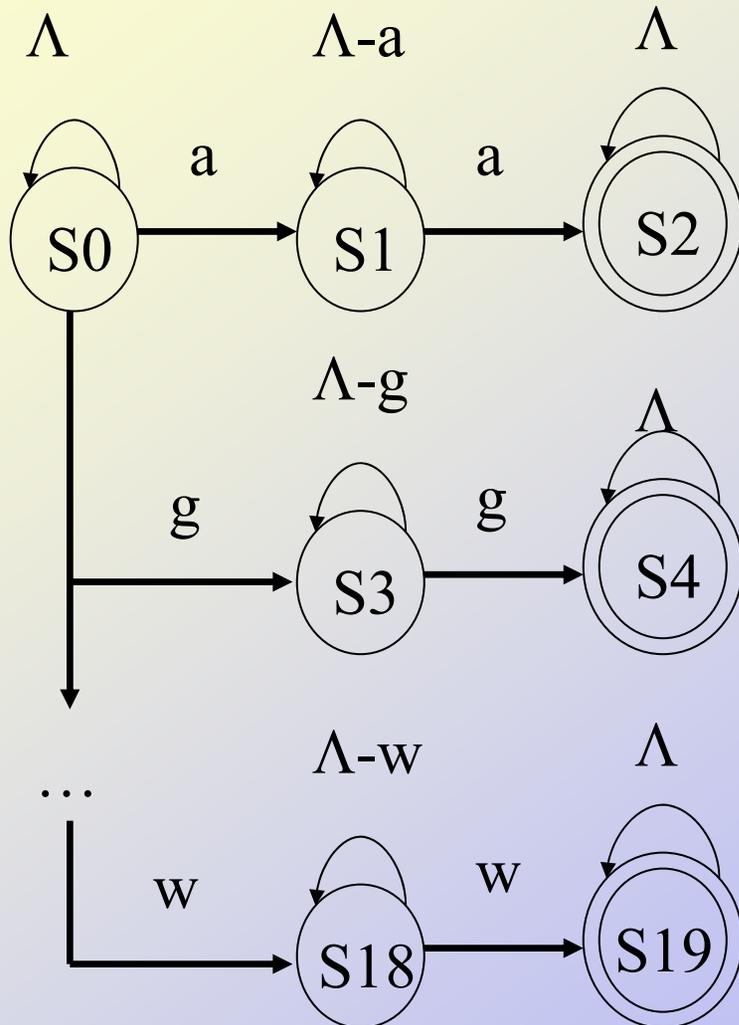


Check Letters



Filter Double a's

Problem 5 - Solution 1, cont.



Problem 5 Solution 2

```
$tr A-Z a-z </usr/dict/words | \  
egrep '^[aghinostw]*$' | \  
egrep -v \  
'a.*a|g.*g|h.*h|i.*i|n.*n.*n|o.*o|s.*s|t.*t|w.*  
w'
```

a

ago

ah

an

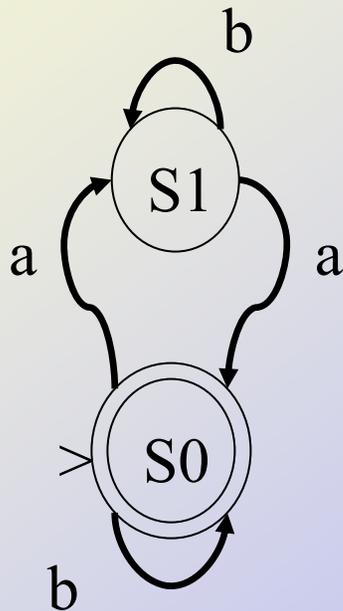
angst

State Machines and Automata

- ❖ Finite set of states, start state, Accepting States
- ❖ Transition from state to state depending on next input
- ❖ The language accepted by a finite automata is the set of input strings that end up in accepting states

Problem 6

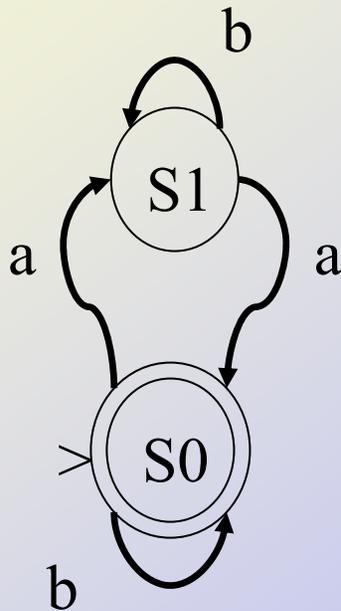
- ❖ Create a finite state automata that accepts strings of a's and b's with an even number of a's.



abbbabaabbb
011110010000

Problem 6

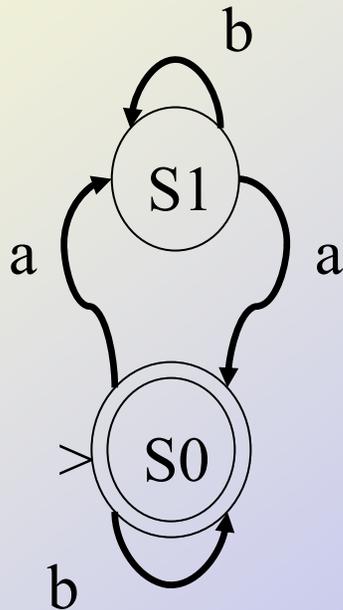
❖ Program to implement FSA



```
bool EA()  
{  
S0: x = getchar();  
    if (x == 'b') goto S0;  
    if (x == 'a') goto S1;  
    if (x == ENDM) return true;  
  
S1: x = getchar();  
    if (x == 'b') goto S1;  
    if (x == 'a') goto S0;  
    if (x == ENDM) return false;  
}
```

Problem 7

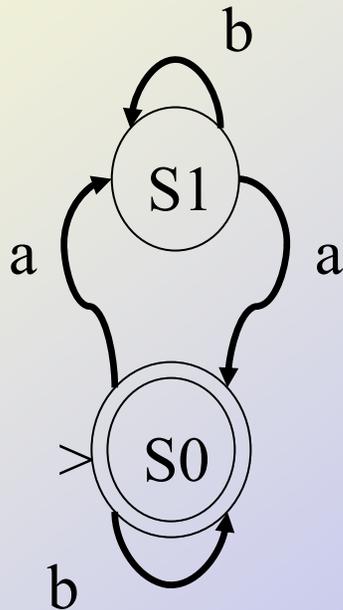
- ❖ Create a regular expression for the language that consists of strings of a's and b's with an even number of a's.



$b^*|(b^*ab^*a)^*$

Problem 8

- ❖ Create a grammar that generates the language that consists of strings of a's and b's with an even number of a's.



$$\langle S0 \rangle \rightarrow b \langle S0 \rangle$$

$$\langle S0 \rangle \rightarrow a \langle S1 \rangle$$

$$\langle S0 \rangle \rightarrow \epsilon$$

$$\langle S1 \rangle \rightarrow b \langle S1 \rangle$$

$$\langle S1 \rangle \rightarrow a \langle S0 \rangle$$

Equivalence of Regular Expressions and Finite Automata

- ❖ The languages accepted by finite automata are equivalent to those generated by regular expressions
 - Given any regular expression R , there exists a finite state automata M such that $L(M) = L(R)$ – see Problems 9 and 10 for an indication of why this is true.
 - Given any finite state automata M , there exists a regular expression R such that $L(R) = L(M)$ – see Problem 7 for an indication why this is true.

Proof of Equivalence of Regular Expressions and Finite Automata

- ❖ Sec. 10.8 of the text proves that there is a finite state automata that recognizes the language generated by any given regular expression.
- ❖ The proof is by induction on the number of operators in the regular expression and uses a finite state automata with ϵ transitions. Epsilon transitions are introduced to simplify the construction used in the proof.
- ❖ It is then shown that any finite state automata with ϵ transitions can be converted to a regular finite state automata.

Proof of Equivalence of Regular Expressions and Finite Automata

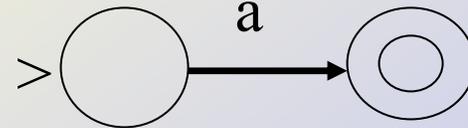
- ❖ Sec. 10.9 of the text shows how to derive a regular expression that generates the same language that is accepted by a given finite state automata.
- ❖ The basic idea is to combine the transitions in each node along all paths that lead to an accepting state. The combination of the characters along the paths are described using regular expressions.
- ❖ See Problem 7 for an example.

Proof of Equivalence of Regular Expressions and Finite Automata

- ❖ The proofs given in Sections 10.8 and 10.9 are constructive: an algorithm is given that constructs a finite state automata given a regular expression, and an algorithm is given that derives the regular expression given a finite state automata.
- ❖ This means the conversion process can be implemented. In fact, it is commonly the case that regular expressions are used to describe patterns and that a program is created to match the pattern based on the conversion of a regular expression into a finite state automata.

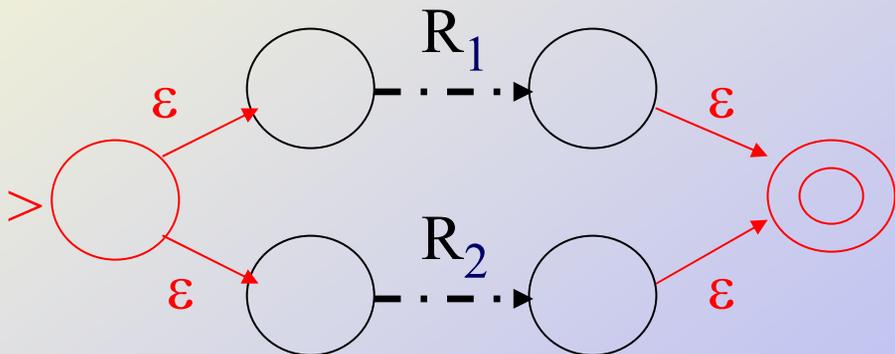
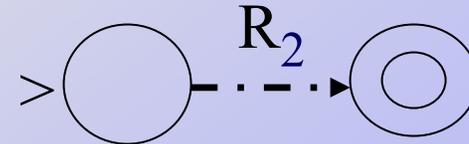
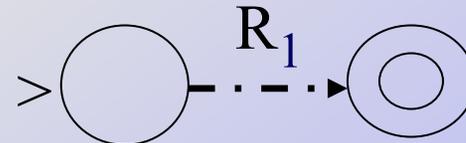
Finite State Automata from Regular Expressions

❖ Base case: a



❖ Union

➤ Given REs R_1 and R_2 †



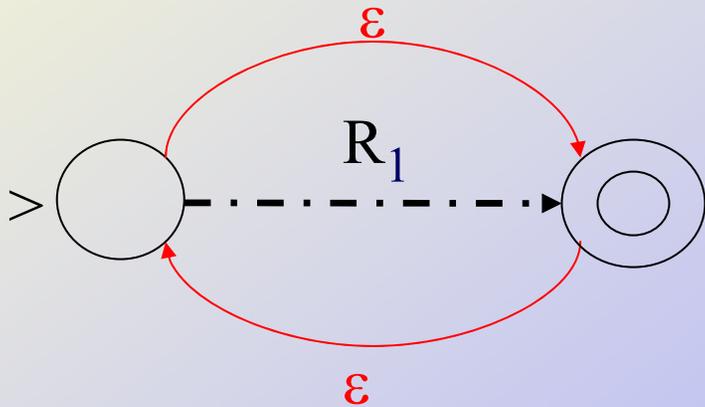
† For any machine w/more than 1 accept state, we can add a new, single accepting state, and add epsilon transitions

FSA from REs

❖ Concatenation

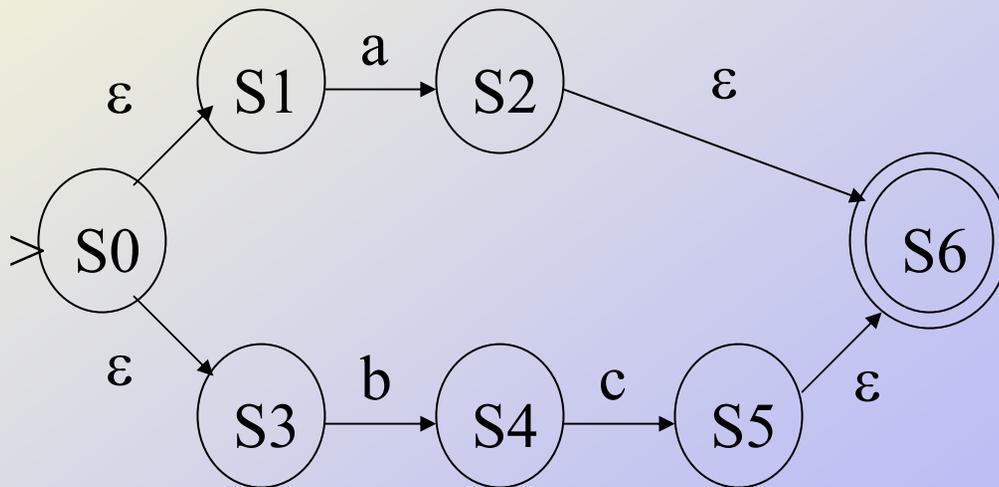


❖ Closure



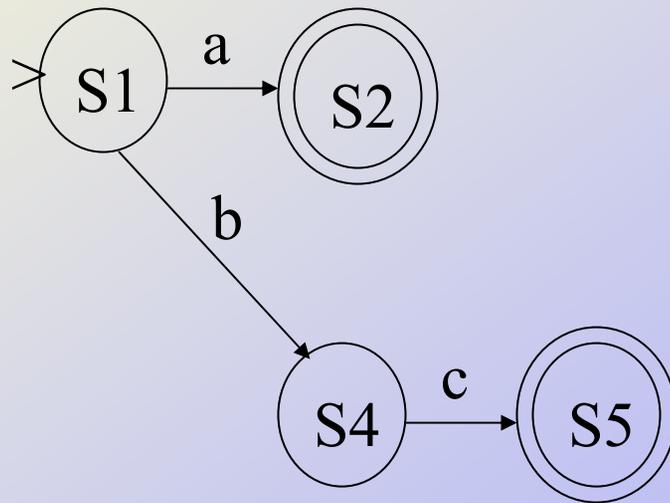
Problem 9

- ❖ Construct a finite state automata with ϵ transitions that accepts the language generated by the regular expression $(a|bc)$



Problem 10

- ❖ Find an equivalent finite state automata to the one in problem 9 that does not use ϵ transitions



Grammars and Regular Expressions

- ❖ Given a regular expression R , there exists a grammar with syntactic category $\langle S \rangle$ such that $L(R) = L(\langle S \rangle)$.
- ❖ There are grammars such that there does NOT exist a regular expression R with $L(\langle S \rangle) = L(R)$
 - $\langle S \rangle \rightarrow a\langle S \rangle b \mid \varepsilon$
 - $L(\langle S \rangle) = \{a^n b^n, n=0,1,2,\dots\}$

Proof that $a^n b^n$ is not Recognized by a Finite State Automata

- ❖ The proof is a proof by contradiction. In this type of proof, we assume that something is true and then show that this leads to a contradiction (something that is false). The only way out of this situation is that the assumption was wrong. This implies that what we assumed true is in fact false.
- ❖ To show that there is no finite state automata that recognizes the language $L = \{a^n b^n, n = 0, 1, 2, \dots\}$, we assume that there is a finite state automata M that recognizes L and show that this leads to a contradiction.

Proof that $a^n b^n$ is not Recognized by a Finite State Automata

- ❖ Since M is a finite state automata it has a finite number of states. Let the number of states = m .
- ❖ Since M recognizes the language L all strings of the form $a^k b^k$ must end up in accepting states. Choose such a string with $k = n$ which is greater than m .
- ❖ Since $n > m$ there must be a state s that is visited twice while the string a^n is read [we can only visit m distinct states and since $n > m$ after reading $(m+1)$ a 's, we must go to a state that was already visited].

Proof that $a^n b^n$ is not Recognized by a Finite State Automata

- ❖ Suppose that state s is reached after reading the strings a^j and a^k ($j \neq k$). Since the same state is reached for both strings, the finite state machine can not distinguish strings that begin with a^j from strings that begin with a^k .
- ❖ Therefore, the finite state automata must either accept or reject both of the strings $a^j b^j$ and $a^k b^j$. However, $a^j b^j$ should be accepted, while $a^k b^j$ should not be accepted.
- ❖ The only way out of this contradiction is that the assumption that there was a finite state automata that recognizes the language L was wrong.