Theory of Computing SE-2112

Lecture-3

Dr. Naushin Nower

Regular expressions

Finite automata are machine-like descriptions of languages

Alternative: declarative description

Regular expressions are an algebraic way to describe languages.

They describe exactly the regular languages.

Notation to specify a language

- Capable of describing the same thing as a NFA
  The two are actually equivalent, so RE = NFA = DFA
- We can define an algebra for regular expressions
- Specifying a language using expressions and operations
- Example: 01* + 10* defines the language containing strings such as 01111, 100, 0, 1000000; * and + are operators in this “algebra”

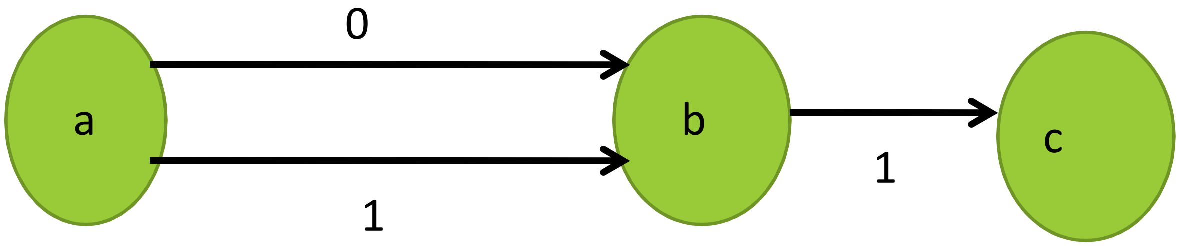
Regular expressions..

Regular expressions denote language.

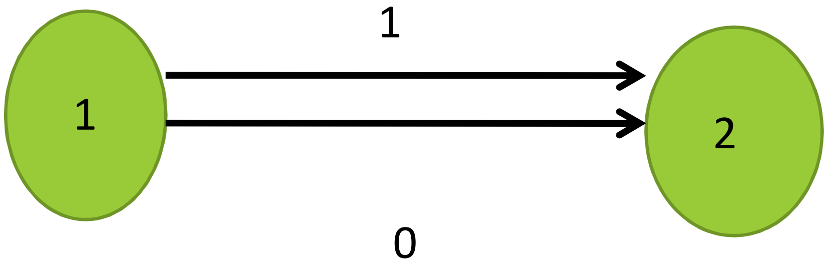
Given that R_1 and R_2 are regular expressions, regular expressions are built from the following operations

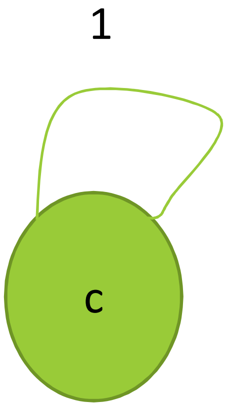
- Union: R_1+R_2
- Concatenation: R_1R_2
- Kleene Star Closure: R_1*
- Parentheses (to enforce precedence): (R_1)

Nothing else is a regular expression unless it is built from the above rules



(0+1)1


1
1
2
0


1
c

$(01)^* + (10)^* + 1(01)^* + 0(10)^*$

01 , 010101, 0101,……

10, 1010……

101

010

Examples

- 1 + ε
- (ab)* + (ba)*
- (0+1+2+3+4+5+6+7+8+9)*(0+5)
- (x+y)*x(x+y)*
- (01)*
- 0(1*)
- 01* equivalent to 0(1*)

RE's: Definition

Basis 1: If *a* is any symbol, then **a** is a RE, and L(**a**) = {a}.

- Note: {a} is the language containing one string, and that string is of length 1.

Basis 2: ϵ is a RE, and L(ϵ) = {ϵ}.

Basis 3: ∅ is a RE, and L(∅) = ∅.

RE's: Definition – (2)

Induction 1: If E_1 and E_2 are regular expressions, then E_1+E_2 is a regular expression, and $L(E_1+E_2) = L(E_1) \cup L(E_2)$.

Induction 2: If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$.

Concatenation : the set of strings wx such that w Is in $L(E_1)$ and x is in $L(E_2)$.

RE’s: Definition – (3)

Induction 3: If E is a RE, then E* is a RE, and L(E*) = (L(E))*.

Closure, or “Kleene closure” = set of strings $w_1w_2\ldots w_n$, for some $n \geq 0$, where each w_i is in L(E).

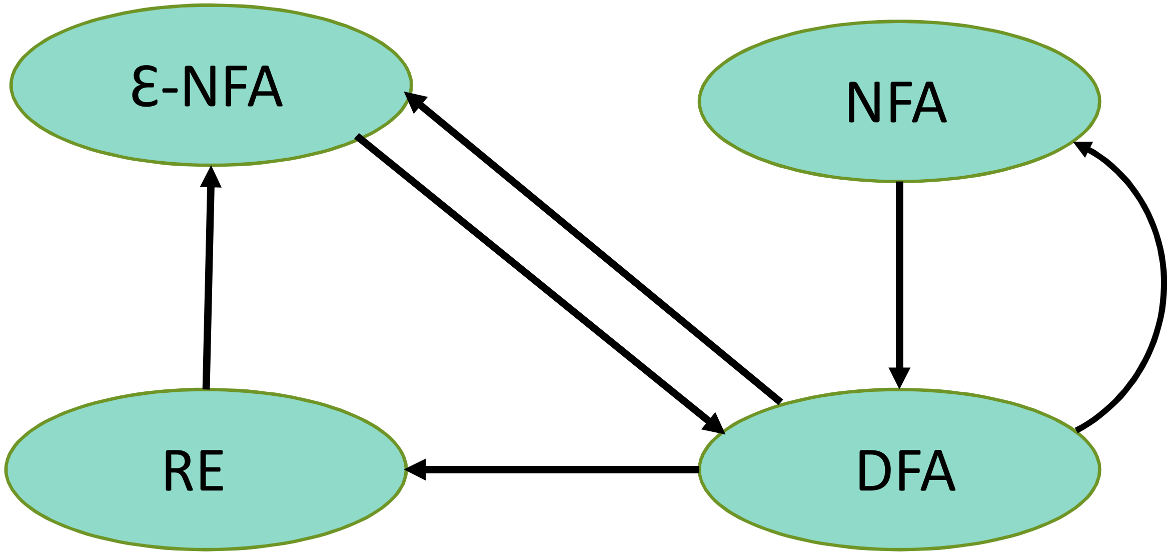
Note: when n=0, the string is ϵ.

Precedence of Operators

Parentheses may be used wherever needed to influence the grouping of operators.

Order of precedence is * (highest), then concatenation, then + (lowest).

From DFA to RE



Equivalence of FA and RE

Finite Automata and Regular Expressions are equivalent. To show this:

- Show we can express a DFA as an equivalent RE
- Show we can express a RE as an ε-NFA. Since the ε-NFA can be converted to a DFA and the DFA to an NFA, then RE will be equivalent to all the automata we have described.

DFA to Regular Expression

Theorem: If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).

Build the regular expression "bottom up" starting with simpler strings that are acceptable using a subset of states in the DFA

Define $R^{k}_{i,j}$ as the expression for strings that have an admissible state sequence from state i to state j with no intermediate states greater than k

- Assume no states are numbered 0, but k can be 0

$R^0_{i,j}$

Observe that $R^0_{i,j}$ describes strings of length 1 or 0, particularly:

- $\{a_1, a_2, a_3, \ldots\}$, where, for each a_x, $\delta(i,a_x) = j$
- Add ε to the set if $i = j$

The 0 in $R^0_{i,j}$ means no intermediate states are allowed, so either no transition is made (just stay in state i to accept ε if $i = j$) or make a single transition from state i to state j

These are the base cases in our construction

$R^{k}_{i,j}$

Recursive step: for each k, we can build $R^{k}_{i,j}$ as follows:

$$R^{k}_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k} (R^{k-1}_{k,k})^* R^{k-1}_{k,j}$$

Intuition: since the accepting sequence contains one or more visits to state k, break the path into pieces that

- first goes from i to its first k-visit ($R^{k-1}_{i,k}$)
- followed by zero or more revisits to k ($R^{k-1}_{k,k}$)
- followed by a path from k to j ($R^{k-1}_{k,j}$)

And finally…

We get the regular expression(s) that represent all strings with admissible sequences that start with the initial state (state 1) and end with a final state

Resulting regular expression built from the DFA: the union of all $R^{n}_{1,f}$ where f is a final state

- Note: n is the number of states in the DFA meaning there are no more restrictions for intermediate states in the accepting sequence

DFA to RE: State Elimination

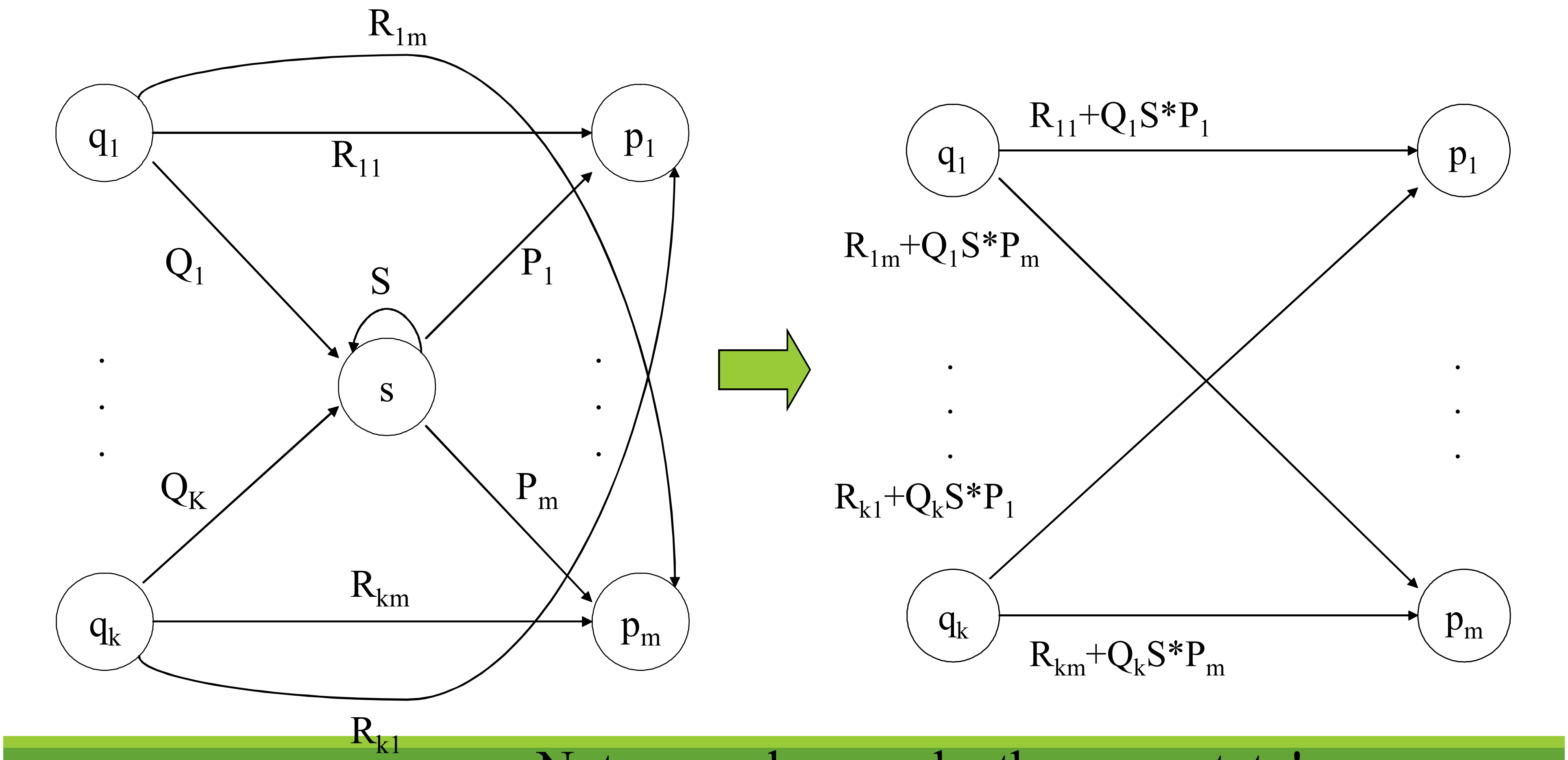
Eliminates states of the automaton and replaces the edges with regular expressions that includes the behavior of the eliminated states.

Eventually we get down to the situation with just a start and final node, and this is easy to express as a RE

State Elimination

Consider the figure below, which shows a generic state S about to be eliminated. The labels on all edges are regular expressions.

To remove S we must make labels from each q_i to p_1 up to p_m that include the paths we could have made through S.



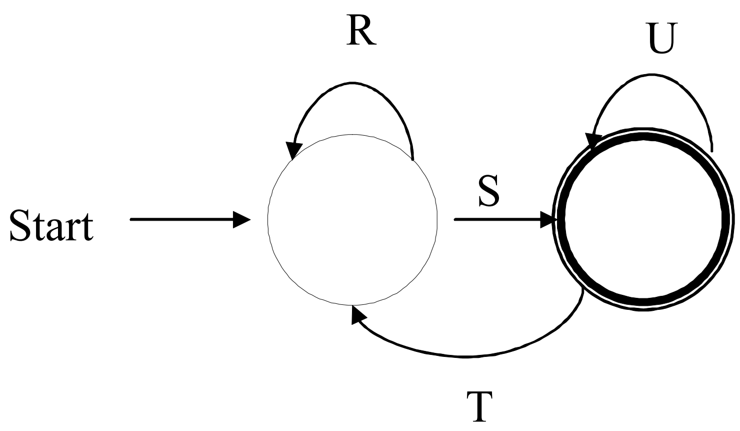
Note: q and p may be the same state!

DFA to RE via State Elimination (1)

1. Starting with intermediate states and then moving to accepting states, apply the state elimination process to produce an equivalent automaton with regular expression labels on the edges.
    - The result will be a one or two state automaton with a start state and accepting state.

DFA to RE State Elimination (2)

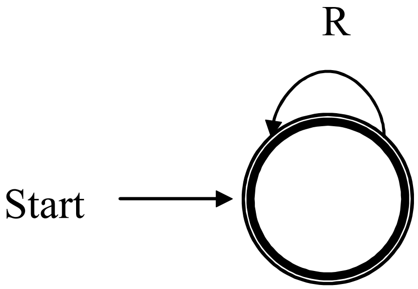
2. If the two states are different, we will have an automaton that looks like the following:



We can describe this automaton as: (R+SU*T)*SU*

DFA to RE State Elimination (3)

3. If the start state is also an accepting state, then we must also perform a state elimination from the original automaton that gets rid of every state but the start state. This leaves the following:



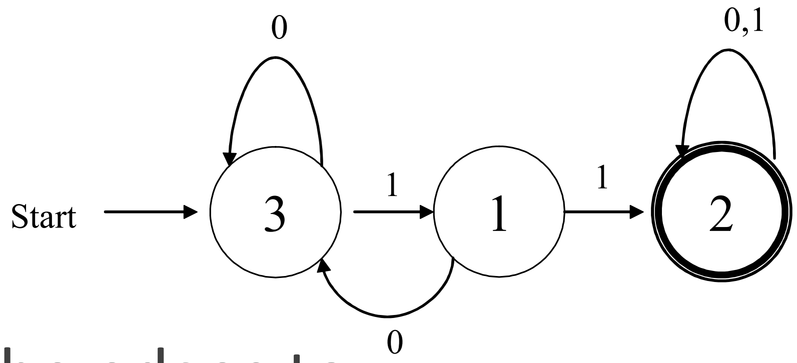
We can describe this automaton as simply R*.

DFA to RE State Elimination (4)

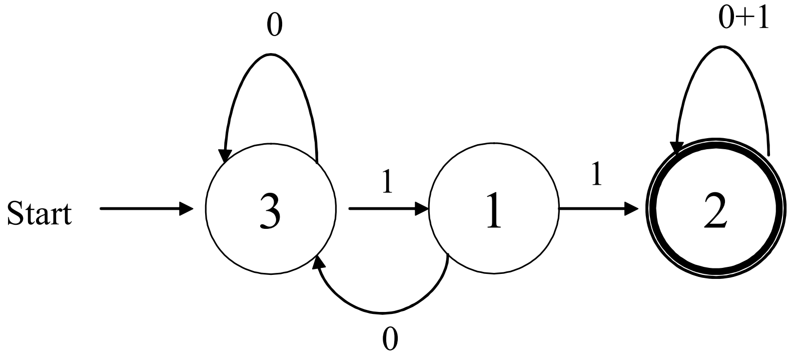
4. If there are n accepting states, we must repeat the above steps for each accepting states to get n different regular expressions, R_1, R_2, … R_n. For each repeat we turn any other accepting state to non-accepting. The desired regular expression for the automaton is then the union of each of the n regular expressions: $R_1 \cup R_2 \ldots \cup R_N$

DFA→RE Example

Convert the following to a RE



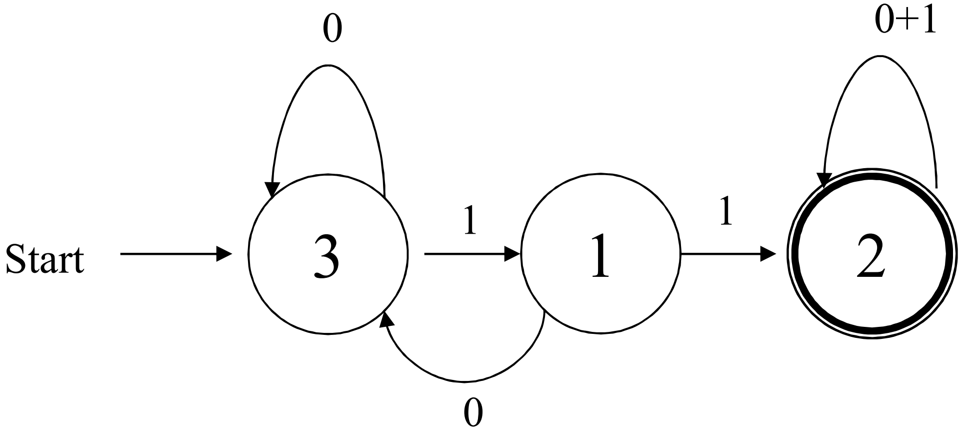
First convert the edges to RE's:



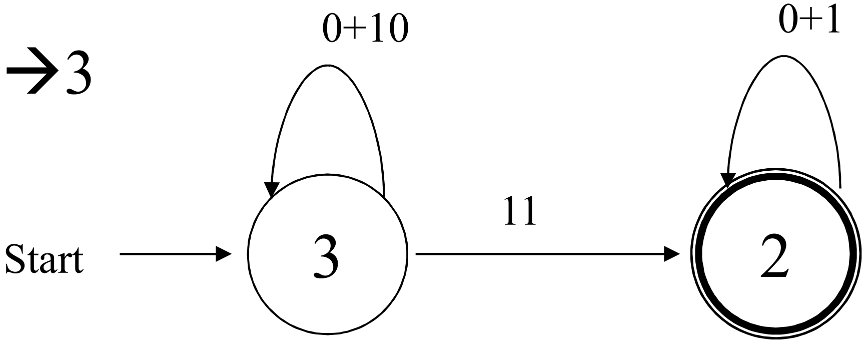
DFA → RE Example (2)

Eliminate State 1:

To:



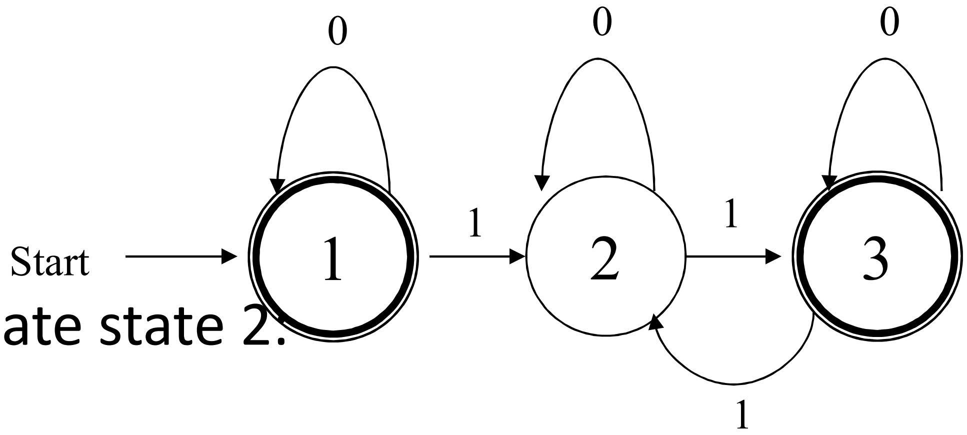
Note edge from 3→3



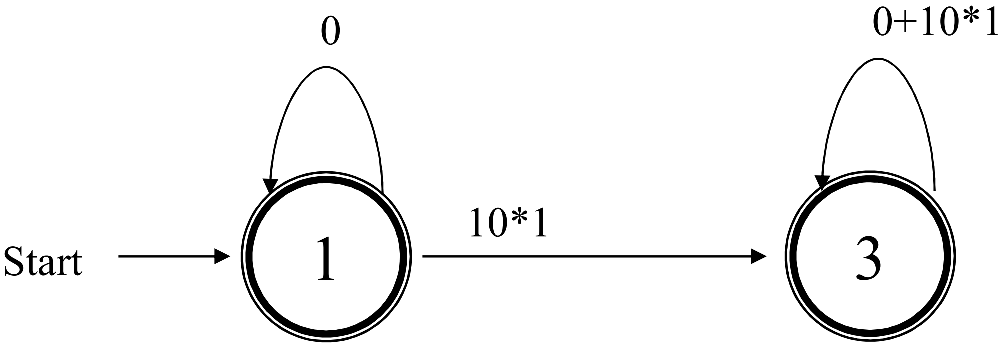
Answer: (0+10)*11(0+1)*

Second Example

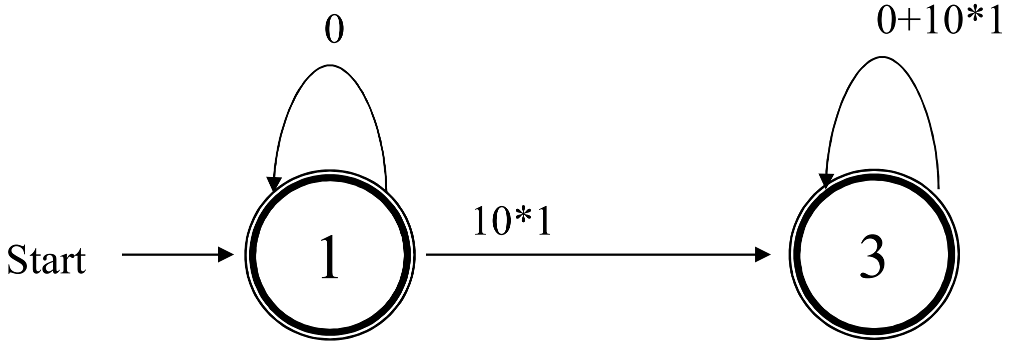
Automata that accepts even number of 1’s



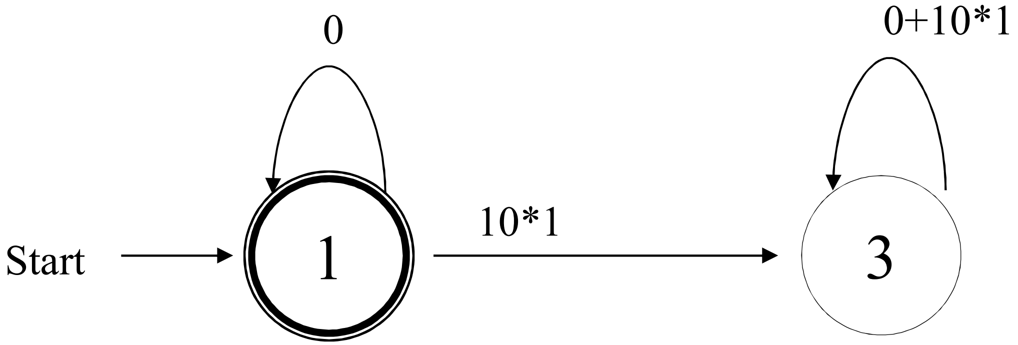
Eliminate state 2:



Second Example (2)

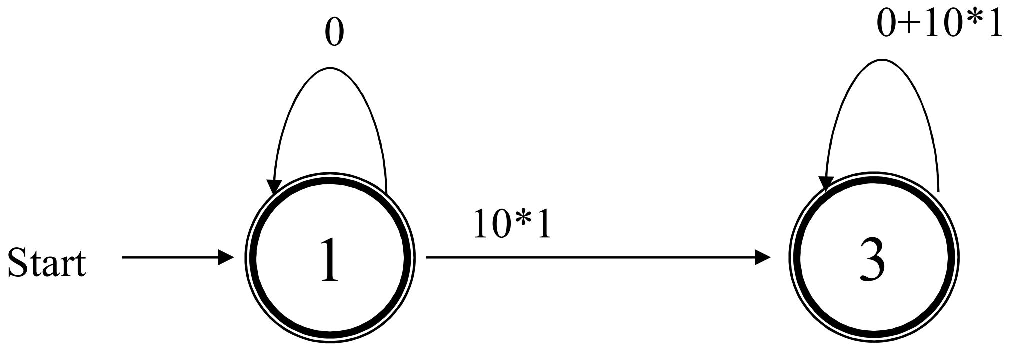


Two accepting states, turn off state 3 first

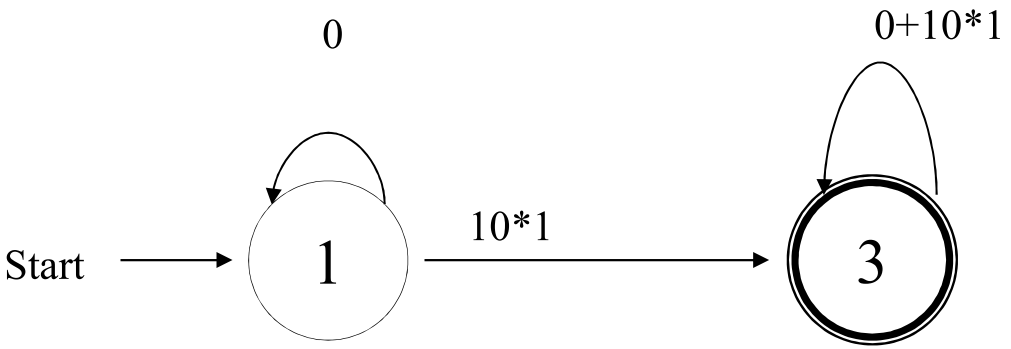


This is just 0*; can ignore going to state 3 since we would “die”

Second Example (3)



Turn off state 1 second:



This is just 0*10*1(0+10*1)*

Combine from previous slide to get
0* + 0*10*1(0+10*1)*

Converting a RE to an Automata

We have shown we can convert an automata to a RE. To show equivalence we must also go the other direction, convert a RE to an automaton.

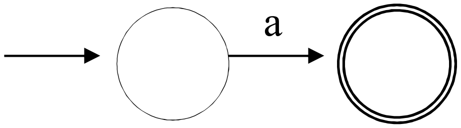
We can do this easiest by converting a RE to an ε-NFA

- Inductive construction
- Start with a simple basis, use that to build more complex parts of the NFA

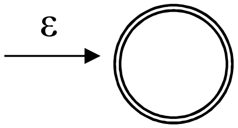
RE to ε-NFA

Basis:

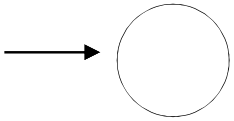
R=a



R=ε

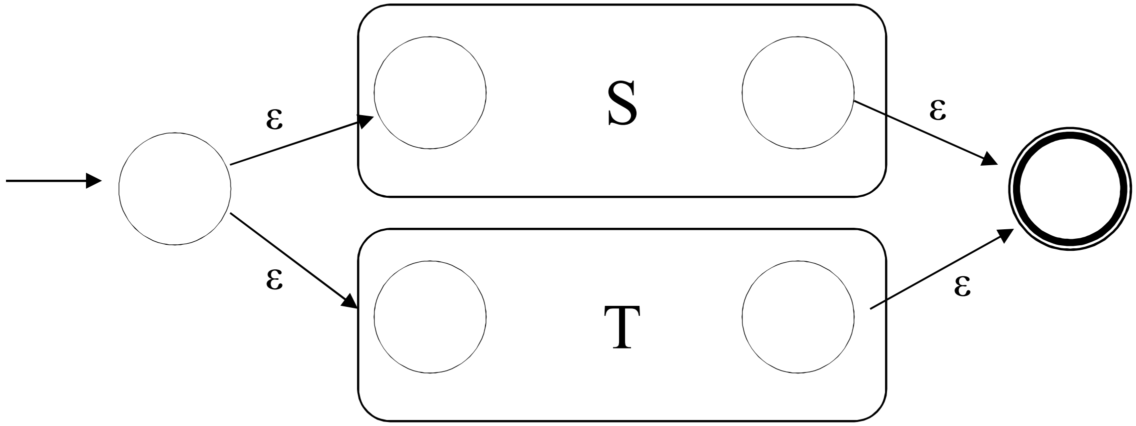


R=Ø

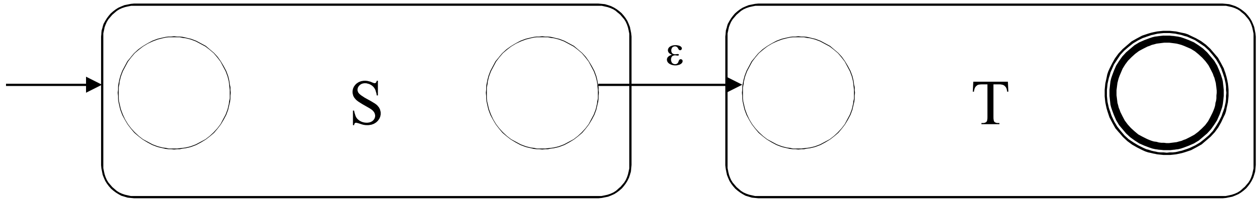


Next slide: More complex RE’s

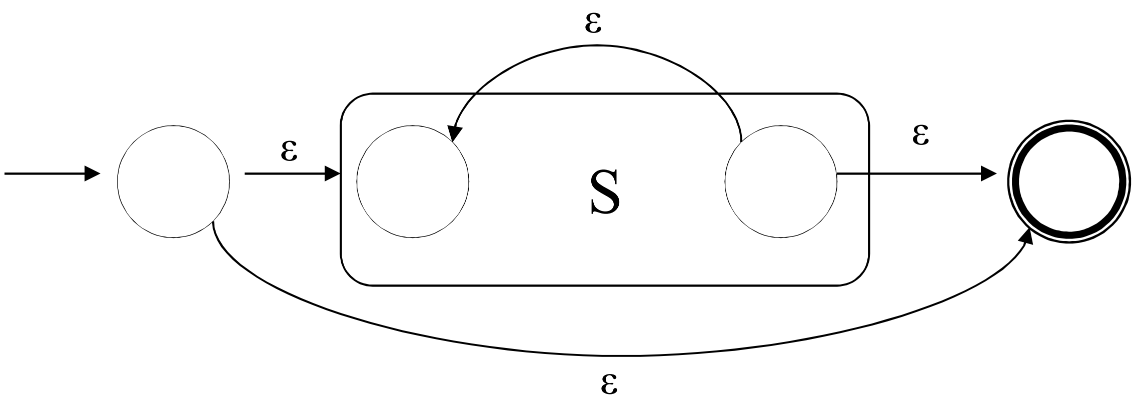
R=S+T



R=ST



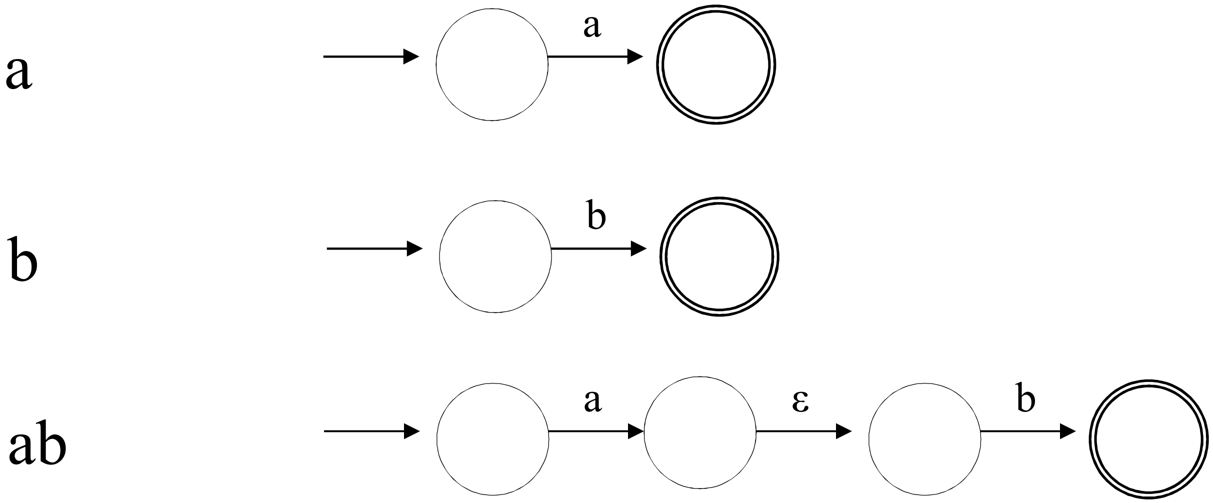
R=S*



RE to ε-NFA Example

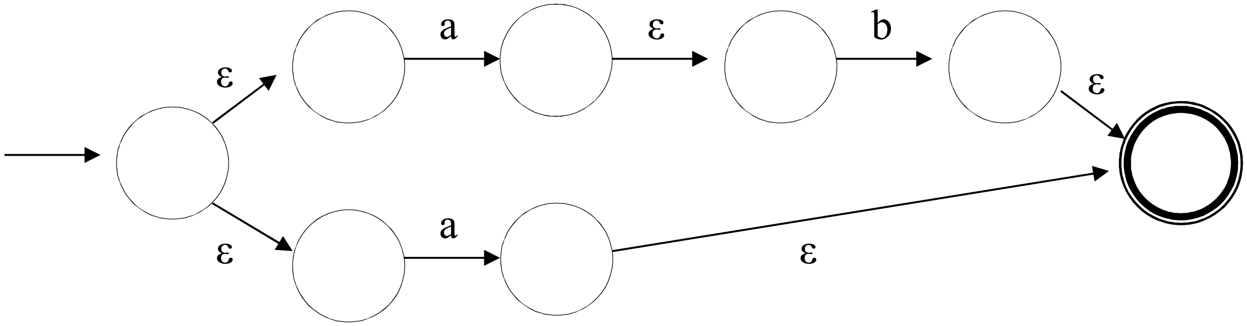
Convert R= (ab+a)* to an NFA

- We proceed in stages, starting from simple elements and working our way up

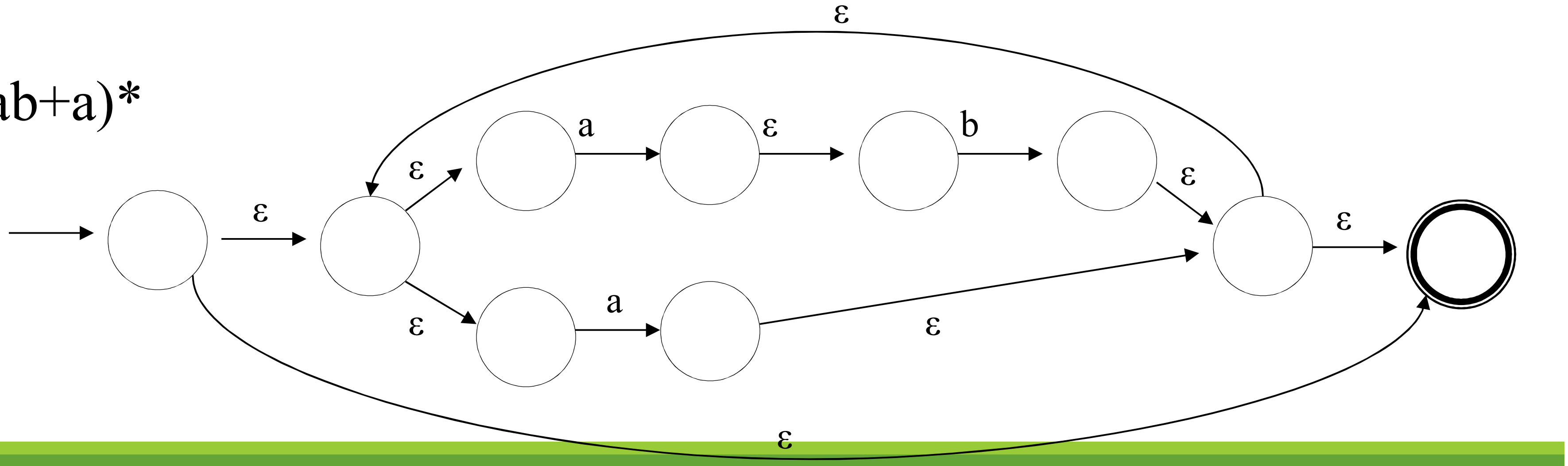


RE to ε-NFA Example (2)

ab+a



(ab+a)*



What have we shown?

Regular expressions and finite state automata are really two different ways of expressing the same thing.

In some cases you may find it easier to start with one and move to the other

- E.g., the language of an even number of one’s is typically easier to design as a NFA or DFA and then convert it to a RE

Algebraic Laws for RE’s

Just like we have an algebra for arithmetic, we also have an algebra for regular expressions.

- While there are some similarities to arithmetic algebra, it is a bit different with regular expressions.

Algebra for RE’s

Commutative law for union:

- L + M = M + L

Associative law for union:

- (L + M) + N = L + (M + N)

Associative law for concatenation:

- (LM)N = L(MN)

Note that there is no commutative law for concatenation, i.e. LM ≠ ML

Algebra for RE’s (2)

The identity for union is:

- $L + \emptyset = \emptyset + L = L$

The identity for concatenation is:

- $L\varepsilon = \varepsilon L = L$

The annihilator for concatenation is:

- $\emptyset L = L\emptyset = \emptyset$

Left distributive law:

- $L(M + N) = LM + LN$

Right distributive law:

- $(M + N)L = LM + LN$

Idempotent law:

- $L + L = L$

Laws Involving Closure

$(L^*)^* = L^*$

- i.e. closing an already closed expression does not change the language

$\emptyset^* = \varepsilon$

$\varepsilon^* = \varepsilon$

$L^+ = LL^* = L^*L$

- more of a definition than a law

$L^* = L^+ + \varepsilon$

$L? = \varepsilon + L$

- more of a definition than a law

Checking a Law

Suppose we are told that the law

(R + S)* = (R*S*)*

holds for regular expressions. How would we check that this claim is true?

1. Convert the RE's to DFA's and minimize the DFA's to see if they are equivalent (we'll cover minimization later)
2. We can use the "concretization" test:
   - Think of R and S as if they were single symbols, rather than placeholders for languages, i.e., R = {0} and S = {1}.
   - Test whether the law holds under the concrete symbols. If so, then this is a true law, and if not then the law is false.

Concretization Test

For our example
(R + S)* = (R*S*)*

We can substitute 0 for R and 1 for S.

The left side is clearly any sequence of 0's and 1's. The right side also denotes any string of 0's and 1's, since 0 and 1 are each in L(0*1*).

Concretization Test

NOTE: extensions of the test beyond regular expressions may fail.

Consider the “law” $L \cap M \cap N = L \cap M$.

This is clearly false

- Let $L=M=\{a\}$ and $N=\emptyset$. $\{a\} \neq \emptyset$.
- But if $L=\{a\}$ and $M = \{b\}$ and $N=\{c\}$ then
- $L \cap M$ does equal $L \cap M \cap N$ which is empty.
- The test would say this law is true, but it is not because we are applying the test beyond regular expressions.

We’ll see soon various languages that do not have corresponding regular expressions.

Thank you ☺