Regular Expressions and Regular Languages

- Regular Expressions
- Converting Regular Expressions to NFA
- Converting Finite Automata to Regular Expressions
- Algebraic Laws for Regular Expressions
Regular Expressions

- We used **Finite Automata** to describe **regular languages**.
- We can also use **regular expressions** to describe **regular languages**.

- **Regular Expressions** are an algebraic way to describe languages.
- **Regular Expressions** describe exactly the **regular languages**.
- If $E$ is a regular expression, then $L(E)$ is the regular language that it defines.
- For each regular expression $E$, we can create a DFA $A$ such that $L(E) = L(A)$.
- For each a DFA $A$, we can create a regular expression $E$ such that $L(A) = L(E)$.
- A regular expression is built up of simpler regular expressions (using defining rules)
Operations on Languages

- Remember: A language is a set of strings
- We can perform operations on languages.

Union: \[ L \cup M = \{ w : w \in L \text{ or } w \in M \} \]

Concatenation: \[ L.M = \{ w : w=xy, x \in L, y \in M \} \]

Powers: \[ L^0 = \{ \varepsilon \}, \quad L^1 = L, \quad L^{k+1} = L \cdot L^k \]

Kleene Closure: \[ L^* = \bigcup_{i=0}^{\infty} L^i \]
Operations on Languages - Examples

$L = \{00,11\} \quad M = \{1,01,11\}$

$L \cup M = \{00,11,1,01\}$

$L.M = \{001,0001,0011,111,1101,1111\}$

$L^0 = \{\varepsilon\} \quad L^1 = L = \{00,11\} \quad L^2 = \{0000,0011,1100,1111\}$

$L^* = \{\varepsilon, 00, 11, 0000, 0011, 1100, 1111, 000000, 000011, \ldots\}$

Kleene closures of all languages (except two of them) are infinite.

1. $\phi^* = \{\}^* = \{\varepsilon\}$
2. $\{\varepsilon\}^* = \{\varepsilon\}$
# Regular Expressions - Definition

Regular expressions over alphabet $\Sigma$

<table>
<thead>
<tr>
<th>Reg. Expr. E</th>
<th>Language it denotes L(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi$</td>
<td>${ }$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>${ \epsilon }$</td>
</tr>
<tr>
<td>$a \in \Sigma$</td>
<td>${a}$</td>
</tr>
</tbody>
</table>

*Note:*

$\{a\}$ is the language containing one string, and that string is of length 1.
Regular Expressions - Definition

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

- Sipser's book uses union symbol $\cup$ to represent the **or** operator instead of $+$. Some people also use the bar symbol $|$ to represent the **or** operator.

Induction 2 – concatenation: If $E_1$ and $E_2$ are regular expressions, then $E_1 E_2$ is a regular expression, and $L(E_1 E_2) = L(E_1).L(E_2)$ where $L(E_1).L(E_2)$ is the set of strings $wx$ such that $w$ is in $L(E_1)$ and $x$ is in $L(E_2)$.

Induction 3 – Kleene Closure: If $E$ is a regular expression, then $E^*$ is a regular expression, and $L(E^*) = (L(E))^*$.

Induction 4 – Parentheses: If $E$ is a regular expression, then $(E)$ is a regular expression, and $L((E)) = L(E)$.
Regular Expressions - Parentheses

- Parentheses may be used wherever needed to influence the grouping of operators.
- We may remove parentheses by using precedence and associativity rules.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>highest</td>
<td></td>
</tr>
<tr>
<td>concatenation</td>
<td>next</td>
<td>left associative</td>
</tr>
<tr>
<td>+</td>
<td>lowest</td>
<td>left associative</td>
</tr>
</tbody>
</table>

\[ ab^*+c \text{ means (a((b)^*))}+(c) \]
Regular Expressions - Examples

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

Regular Expression: $01$

- $L(01) = \{01\}$
- $L(01) = L(0) L(1) = \{0\} \{1\} = \{01\}$

Regular Expression: $01+0$

- $L(01+0) = \{01, 0\}$
- $L(01+0) = L(01) \cup L(0) = (L(0) L(1)) \cup L(0)$
- $= (\{0\}\{1\}) \cup \{0\} = \{01\} \cup \{0\} = \{01,0\}$

Regular Expression: $0(1+0)$

- $L(0(1+0)) = \{01, 00\}$
- $L(0(1+0)) = L(0) L(1+0) = L(0) (L(1) \cup L(0))$
- $= \{0\} (\{1\} \cup \{0\}) = \{0\} \{1,0\} = \{01,00\}$

- Note order of precedence of operators.
Regular Expressions -- Examples

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

**Regular Expression:** $0^*$

- $L(0^*) = \{\varepsilon, 0, 00, 000, \ldots \} =$ all strings of 0’s, including the empty string

**Regular Expression:** $(0+10)^*(\varepsilon+1)$

- $L((0+10)^*(\varepsilon+1)) =$ all strings of 0’s and 1’s without two consecutive 1’s.

**Regular Expression:** $(0+1)(0+1)$

- $L((0+1)(0+1)) = \{00,01,10,11\} =$ all strings of 0’s and 1’s of length 2.

**Regular Expression:** $(0+1)^*$

- $L((0+1)^*) =$ all strings with 0 and 1, including the empty string
Regular Expressions
for Given Regular Languages -- Examples

Language: All strings of 0’s and 1’s starting with 0 and ending with 1
0(0+1)*1

Language: All strings of 0’s and 1’s with at least two consecutive 0’s
(0+1)*00 (0+1)*

Language: All strings of 0’s and 1’s without two consecutive 0’s
((1+01)*(ε+0))

Language: All strings of 0’s and 1’s with even number of 0’s
1*(01*01*)*
Converting Regular Expressions to NFA
Converting Regular Expressions to NFA

- For every regular expression there is a finite automaton.
- We will give an algorithm which converts a given regular expression to a NFA.
- We have already discussed how to convert a NFA to a DFA using subset construction.
- Thus, there is a NFA for each regular expression and their languages are equivalent.
- And, there is a DFA for each regular expression and their languages are equivalent.

Regular Expression $\rightarrow$ NFA $\rightarrow$ DFA

NFA construction algorithm

subset construction algorithm
Theorem: Every language defined by a regular expression is also defined by a finite automaton.

• This theorem says that every language represented by a regular expression is a regular language (i.e. There is a DFA which recognizes that language)

• In the proof of this theorem, we will create a NFA which recognizes the language of a given regular expression. This means that any language represented by a regular expressions can be recognized by a NFA.
  – Previously, we show how to create an equivalent DFA for a given NFA. This means that any language recognized by a NFA can be recognized by a DFA.

Regular Expressions $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Regular Languages
Theorem: Every language defined by a regular expression is also defined by a finite automaton.

Proof:

• Suppose that L(R) is the language of a regular expression R.

• A NFA construction for a regular expression: We show that for some NFA A whose language L(A) is equal to L(R), and this NFA A has following properties:
  1. NFA A has exactly one accepting state.
  2. No arcs into the initial state.
  3. No arcs out of the accepting state.

• The proof is by structural induction on R following the recursive definition of regular expressions
There are 3 base cases.

a) Regular Expression $R = \varepsilon$
   
   $L(\varepsilon) = \{\varepsilon\}$

   NFA A:

   ![NFA A](image1)

   $L(A) = \{\varepsilon\}$

b) Regular Expression $R = \phi$

   $L(\phi) = \{\}$

   NFA A:

   ![NFA A](image2)

   $L(A) = \{\}$

c) Regular Expression $R = a \in \Sigma$

   $L(a) = \{a\}$

   NFA A:

   ![NFA A](image3)

   $L(A) = \{a\}$
Converting Regular Expressions to NFA

Induction

Inductive Hypothesis:

• We assume that the statement of the theorem is true for immediate subexpressions of a given regular expression; i.e. the languages of these subexpressions are also the languages of NFAs with a single accepting state.

Induction:

• There are four cases for the induction:
  1. $R + S$
  2. $R S$
  3. $R^*$
  4. $(R)$
Converting Regular Expressions to NFA

Induction Case:  \( R + S \)

Regular Expression:  \( R + S \)  \( L(R+S) = L(R) \cup L(S) \)

NFA \( A \):

- By IH, we have automaton \( R \) for regular expression \( R \), and automaton \( S \) for regular expression \( S \), and a new automaton for \( R+S \) is constructed as above.
- Starting at new start state, we can go to start states of automatons \( R \) or \( S \).
- For some string in \( L(R) \) or \( L(S) \), we can reach accepting state of \( R \) or \( S \).
- From there, we can reach accepting state of the new automaton by \( \varepsilon \)–transition.

• Thus,  \( L(A) = L(R) \cup L(S) \)
Converting Regular Expressions to NFA

Induction Case: \( RS \)

Regular Expression: \( RS \)  \[ L(RS) = L(R) \ L(S) \]

NFA A:

- By IH, we have automaton R for regular expression R, and automaton S for regular expression S, and **a new automaton for RS is constructed as above**.
- Starting at **starting state of R**, we can reach **accepting state of R** by recognizing a string in \( L(R) \).
- From **accepting state of R**, we can reach **starting state of S** by \( \varepsilon \)–transition.
- From **starting state of S**, we can reach **accepting state of S** by recognizing a string in \( L(S) \).
- The **accepting state of S** is also the accepting state of the new automaton A.

• **Thus,** \( L(A) = L(R) \ L(S) \)
Regular Expression: \( R^* \)

\[ \text{L}(R^*) = (\text{L}(R))^* \]

NFA A:

By IH, we have automaton R for regular expression R, and a new automaton for \( R^* \) is constructed as above.

Starting at new starting state, we can reach new accepting state. \( \varepsilon \) is in \( (\text{L}(R))^* \).

Starting at new starting state, we can reach starting state of R. From starting state of R, we can reach accepting state of R recognizing a string in L(R). We can repeat this one or more times by recognizing strings in L(R), L(R)L(R),….

Thus, \( \text{L}(A) = (\text{L}(R))^* \)
Converting Regular Expressions to NFA

Induction Case: \((R)\)

Regular Expression: \((R)\)

- By IH, we have automaton \(R\) for regular expression \(R\), and a new automaton for \((R)\) is same as the automaton of \(R\).

- The automaton for \(R\) also serves as the automaton for \((R)\) since the parentheses do not change the language defined by the expression.
Example: Convert \((0+1)^*1(0+1)\) to NFA

Automaton for 0:

 Automaton for 1:

Automaton for 0+1:
Example: Convert \((0+1)^*1(0+1)\) to NFA

Automaton for \((0+1)^*\):
Example: Convert \((0+1)^*1(0+1)\) to NFA

Automaton for \((0+1)^*1(0+1)\):
Example: Convert \((0+1)^*1\) to NFA

Automaton for 1:

Automaton for \((0+1)^*\):

Automaton for \((0+1)^*1\):
Example: Conversion of NFA of \((0+1)^*1\) to DFA

- Convert this NFA to a DFA using subset construction
Example: Conversion of NFA of $(0+1)^*1$ to DFA

$E\text{-CL}(q_0) = \{q_0, q_1, q_2, q_3, q_7, q_8\}$

$E\text{-CL}(\{q_4\}) = \{q_4, q_1, q_2, q_3, q_6, q_7, q_8\}$

$E\text{-CL}(\{q_5, q_9\}) = \{q_5, q_9, q_1, q_2, q_3, q_6, q_7, q_8\}$
Converting Finite Automata to Regular Expressions
Converting DFA to Regular Expressions

**Theorem:** If a language is regular, then it is described by a regular expression.

- In order to prove this theorem, we will create a regular expression for any given DFA and the language of this regular expression is equivalent to the language of that DFA.
  - Since a regular language is described by a DFA, a regular language is also described by a regular expression.
Converting DFA to Regular Expressions

- In order to create a *regular expression which describes the language of the given DFA*:

- First, we create a **Generalized NFA (GNFA)** from the given DFA

- A GNFA has **generalized transitions** and a **generalization transition** is a *transition whose label is a regular expression*.

- Then, we will iteratively eliminate states of the GNFA one by one, until only two states (start state and an accepting state) and a single generalized transition is left.

- The label of this single transition (a regular expression) will be the regular expression describes the language of the given DFA.
Converting DFA to Regular Expressions

**Generalization Transitions**

- When a DFA has single symbols as transition labels:
  - If we are in state $p$ and the next input symbol matches $a$, go to state $q$.

- Now, look at a **generalized transition**:
  - If we are in state $p$ and a prefix of the remaining input matches the regular expression $ab^*+ba$ then go to state $q$.
  - A **generalization transition** is a transition whose label is a regular expression.
A Generalized NFA (GNFA) is an NFA with generalized transitions.

In fact, all standard DFA transitions with single symbols are generalized transitions with regular expressions of a single symbol!
Consider the following DFA.

![DFA Diagram]

What will be the corresponding GNFA with two states (start state and an accepting state) with a single generalized transition.

- $0^*1$ takes the DFA from state $p$ to $q$
- $(0+10^*1)^*$ takes the DFA from $q$ back to $q$
- So, $0^*1(0+10^*1)^*$ represents all strings take the DFA from state $p$ to $q$. 

![GNFA Diagram]
Converting DFA to GNFA

- We will convert the given DFA to a **GNFA in a special form**. We will add two new states to a DFA:
  - A **new start state** with an ε-transition to the original start state, but there will be **no other transitions from any other state to this new start state**.
  - A **new final state** with an ε-transition from all the original final states, but there will be **no other transitions from this new final state to any other state**.

- If the label of the DFA is a single symbol, the corresponding label of the GNFA will be that single symbol: 0 ⟵ 0

- If there are more than one symbol on the label of the DFA, the corresponding label of the GNFA will be union (OR) of those symbols: 0,1 ⟵ 0+1

- The previous start and final states will be non-accepting states in this GNFA.
Converting DFA to GNFA

DFA

GNFA in a special form
Reducing A GNFA

- We eliminate all states of the GNFA one-by-one leaving only the start state and the final state.

- When the GNFA is fully converted, the label of the only generalized transition is the regular expression for the language accepted by the original DFA.
Converting a DFA to a Regular Expression

- Assume that our DFA has 3 states.
  - Create a GNFA with 5 states in a special form.
  - Eliminate a state on-by-one until we obtain a GNFA with two states (start state and final state).
  - Label on the arc is the regular expression describing the language of the DFA.
Suppose we want to eliminate state $q_k$, and $q_i$ and $q_j$ are two of the remaining states (i=j is possible; i.e. $q_i$ can be equal to $q_j$).

How can we modify the transition label between $q_i$ and $q_j$ to reflect the fact that $q_k$ will no longer be there?

- There are two paths between $q_i$ and $q_j$
  - Direct path with regular expression $R_{ij}$
  - Path via $q_k$ with the regular expression $(R_{ik})(R_{kk})^*(R_{kj})$
Eliminating States

- There are two paths between $q_i$ and $q_j$
  - Direct path with regular expression $R_{ij}$
  - Path via $q_k$ with the regular expression $(R_{ik})(R_{kk})^*(R_{kj})$

- After removing $q_k$, the new label would be
  new $(R_{ij}) = (R_{ij}) + (R_{ik})(R_{kk})^*(R_{kj})$
Eliminating States

- When we are eliminating a state q, we have to update labels of state pairs p and r such that there is a transition from p to q and there is a transition from q to r.
  - p and r can be same state.
  - Missing arc labels are $\phi$

$$R_{pp} = R_{pp} + R_{pq} (R_{qq})^* R_{qp} = \phi + 1(1)^* \phi = \phi$$

$$R_{pr} = R_{pr} + R_{pq} (R_{qq})^* R_{qr} = 0 + 1(1)^*0 = 0 + 11*0$$

$$R_{rr} = R_{rr} + R_{rq} (R_{qq})^* R_{qr} = \phi + 1(1)^*0 = 11*0$$

$$R_{rp} = R_{rp} + R_{rq} (R_{qq})^* R_{qp} = \phi + 1(1)^* \phi = \phi$$
Some Simplification Rules for Regular Expressions

\( \phi^* = \varepsilon \)

\( \varepsilon^* = \varepsilon \)

\( (\varepsilon + R)^* = R^* \)

\( \varepsilon R = R \varepsilon = R \) \( \varepsilon \) is the identity for concatenation.

\( \phi R = R \phi = \phi \) \( \phi \) is an annihilator for concatenation.

\( \phi + R = R + \phi = R \) \( \phi \) is the identity for union.
Converting DFA to Regular Expressions: Example

A DFA

A GNFA in a special form:
new $R_{SB} = R_{SB} + R_{SA} (R_{AA})^* R_{AB} = \phi + \varepsilon (\phi)^* 0 = 0$
Converting DFA to Regular Expressions: Example

Eliminate B

\[ R_{SC} = R_{SC} + R_{SB} \left( R_{BB} \right)^* R_{BC} = \phi + 0 (0)^* 1 = 00^*1 \]
\[ R_{CC} = R_{CC} + R_{CB} \left( R_{BB} \right)^* R_{BC} = 1 + 0 (0)^* 1 = 1+00^*1 \]
Converting DFA to Regular Expressions: Example Eliminate C

\[ R_{SF} = R_{SF} + R_{SC} (R_{CC})^* \quad R_{CF} = \phi + 00*1 (1+00*1)^* \quad \varepsilon = 00*1 (1+00*1)^* \]

Thus, the regular expression is: \[ 00*1 (1+00*1)^* \]
Converting DFA to Regular Expressions: Example 2

- A DFA

```
start → 0 → A → 1 → B → 0 → F
```

- A GNFA in a special form:

```
S → ε → A → 1 → B → ε → F
```

BBM401 Automata Theory and Formal Languages
Converting DFA to Regular Expressions: Example 2

Eliminate A

\[ R_{SF} = R_{SF} + R_{SA} (R_{AA})^* R_{AF} = \phi + \varepsilon (0)^* \varepsilon = 0^* \]

\[ R_{SB} = R_{SB} + R_{SA} (R_{AA})^* R_{AB} = \phi + \varepsilon (0)^* 1 = 0^*1 \]

\[ R_{BB} = R_{BB} + R_{BA} (R_{AA})^* R_{AB} = \phi + 0 (0)^* 1 = 00^*1 \]

\[ R_{BF} = R_{BF} + R_{BA} (R_{AA})^* R_{AF} = \varepsilon + 0 (0)^* \varepsilon = \varepsilon + 00^* = 0^* \]
Converting DFA to Regular Expressions: Example 2

Eliminate B

$R_{SF} = R_{SF} + R_{SB} \ (R_{BB})^* \ R_{BF} = 0^* + 0^*1 \ (00^*1)^* \ 0^* = 0^* + 0^*1(00^*1)^* \ 0^*$

Thus, the regular expression is: $0^* + 0^*1(00^*1)^*0^*$
We can use the conversion by state elimination algorithm for NFA too.

First, we have to represent the given NFA as a GNFA.

- If the label is a single symbol, the label of the generalized automaton will be that single symbol.
  - $0 \Rightarrow 0$
  - $\varepsilon \Rightarrow \varepsilon$

- If there are more than one symbol, the label will be union (OR) of those symbols.
  - $0,1 \Rightarrow 0+1$
  - $0,1,\varepsilon \Rightarrow 0+1+\varepsilon$
Converting NFA to Regular Expressions: Example

Convert a NFA to a regular expression

Convert a NFA to a GNFA in a special form.
Converting NFA to Regular Expressions: Example

Eliminate A

\[ R_{SB} = R_{SB} + R_{SA} (R_{AA})^* R_{AB} = \phi + \varepsilon (0+1)^* 1 = (0+1)^*1 \]
Converting NFA to Regular Expressions: Example Eliminate B

\[
R_{SC} = R_{SC} + R_{SB} \ (R_{BB})^* \ R_{BC} = \phi + (0+1)^*1 (\phi)^* (0+1) = (0+1)^*1(0+1)
\]
Converting NFA to Regular Expressions: Example

Eliminate C

\[ R_{SD} = R_{SD} + R_{SC} (R_{CC})^* R_{CD} = \phi + (0+1)^*1(0+1) (\phi)^* (0+1) = (0+1)^*1(0+1)(0+1) \]

\[ R_{SF} = R_{SF} + R_{SC} (R_{CC})^* R_{CF} = \phi + (0+1)^*1(0+1) (\phi)^* \epsilon = (0+1)^*1(0+1) \]
Converting NFA to Regular Expressions: Example

Eliminate D

R_{SF} = R_{SF} + R_{SD} \cdot (R_{DD})^* \cdot R_{DF} = (0+1)^*1(0+1) + (0+1)^*1(0+1)(0+1) \cdot (\phi)^* \cdot \varepsilon

= (0+1)^*1(0+1) + (0+1)^*1(0+1)(0+1)

Thus, the regular expression is:  \[(0+1)^*1(0+1) + (0+1)^*1(0+1)(0+1)\]
Regular Languages, DFA, Regular Expressions

Regular Languages $\xrightarrow{\text{DFA}}$ Regular Expressions $\xleftarrow{\text{DFA}}$ Regular Languages

Regular Languages $\xrightarrow{\text{NFA}}$ Regular Expressions $\xrightarrow{\text{DFA}}$ Regular Languages
Algebraic Laws for Regular Expressions
(or Algebraic Laws for Regular Languages)
Algebraic Laws for Regular Expressions

- *Two regular expressions were equivalent* \textit{iff} *they define the same language*.

- \textbf{Algebraic laws} that bring to a higher level the issue of when two regular expressions are equivalent.

- Instead of examining specific regular expressions, we consider pairs of regular expressions with variables as arguments.

- Two regular expressions with variables are equivalent if whatever languages we substitute for the variables, the results of the two expressions are the same language.
Algebraic Laws for Languages – 
**Associativity and Commutativity**

- **Commutativity** is the property of an operator that says we can switch the order of its operands and get the same result.

- **Associativity** is the property of an operator that allows us to regroup the operands when the operator is applied twice.

**Commutative Law for Union:** \( M \cup N = N \cup M \)

- we may take the union of two languages in either order.

**Associative Law for Union:** \((M \cup N) \cup R = M \cup (N \cup R)\)

- we may take the union of three languages either by taking the union of the first two initially or taking the union of the last two initially.

**Associative Law for Concatenation:** \((M N) R = M (N R)\)

- we can concatenate three languages by concatenating either first two or last two initially.

**Concatenation is NOT commutative:** \(MN \neq NM\)
An identity for an operator is a value such that when the operator is applied to the identity and some other value, the result is the other value.

An annihilator for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is the annihilator.

- $\Phi$ is identity for union: $\Phi \cup N = N \cup \Phi = N$
- $\{\varepsilon\}$ is left and right identity for concatenation: $\{\varepsilon\} N = N \{\varepsilon\} = N$
- $\Phi$ is left and right annihilator for concatenation: $\Phi N = N \Phi = \Phi$
A distributive law involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually.

Concatenation is left and right distributive over union:

\[ R (M \cup N) = RM \cup RN \]
\[ (M \cup N) R = MR \cup NR \]

An operator is said to be idempotent if the result of applying it to two of the same values as arguments is that value.

Union is idempotent:

\[ M \cup M = M \]
Algebraic Laws for Languages – 
*Closure Laws*

<table>
<thead>
<tr>
<th>Languages</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi^*$ = ${\varepsilon}$</td>
<td>$\Phi^*$ = $\varepsilon$</td>
</tr>
<tr>
<td>${\varepsilon}^*$ = ${\varepsilon}$</td>
<td>$\varepsilon^*$ = $\varepsilon$</td>
</tr>
<tr>
<td>$L^+$ = $LL^* = L^*L$</td>
<td>$R^+$ = $RR^* = R^*R$</td>
</tr>
<tr>
<td>$L^*$ = $L^+ \cup {\varepsilon}$</td>
<td>$R^*$ = $R^+ + \varepsilon$</td>
</tr>
<tr>
<td>$L^? = L \cup {\varepsilon}$</td>
<td>$R^? = R + \varepsilon$</td>
</tr>
<tr>
<td>$(L^<em>)^</em> = L^*$</td>
<td>$(R^<em>)^</em> = R^*$</td>
</tr>
</tbody>
</table>
Discovering Algebraic Laws for Regular Expressions

- There is an infinite variety of algebraic laws about regular expressions that might be proposed.

- **Methodology**: \( \text{Exp1} = \text{Exp2} \)
  - Replace each *variable* in the law (in \( \text{Exp1} \) and \( \text{Exp2} \)) with *unique symbols* to create concrete regular expressions, \( \text{RE1} \) and \( \text{RE2} \).
  - Check *the equality of the languages of RE1 and RE2*, i.e. \( L(\text{RE1}) = L(\text{RE2}) \)
  - Two regular languages are equal if their DFAs are equal.
Discovering Algebraic Laws for Regular Expressions - Example

Law: \( R(M+N) = RM + RN \)

Replace \( R \) with \( a \), \( M \) with \( b \), and \( N \) with \( c \).

\[ a(b+c) = ab + ac \]

Then, check whether \( L(a(b+c)) \) is equal to \( L(ab+bc) \)

If their languages are equal, the law is TRUE.

Since, \( L(a(b+c)) \) is equal to \( L(ab+bc) \)

\[ R(M+N) = RM + RN \] is a true algebraic law
Discovering Algebraic Laws for Regular Expressions – Example 2

Law: \((M+N)^* = (M*N^*)^*\)

Replace M with a, and N with b.

\((a+b)^* = (a*b^*)^*\)

Then, check whether \(L((a+b)^*)\) is equal to \(L((a*b)^*)\)

Since, \(L((a+b)^*)\) is equal to \(L((a*b^*)^*)\)

\( \Rightarrow (M+N)^* = (M*N^*)^* \) is a true law