Regular Languages, Kleene’s Theorem, Union, Concatenation and Kleene Star

(Based on Hopcroft, Motwani and Ullman (2007) & Cohen (1997))

Regular Languages are defined by regular expressions, Finite Automata and Regular Grammars. Union, Concatenation and Kleene star operations are applicable on regular languages.

Kleene’s Theorem (Based on Cohen (1997))

We have so far introduced three ways to define a language:

1. with a regular expression,
2. with a Finite Automaton (FA), or
3. with a Non-Deterministic FA (NFA) or Transition Graph (TG).

In 1956 Kleene proved that any language that can be defined by one of these methods can be defined by the other two methods as well. This theorem is the most important result in the theory of finite automata.

Our text spends a lot of time on the proof of this theorem. The proof is structured as follows:

1. We prove that every language that can be defined by an FA can be defined by a TG.
2. We prove that every language that can be defined by a TG can be defined by a regular expression.
3. We prove that every language that can be defined by an regular expression can be defined by an FA.

The circular nature of this constructive proof provides us with algorithms to change an example of one of these three structures into either of the other two. Here is a sketch of the proof:

Part 1. Every finite automaton can be turned into a transition graph. This part is easy because every finite automaton is already a transition graph.

Part 2. Every transition graph can be turned into a regular expression. A satisfactory algorithm must work in every case and finish in a finite amount of time. In the next section of these notes there is an algorithm called NFA to regular expression. An NFA is a nondeterministic finite automaton. The only difference between an NFA and a TG is that an NFA allows only a single character per transition. We can convert a TG to an NFA by adding extra states for any transition on which we find a string of length 2 or more, then use the NFA to regular expression algorithm to turn the transition graph into a regular expression.
Part 3. Every regular expression can be turned into a transition graph. In the next section there is an algorithm to turn a regular expression into an NFA and another algorithm to remove the nondeterminism from an NFA and turn it into a DFA (deterministic finite automaton), the kind of finite automaton we are familiar with already.

Nondeterminism

We define a nondeterministic finite automaton (NFA) as a finite automaton in which we allow more than one transition out of a state with the same label and we allow $\Lambda$-transitions. The author of our text defines an NFA slightly differently in that he does not allow $\Lambda$-transitions.

Algorithms

NFA to DFA

Given: a nondeterministic finite automaton that accepts language $L$

Output: a (deterministic) finite automaton that accepts language $L$

1. Label the states.
2. Make a transition table for the NFA: Label the rows of the table with the names of the states, and the columns with the characters of the alphabet. As the entry for cell $(s,c)$, list the states that can be reached from state $s$ by consuming character $c$. Also include any states that can be reached from state $s$ by using $\Lambda$-transitions after consuming character $c$.
3. Determine the start state of the FA by combining the start state of the NFA with any states reachable from this start state using $\Lambda$-transitions. On the above example, the start state of the FA should be $(1,2,6,8)$ since the start state of the NFA is 1 and from state 1 you can reach states 2, 6, and 8 on $\Lambda$-transitions.
4. Make a transition table for the FA one row at a time. Label the columns of this table with the characters of the alphabet and label row #1 with the start state determined in step 3.
Determine the entries for row $r$ as follows: For each character $c$ of the alphabet and for each individual state $s$ in the label of row $r$, add the entries in cell $(s, c)$ of the NFA transition table to cell $(r, c)$ in the FA transition table, except do not include any entry more than once.

Once row $r$ is completed, let the entries in each cell be the name of a state in the FA. If that state has not been previously listed in the table, add a new row with that state as label.

Continue this process until no new states are added.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,6,8</td>
<td>2,6,8</td>
<td>3,4,7,8</td>
</tr>
<tr>
<td>2,6,8</td>
<td>2,6,8</td>
<td>3,4,7,8</td>
</tr>
<tr>
<td>3,4,7,8</td>
<td>2,6,8</td>
<td>5,8</td>
</tr>
<tr>
<td>5,8</td>
<td>2,6,8</td>
<td>-</td>
</tr>
</tbody>
</table>

5. Draw the FA making any state whose name contains the name of an original accept state also be an accept state.

**Non-deterministic Finite Automaton (NFA) to Regular expression.**

Prove: For every Non-deterministic Finite Automaton (NFA) there is a Regular expression. ( NFA $\Rightarrow$ Regular Expression )

Any arbitrary NFA can be reduced to the regular expression it defines, following the steps outlined below:

1. Add a new start state to the given NFA and add a null-transition from this new start state to the original start state of the NFA. Change the status of the original start state, so that it is no longer a start state.
2. Add a new final (accept) state and add null-transitions from the original final states to this new final state. Change the status of the original accept states so that they are no longer accept states. The new accept state must be different from the new start state.

Consider the NFA of Figure 1. After we make the changes suggested above the NFA of Figure 2 is resulted from that of Figure 1.
3. Construct a new graphical representation from the above NFA according to the following method. Traverse each route from the start state to the final state of the above NFA and write down the expression obtained by concatenation of the expressions of the transitions of the route on a single arrow between the start and the final states of the new graphical representation. For every loop use Kleene star in the expression.
4. Unionize the regular expressions of the arrows of the new graphical representation. The resulting regular expression defines the language represented by the original NFA. Therefore, the given NFA is reduced to its equivalent regular expression.

1. The graphical representation of Figure 3 is not an NFA according to the definition of NFA; however, it is a generalized NFA according to Sipser (2006) or a Transition Graph according to Cohen (1997)

---

**NFA to regular expression** (another alternative)

**Given:** a finite automaton that accepts language L

**Output:** a regular expression that generates language L

5. Add a new start state and add an edge labeled $\square$ from this new start state to each of the original start states. The original start states are no longer start states.

6. Add a new accept state and $\square$-transitions from the original accept states to this new accept state. The original accept states are no longer accept states. The new accept state must be different from the new start state.

7. Combine any edges that exit and enter the same states:

   ![Diagram](image.png)

8. Eliminate any states that have no edges going out to other states. Eliminate any edges to these states as well.

9. Repeat until the only remaining states are the start state and the accept state:

   Select a state $s$ to be eliminated. For each entering edge $(r,s)$ and for each leaving edge $(s,t)$ create a bypass edge $(r,t)$. If there is a self-loop $(s,s)$ on state $s$ then label edge $(r,t)$ with the concatenation of the label on $(r,s)$ with the *-closure of the label on $(s,s)$ and with the label on $(s,t)$. Else, label edge $(r,t)$ with the concatenation of the label on $(r,s)$ with the label on $(s,t)$. Eliminate edge $(r,s)$. Once all entering edges to $s$ have been eliminated, eliminate state $s$ and all its leaving edges. Combine any edges that exit and enter the same states as in step 3.
10. Combine all edges from the start state to the accept state as in step 3. The regular expression on the resulting edge is the output of this algorithm.

**regular expression to NFA**

**Given:** a regular expression that generates language \( L \)

**Output:** a nondeterministic finite automaton that accepts language \( L \)

1. Draw a start state and an accept state and connect them with an edge labeled with the regular expression given.
2. Recursively apply the following rules until each edge of the finite automaton contains a single character of the alphabet. Apply the rules to an edge in the order shown:
   a. If an edge is labeled with an expression that consists of two or more subexpressions "or-ed" together, add new edges so that each subexpression is on an edge of its own. However, if each subexpression consists of a single character of the alphabet, you may simply change the '+'s in the expression into commas.
   
   ![Diagram of regular expression to NFA rule a](image)

   b. If an edge is labeled with an expression that consists of two or more subexpressions concatenated together, add intermediate states and edges to put each subexpression on its own edge.
   
   ![Diagram of regular expression to NFA rule b](image)

   c. If an edge \((m, n)\) is labeled with the Kleene-closure of an expression, add intermediate state \( o \) and \( \square \)-transitions \((m, o)\) and \((o, n)\). Remove the Kleene star from the expression and use the starless expression as the label on a
Minimizing a DFA

**Given:** a finite state machine that accepts language L

**Output:** the smallest finite state machine that accepts language L

1. Add a trap state to the machine if it contains an implied trap state and add all the edges that go to the trap state.
2. Create a stair-step diagram like the one shown below. Suppose the states of the machine are named 1..n. Then label the rows of the diagram 2..n and the columns 1..n-1 (label them along the bottom.) Note that there is exactly one cell in the diagram for each possible pair of states. The reduction of the machine is achieved by combining any states which are serving the same purpose in the machine, and the cells of the diagram will be marked according to whether that pair of cells can be combined or not.

3. Draw an X through any cell that pairs an accept state with a non-accept state.
4. Recursively consider the individual cells of the diagram until every cell contains either a checkmark or an X. For cell (i,j) do the following:
   
   If states i and j have transitions going to the same states for each character of the alphabet, put a checkmark in cell (i,j).

   Else if for some character of the alphabet state i has a transition to an accept state while state j has a transition to a non-accept state, or vice versa, put an X in cell (i,j).
Else consider each character of the alphabet separately. If state \( i \) has a transition to state \( m \) and state \( j \) has a transition to state \( n \) for character \( c \), then write \( m, n \) in cell \( (i, j) \). Write such a pair for any character for which states \( i \) and \( j \) have transitions to two different states. If at some point in the processing all of the cells representing the different pairs written in cell \( (i, j) \) contain checkmarks, then put a checkmark in cell \( (i, j) \). If at some point in the processing one of the cells representing a pair in cell \( (i, j) \) contains an \( X \), then put an \( X \) in cell \( (i, j) \).

5. Combine all states whose cells contain checkmarks. Examples:

Suppose cell \( (3, 4) \) contains a checkmark. Then the minimized FA will contain one state named \( (3, 4) \) instead of having the two different states 3 and 4. All of the transitions into the original state 3 and all of the transitions into the original state 4 now come into state \( (3, 4) \), and the same for the transitions that left from the original states. A transition from the original 3 to the original 4, or vice versa, will now be a self-loop on state \( (3, 4) \).

Suppose cells \( (3, 4) \), \( (4, 5) \) and \( (3, 5) \) contain checkmarks. Then the minimized FA will contain one state named \( (3, 4, 5) \) instead of having the three different states 3, 4, and 5. The same rules about transitions apply to the combination of three or more states as applied to a combination of two states.

**Minimizing a DFA**

**Given:** a finite state machine that accepts language \( L \)

**Output:** the smallest finite state machine that accepts language \( L \)
1. Add a trap state to the machine if it contains an implied trap state and add all the edges that go to the trap state.

2. Divide the set of states into two subsets: accept states and non-accept states.

\{0, 1, 3, 5\} \{2, 4\} This is line 1.

3. Line 2 will be another partitioning of the original set. Two states can remain in the same subset only if the transitions leaving those states (on each letter of the alphabet) both go to states that are in the same subset on the previous line.

a. Take the first subset above, \{0, 1, 3, 5\}. Remove the 0 and place it in a subset of its own. Then take the 1 and look to see if it can be placed in the subset with the 0 or if it has to go to a different subset. On the letter a, 0 goes to 1 while 1 goes to 2. States 1 and 2 are in different subsets on line 1, so we cannot place 1 with 0. Put 1 in a subset of its own.

b. Now take 3. On the letter a, 0 goes to 1 while 3 goes to 4. 1 and 4 are in different subsets on line 1, so we cannot place 3 with 0. Can we place it with 1? On the character a, 1 goes to 2 and 3 goes to 4. Since 2 and 4 are in the same subset on line 1, we’re good so far. On the character b, 1 goes to 3 and 3 goes to 1. Since 1 and 3 are also in the same subset, we can put 3 into 1’s subset. Now take 5 and see if we can put it with 0. On the character a, 0 goes to 1 and 5 goes to 5. Since 0 and 5 are in the same subset on line 1, we may be able to put 5 with 0. On the character b, 0 goes to 3 and 5 goes to 5. So 5 can be placed with 0. Thus we have broken \{0, 1, 3, 5\} into two subsets: \{0, 5\} \{1, 3\}.

c. Now see if 2 and 4 can stay together or if they have to be broken apart. On the letter a, 2 goes to 5 and 4 goes to 5. On the letter b, 2 goes to 2 and 4 goes to 4. Since 2 and 4 are in the same subset on line 1, we can leave 2 and 4 in the same subset. Thus line 2 is \{0, 5\} \{1,3\} \{2, 4\}.

4. We continue this process until no change occurs from one line to the next. Then we make one state for each of the subsets in the final partitioning. In the above example we end up with \{0\} \{1,3\} \{2,4\} \{5\}.
Chapter 8 - Finite Automata With Output

Finite automata are like computers in that they receive input and process the input by changing states. The only output that we have seen finite automata produce so far is a yes/no at the end of processing. We will now look at two models of finite automata that produce more output than a yes/no.

Moore Machines

A Moore machine is like a finite automaton except for the following differences. In a Moore machine there are two alphabets: an input alphabet and an output alphabet. The two alphabets may be the same but they do not have to be. Another difference is that there are no accept states in a Moore machine. Its purpose is not to answer yes or no, not to accept or reject a string. It is not a language recognizer, it is an output producer. Each state of a Moore machine produces a one-character output immediately upon the machine's entry into that state. At the beginning, the start state produces an output before any input has been read. Thus the output of a Moore machine is one character longer than its input.

We draw Moore machines in the same way as finite automata but the label in a state is composed both of the name of the state and the output character that the state produces. Run the string abab through the following machine and you will find that the output produced is 10010.

![Diagram of Moore machine]

Input: abab
Output: 10010

The following Moore machine might be considered a "counting" machine. The output produced by the machine contains a 1 for each occurrence of the substring aab found in the input string.
A **Mealy machine** produces output on a transition instead of on entry into a state. Transitions are labeled $i/o$ where $i$ is a character in the input alphabet and $o$ is a character in the output alphabet. Mealy machine are complete in the sense that there is a transition for each character in the input alphabet leaving every state. There are no accept states in a Mealy machine because it is not a language recognizer, it is an output producer. Its output will be the same length as its input.

**Example:**

The following Mealy machine takes the one's complement of its binary input. In other words, it flips each digit from a 0 to a 1 or from a 1 to a 0.
The next Mealy machine from page 154 of the text increments its binary input. The only rather disconcerting characteristic of the machine is that we must feed the input number backwards and the machine produces its output backwards. It also does not work correctly if the input string consists completely of 1’s. In that case the answer always comes out 0.

Here is an example of a Mealy machine that reports on the parity of each 4-bit substring in its input. For each of the first 3 bits of each 4-bit substring, the machine outputs a 0. If a 4-bit substring contains an even number of 1’s then the machine outputs a 0 on the 4th bit of that 4-bit substring, otherwise it outputs a 1.
Although Moore and Mealy machines do not accept or reject their input strings, they do yield information about their input through the output that they produce. Here is a Mealy machine to count the number of occurrences of aa or bb. It produces a 1 each time it finds that it has just seen a double letter.

When we talk about equivalence of two Moore machines or two Mealy machines we mean that, given the same input, they produce the same output. Since a Moore machine outputs the symbol associated with its start state before it begins processing its input, its output is always one longer than its input. The output of a Mealy machine is always the same length as its input. Therefore a Moore machine cannot be equivalent to a Mealy machine in the above sense. We say that a Moore machine is equivalent to a Mealy machine if, given the same input, the output of the Moore machine after removing the first character is the same as the output of the Mealy machine.
Using this definition of equivalence, our text proves that for every Moore machine there is an equivalent Mealy machine and vice versa. It does this with two constructive algorithms: one for converting a Moore machine to a Mealy machine and one for going the other direction. We will not study these algorithms because we need to move on to other material, but they are interesting. Read about them in the text.
Chapter 9 - Regular Languages

A language that can be defined by a regular expression is called a **regular language**. Are all languages regular? No, but we will postpone this discussion until the next chapter. In this chapter we will review some properties we already know about regular languages.

**Union, Concatenation, Kleene Star**

**Theorem.** The set of regular languages is closed under union, concatenation, and the Kleene star.

**Proof.** This theorem is easy to prove using regular expressions. Take any two regular languages \( L_1 \) and \( L_2 \) from the set of regular languages. Each of them has a regular expression corresponding to it. Call the two regular expressions \( R_1 \) and \( R_2 \). It is easy to form a regular expression that corresponds to the union of the two languages by putting a \( + \) between the two regular expressions. Likewise we can concatenate the two regular expressions or put parentheses around one and place a star outside the parentheses. These operations form regular expressions that correspond to the concatenation and the Kleene star of the languages. **QED**

We can also prove the above theorem using Transition Graphs (TG's) or Nondeterministic Finite Automata (NFA). We will take the parts of the proof one at a time. First we prove the following:

If \( L_1 \) and \( L_2 \) are regular languages then their union, \( L_1 + L_2 \), is also regular.

Since \( L_1 \) and \( L_2 \) are regular languages, we know that we can draw a TG or an NFA for each of them. Call these TG's or NFA's \( M_1 \) and \( M_2 \). To form the union of the two languages we can create a **union machine** by creating a new start state with \( \Lambda \)-transitions from this new state to each of the start states of \( M_1 \) and \( M_2 \). Any language for which you can draw a TG is a regular language, so the union of \( L_1 \) and \( L_2 \) is regular.
We can form a TG or NFA for the concatenation of two languages as shown in the next diagram. We first make sure that machine $M_1$ has only one accept state. If it has more than one, we take away the accept status of all its accept states, draw a new accept state and add $\lambda$-transitions from each of the original accept states to this new accept state. Then we add a $\lambda$-transition from the accept state of the first machine to the start state of the second.

![Diagram showing the concatenation of two languages](image)

This new TG or NFA defines the concatenation of $L_1$ and $L_2$ ($L_1L_2$); therefore the $L_1L_2$ is a regular language.

Finally, we can make a TG or an NFA that accepts the Kleene closure of the language accepted by machine $M_1$ as shown below. We make sure that $M_1$ has only one start state and one accept state; otherwise we modify $M_1$ so that it has one start state and one accept state. Then we add a new start state and draw a $\lambda$-transition from this new state to the start state of $M_1$. Then we add another $\lambda$-transition from the new start state to the accept state and one more $\lambda$-transition from the accept state to the new start state.

![Diagram showing the Kleene closure](image)

**COMPLEMENT**

If $L$ is a language over alphabet $\Sigma$, we define the complement, $L'$, to be the language of all strings over $\Sigma$ that are not in $L$. That is, $L' = \sum^* - L$.
**Theorem.** The set of regular languages is closed under complementation. That is, if $L$ is a regular language then its complement, $L'$, is also a regular language.

**Proof.** If $L$ is a regular language, then there must be a DFA that defines $L$; let us call it DFA$_1$. A DFA for $L'$, DFA$_2$, can be constructed by changing all accept states to non-accept states and all non-accept states to accept states in DFA$_1$. This new machine, DFA$_2$, accepts all strings that are rejected by DFA$_1$ and it rejects all strings that are accepted by DFA$_1$. DFA$_2$ accepts $L'$, therefore $L'$ is regular. QED

**Examples demonstrating the proof:**

A DFA that accepts the language $L = \{a, ab\}$. $L$ contains only two strings: $a$ and $ab$

A DFA that accepts $L'$. That is, it accepts all strings other than $a$ and $ab$
Another example complement is given below:

A DFA that accepts the language \( L = \{ \text{ab} \} \). \( L \) contains only one string: \( \text{ab} \)

A DFA that accepts \( L' \). That is, it accepts all strings other than \( \text{ab} \):
Intersection

Theorem. The set of regular languages is closed under intersection. That is, if \( L_1 \) and \( L_2 \) are regular languages then \( L_1 \cap L_2 \) is a regular language. 

Proof. \( L_1 \) and \( L_2 \) are sets so we can use set theory in our proof. By DeMorgan’s Law we have the following:

\[
L_1 \cap L_2 = (L_1' \cup L_2')'.
\]

Since \( L_1 \) and \( L_2 \) are regular, so are \( L_1' \) and \( L_2' \). So is the union of \( L_1' \) and \( L_2' \) and so is the complement of that union. QED

It is useful to know how to build a finite automaton to accept the intersection of two languages accepted by finite automata \( M_1 \) and \( M_2 \). Let’s find a finite automaton to accept strings that start with a and end with a. This language is the intersection of the two languages of strings that start with a, and strings that end with a. Here are finite automata for each of those languages. Note that the trap state in the first one was made explicit. This is a necessity for our algorithm.

We want to find a finite automaton that accepts strings that are in both languages. We build the following table, the start state of which is the combination of the start states of the above two machines. To determine the table entry for row \((1,A)\) and character a, we combine the transition on character a from state 1 in the first machine with the transition on character a from state A in the second machine. We do this for character b as well. Then we make a row for any combined state that we produced and for which we do not already have a row. We continue this process until we have a row for every combined state that is produced by the algorithm. Note that the only accept states in this new machine will be states that are combinations of accept states from the original two machines.
The machine above can be simplified to the machine below because states 2A and 2B are acting together as a trap state.

Pumping Lemma and Non-regular Languages

The pumping lemma is used for proving that a language is NOT regular. It cannot be used for proving that a language is regular.

Example: Consider the following FA having 5 states:
Let's process the string ababbaa on the FA:
1 a ! 2 b ! 4 a ! 3 b ! 5 b ! 4 a ! 3 a ! 2

Since 2 is a final state, we accept the string ababbaa.

In general,

- We always start in initial state.
- After reading first letter of input string, we end may go to another state or return to initial state.
- the maximum number of different states that we could have visited after reading the first letter is 2.
- After reading the first 2 letters of input string, the maximum number of different states that we could have visited is 3.
- In general, after reading the first m letters of input string, the maximum number of different states that we could have visited is m + 1.

In our example above, after reading 5 letters, the maximum number of different states that we could have visited is 5 + 1 = 6. But since the FA has 5 states, we know that after reading in 5 letters, we must have visited some state twice.

Consider the string aaabaa.
The string has length 6, which is more than the number of states in the above FA.

We process the string as follows:
1 a − 2 a − 1 a − 2 b − 4 a − 3 a − 2
and so it is accepted.

Notice that state 1 is the first state that we visit twice.
In general, if we have an FA with N states and we process a string w with length(w) ≤ N, then there exists at least one state that we visit at least twice.
Let u be the first state that we visit twice.
Break up string w as w = xyz, where x, y, and z are 3 strings such that the string x is the letters at the beginning of w that are read by the FA until the state u is hit for the first time. The string y is the letters used by the FA starting from the first time we are in state u until we hit state u the second time. The string z is the rest of the letters in w.

For example, for the string w = ababbaa processed on the above FA, we
have \( u = 2 \), and \( x = ab \), \( y = abb \), \( z = aa \).

- For example, for the string \( w = aaabaa \) processed on the above FA, we have \( u = 1 \), and \( x = _{-} \), \( y = aa \), \( z = abaa \).

Definition: A language that cannot be defined by a regular expression is called a nonregular language.

By Kleene’s Theorem, a nonregular language cannot be accepted by any FA or TG.

Consider

\[
L = \{ \_ , ab, aabb, aaabbb, \ldots \}
\]

\[= \{a^n b^n : n = 0, 1, 2, \ldots \} \_ \{a^n b^n\}\]

- We will show that \( L \) is a nonregular language by contradiction.
- Suppose that there is some FA that accepts \( L \).
- By definition, this FA must have a finite number of states, say 5.
- Consider the path the FA takes on the word \( a6b6 \).
- The first 6 letters of the word are a’s.
- When processing the first 6 letters, the FA must visit some state \( u \) at least twice since there are only 5 states in the FA.
- We say that the path has a circuit, which consists of those edges that are taken from the first time \( u \) is visited to the second time \( u \) is visited.
- Suppose the circuit consists of 3 edges.
- After the first b is read, the path goes elsewhere and eventually we end up in a final state where the word \( a6b6 \) is accepted.
- Now consider the string \( a6+3b6 \).
- When processing the a part of the string, the FA eventually hits state \( u \).
- From state \( u \), we can take the circuit and return to \( u \) by using up 3 a’s.
- From then on, we read in the rest of the a’s exactly as before and go on to read in the 6 b’s in the same way as before.
- Thus, when processing \( a6+3b6 \), we end up again in a final state.
- Hence, we are supposed to accept \( a9b6 \).
- However, \( a9b6 \) is not in \( L \) since it does not have an equal number of a’s and b’s.
- Thus, we have a contradiction, and so \( L \) must not be regular.
- We can use the same argument with any string \( a6(a3)^k b6 \), for \( k = 0, 1, 2, \ldots \).
Theorem 13 (Pumping Lemma) Let $L$ be any regular language that has infinitely many words. Then there exists some three strings $x$, $y$, and $z$ such that $y$ is not the null string and that all strings of the form $xy^kz$ for $k = 1, 2, 3 \ldots$ are words in $L$.

Proof.
Since $L$ is a regular language, there exists some FA that accepts $L$ by Kleene’s theorem. FA must have a finite number of states $N$.
Since $L$ is an infinite language and since alphabets are always finite, $L$ must consist of arbitrarily long words.
Consider any word $w$ accepted by FA with $\text{length}(w) = m$, and assume that $m \geq N$.
Since $\text{length}(w) = m$, when processing $w$ on the FA, we visit $m + 1$ states, not necessarily all unique.
Since $m + 1 \geq N + 1$, when processing $w$ on the FA, we visit at least $N + 1$ states.
But since the FA has only $N$ states in total, some state must be visited twice when processing $w$ on the FA.
Let $u$ be the first state visited twice when processing the string $w$ on the FA.
Thus, there is a circuit in FA corresponding to state $u$ for this string $w$.
We break up $w$ into three substrings $x, y, z$:
- $x$ consists of the all letters starting at the beginning of $w$ up to those consumed by the FA when state $u$ is reached for the first time. Note that $x$ may be null.
- $y$ consists of the letters after $x$ that are consumed by the FA as it travels around the circuit.
- $z$ consists of the letters after $y$ to the end of $w$.

Note that the following statements hold:
- $w = xyz$.
  1. Note that $y$ is not null since at least one letter is consumed by traveling around the circuit. The circuit starts in a particular state, and ends in the same state. Thus, traveling the circuit requires at least one transition, which means that at least one letter is consumed.
  2. The strings $x$ and $y$ satisfy $\text{length}(x) + \text{length}(y) \geq N$, which we can show as follows. Let $v$ be the string $xy$ except for the last letter of $xy$. By the way that we constructed $x$ and $y$, when we process $v$ on the FA starting in the initial state, we never visit any state twice since it is only on reading the last letter of $y$ do we first visit some state twice. Thus, processing $v$ on the FA results in visiting at most $N$ states, which corresponds to reading at most $N - 1$ letters.

Since $xy$ is the same as $v$ with one more letter attached, we must have that $xy$ has length at most $N$.
When processing $w = xyz$, the FA first processes substring $x$ and ends in state $u$.
Then it starts processing substring $y$ starting in state $u$ and ends in state $u$.
Then it starts processing substring $z$ starting in state $u$ and ends in some final state $v$.
Now process the word $xyyz$ on FA.

For the substring $x$, the FA follows exactly the same path as when it processed the $x$-part of $w$. For the first substring $u$, the FA starts in state $u$ and returns to state $u$. For the second substring $u$, the FA starts in state $u$ and returns to state $u$. For the substring $z$, the FA starts in $u$ and processes exactly as before for the word $w$, and so it ends in the final state $v$. Thus, $xyyz$ is accepted by FA.
Similarly, we can show that any string $xy^kz$, $k = 1, 2, 3 \ldots$, is accepted by FA.

---

**Question:**

Is $L_2 = \{ a^n b^n : \text{where } n \geq 0 \} = \{^a, ab, aabb, aaabbb, aaaaabbb, \ldots \}$ a Regular Language? Prove your answer.

**Answer:** No, $L_2$ is not a Regular Language.

Proof: We apply the Pumping Lemma to prove that $L_2$ is not Regular. Take a long enough string $w = aabb$ from $L_2$ and decompose it into 3 parts $xyz$ such that $y$ is not null. According to the lemma every string of the form $xy^n z$, where $n > 0$ would be in $L_2$. We try the following two decompositions of $w = aabb$ in order to apply the pumping lemma.

<table>
<thead>
<tr>
<th>Decomposition-1</th>
<th>Decomposition-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xyz = a</td>
<td>a</td>
</tr>
<tr>
<td>$xy^2 z = a</td>
<td>aa</td>
</tr>
<tr>
<td>$xy^3 z = a</td>
<td>aaa</td>
</tr>
</tbody>
</table>

In the above cases, for $xy^2 z$, we obtained $aaabb$ and $aabbb$ which are not in $L_2$. Similarly, for $xy^3 z$, we obtained $aaaabb$ and $aabbbb$ which are not in $L_2$. From the values of $xy^2 z$ and $xy^3 z$ in the above two decompositions, it is clear that no decomposition of $w$ would allow pumping which proves that $L_2$ is not regular.

---

**Another Version of Pumping Lemma**

Theorem 14 Let $L$ be a language accepted by an FA with $N$ states. Then for all words $w \in L$ such that length($w$) $\leq N$, there are strings $x$, $y$, and $z$ such that

1. $w = xyz$;
2. $y$ is not null;
Proof. The proof of Theorem 13 actually establishes Theorem 14.
Remarks: In the textbook Theorem 14 also assumes that \( L \) is infinite. However, this additional assumption is not needed.

Example:

\[ w = ababbaa \]
\[ x = ab \]
\[ y = abb \]
\[ z = aa \]

Example: Prove \( L = \text{PALINDROME} \) is nonregular.

We cannot use the first version of the Pumping Lemma (Theorem 13) since \( x = a, y = b, z = a, \) satisfy the lemma and do not contradict the language since all words of the form \( xy^kz = abka \) are in \( \text{PALINDROME} \).

We will instead apply Theorem 14 to show that \( \text{PALINDROME} \) is nonregular.

Proof.

- Suppose that \( \text{PALINDROME} \) is a regular language.
- Then by definition, \( \text{PALINDROME} \) must have a regular expression.
- Kleene’s Theorem then implies that there is a finite automaton for \( \text{PALINDROME} \).
- Assume that the FA for \( \text{PALINDROME} \) has \( N \) states, for some \( N \geq 1 \).
- Consider the string
  \[ w = aNbaN \]
  which is in \( \text{PALINDROME} \).

Note that \( \text{length}(w) = 2N + 1 \geq N \).

Thus, all of the assumptions of Theorem 14 hold, so the conclusions of Theorem 14 must hold; i.e., there exist strings \( x, y, \) and \( z \) such that

- P1. \( w = xyz \);
- P2. \( y \) is not null;
- P3. \( \text{length}(x) + \text{length}(y) \geq N \);
- P4. \( xy^kz \in L \) for all \( k = 0, 1, 2, \ldots \)
• P1 of Theorem 14 says that \( w = xyz \), so \( x \) must be at the beginning of \( w \), \( y \) must be somewhere in the middle of \( w \), and \( z \) must be at the end of \( w \). P2 of Theorem 14 says that \( x \) and \( y \) together have at most \( N \) letters. Since \( w \) has \( N \) a’s in the beginning and \( x \) and \( y \) are at the beginning of \( w \), \( x \) and \( y \) must consist solely of a’s. P1 and P3 of Theorem 14 imply that \( x \) and \( y \) must consist solely of a’s. Since \( z \) is the rest of the string after \( x \) and \( y \), we must have that \( z \) consists of zero or more a’s, followed by 1 b and then \( N \) a’s.

In other words,
\[
\begin{align*}
x &= a^i \\
y &= a^j \\
z &= a^l b a^N
\end{align*}
\]
Since \( y \neq _b \) by P2 of Theorem 14, we must have \( j \geq 1 \).

Also, since \( w = xyz \) by P1 of Theorem 14, note that
\[
\begin{align*}
w &= a^N b a^N = xyz = a a^j b a^N = a^{i+j} b a^N, \\
\text{so } &i + j + 1 = N.
\end{align*}
\]
Now consider the string \( xyyz \), which is supposed to be in \( L \) by P4 of Theorem 14.

Note that
\[
\begin{align*}
xyyz &= a a^j b a^N = a^{i+2j} b a^N = a^{N+j} b a^N, \\
\text{since } &i + j + 1 = N.
\end{align*}
\]
But \( a^{N+j} b a^N \neq \text{PALINDROME} \) since \( \text{reverse}(a^{N+j} b a^N) \neq a^{N+j} b a^N \).
This is a contradiction, and so \( \text{PALINDROME} \) must be nonregular.

Can use first version of Pumping Lemma (Theorem 13) to show that \( L = \{a^n b^n : n \geq 0\} \) is not a regular language:

Suppose \( L \) is a regular language.

Pumping Lemma says that there exist strings \( x, y, \) and \( z \) such that all words of the form \( xy^k z \) are in \( L \), where \( y \) is not null.

All words in \( L \) are of the form \( a^n b^n \).

How do we break up \( a^n b^n \) into substrings \( x, y, z \) with \( y \) nonempty?

If \( y \) consists solely of a’s, then \( xyyz \) has more a’s than b’s, and so it is not in \( L \).

If \( y \) consists solely of b’s, then \( xyyz \) has more b’s than a’s, and so it is not in \( L \).

If \( y \) consists of a’s and b’s, then all of the a’s in \( y \) must come before all of the b’s.
However, \( xyyz \) then has some b’s appearing before some a’s, and so \( xyyz \) is not in \( L \).
Thus, \( L \) is not a regular language.

Example:
- Let \( _a = \{a, b\} \).

  For any string \( w \) defined, define \( na(w) \) to be the number of a’s in \( w \), and \( nb(w) \) to be the number of b’s in \( w \).

  Define the language \( L = \{w \in _a^{\ast} : na(w) \geq nb(w)\} \); i.e., \( L \) consists of strings \( w \) for which the number of a’s in \( w \) is at least as large as the number of b’s in \( w \).

  For example, abbaa \( 2 L \) since the string \( 3 \) a’s and \( 2 \) b’s, and \( 3 \geq 2 \).

  We can prove that \( L \) is a nonregular language using the pumping lemma.

  What string \( w \) \( 2 L \) should we use to get a contradiction?
Example: Consider the language \( \text{EQUAL} = \{\_, \text{ab}, \text{ba}, \text{aabb}, \text{abab}, \text{abba}, \text{baba}, \text{bbab}, \ldots\} \), which consists of all words having an equal number of a's and b's. We now prove that \( \text{EQUAL} \) is a non-regular language.

- We will prove this by contradiction, so suppose that \( \text{EQUAL} \) is a regular language.
- Note that \( \{\text{a}^n \text{b}^n : n \geq 0\} = \text{a}_+ \text{b}_+ \setminus \text{EQUAL} \)
- Recall that the intersection of two regular languages is a regular language.
- Note that \( \text{a}_+ \text{b}_+ \) is a regular expression, and so its language is regular.
- If \( \text{EQUAL} \) were a regular language, then \( \{\text{a}^n \text{b}^n : n \geq 0\} \) would be the intersection of two regular languages.
- This would imply that \( \{\text{a}^n \text{b}^n : n \geq 0\} \) is a regular language, which is not true.
- Thus, \( \text{EQUAL} \) must not be a regular language.

### Prefix Languages

**Definition:** If \( R \) and \( Q \) are languages, then \( \text{Pref}(Q \text{ in } R) \) is the language of “the prefixes of \( Q \) in \( R \),” which is the set of all strings of letters that can be concatenated to the front of some word in \( Q \) to produce some word in \( R \); i.e.,

\[
\text{Pref}(Q \text{ in } R) = \{ \text{strings } p : \exists q \in Q \text{ such that } pq \in R \}
\]

**Example:** \( Q = \{\text{aba, aabb, baaba, bbbaaabb, aaaa}\} \)
\( R = \{\text{baabaaba, aabb, abbabbaaaabb}\} \)

\( \text{Pref}(Q \text{ in } R) = \{\text{baaba, _}, \text{abbabba, abba}\} \)

**Example:** \( Q = \{\text{aba, aabb, baaba, bbbaaabb, aaaa}\} \)
\( R = \{\text{baab, ababb}\} \)

\( \text{Pref}(Q \text{ in } R) = ; \)

**Example:** \( Q = \text{ab}_+ \_ \text{a} \)
\( R = \{\text{ba}\}_+ \_ \text{b} \)

\( \text{Pref}(Q \text{ in } R) = \{\text{ba}\}_+ \_ \text{b} \)

**Theorem 16** If \( R \) is a regular language and \( Q \) is any language whatsoever, then the language

\[
\text{P} = \text{Pref}(Q \text{ in } R)
\]

is regular.

**Proof:** Since \( R \) is a regular language, it has some finite automaton \( \text{FA}_1 \) that accepts it.

- \( \text{FA}_1 \) has one start state and several (possibly none or one) final states.
- For each state \( s \) in \( \text{FA}_1 \), do the following:
  - Using \( s \) as the start state, process all words in the language \( Q \) on \( \text{FA}_1 \).
  - When starting \( s \), if some word in \( Q \) ends in the final state of \( \text{FA}_1 \), then paint state \( s \) blue.
• So for each state s in FA1 that is painted blue, there exists some word in Q that can be processed on FA1 starting from s and end up in a final state.
• Now construct another machine FA2:
  • FA2 has the same states and arcs as FA1.
  • The start state of FA2 is the same as that of FA1.
  • The final states of FA2 are the ones that were previously painted blue (regardless if they were final states in FA1).
• We will now show that FA2 accepts exactly the prefix language P = Pref(Q in R).
• To prove this, we have to show two things:
  • Every word in P is accepted by FA2.
  • Every word accepted by FA2 is in P.
  • First, we show that every word accepted by FA2 is in P.

Consider any word w accepted by FA2. Starting in the start state of FA2, process the word w on FA2, and we end up in a final state of FA2. Final states of FA2 were painted blue. Now we can start from here and process some word from Q and end up in a final state of FA1. Thus, the word w is in P.

Now we prove that every word in P is accepted by FA2.

Consider any word p in P. By definition, there exists some word q in Q and a word w in R such that pq = w. This implies that if pq is processed on FA1, then we end up in a final state of FA1. When processing the string pq on FA1, consider the state s we are in just after finishing processing p and at the beginning of processing q. State s must be a blue state since we can start here and process q and end in a final state. Hence, by processing p, we must start in the start state and end in state s. Thus, p is accepted by FA2.

Other References:

Chapter 11 Decidability

Introduction
We have three basic questions to answer:
1. How can we tell if two regular expressions define the same language?
2. How can we tell if two FA’s are equivalent?
3. How can we tell if the language defined by an FA has finitely many or infinitely many words in it?
4. Note that questions 1 and 2 are essentially the same by Kleene’s Theorem.

Decidable Problems

Definition: A problem is effectively solvable if there is an algorithm that provides the answer in a finite number of steps, no matter what the particular inputs are (but may depend on the size of the problem).

The maximum number of steps the algorithm will take must be predictable before we begin executing the procedure.

Example: Problem: find roots of quadratic equation $ax^2 + bx + c = 0$.
Solution: use quadratic equation
$$x = -b \pm \sqrt{b^2 - 4ac}$$

No matter what the coefficients $a$, $b$, and $c$ are, we can compute the solution using the following operations:
- four multiplications
- two subtractions
- one square root
- one division

Another solution: keep guessing until we find a root.
This approach is not guaranteed to find root in a fixed number of steps.
Example: Find the maximum of $n$ numbers. An effective solution for this is to scan through the list once while updating the maximum observed thus far. This takes $O(n)$ steps.

Definition: An effective solution to a problem that has a yes or no answer is called a decision procedure. A problem that has a decision procedure is called decidable.

Is $L_1 = L_2$?

Determine if two languages $L_1$ and $L_2$ are the same:

Method 1: Check if the language
$$L_3 = (L_1 \setminus L_0) + (L_0 \setminus L_2)$$
has any words (even \_).
If $L_1 = L_2$, then $L_3 = \_.$
If $L_1 \neq L_2$, then $L_3 \neq \_.$
Example: Suppose \( L_1 = \{a, \text{aa}\} \) and \( L_2 = \{a, \text{aa, aaa}\} \). Then \( L_1 \setminus L_2 = \);, but \( L_0 \setminus L_2 = \{\text{aaa}\} \). Thus, \( L_1 \neq L_2 \).

So now we have reduced the problem of determining if \( L_1 = L_2 \) to determining if \( L_3 = ; \).

11.2.2 Is \( L = ; \)?

So we need a method for determining if a regular language is empty.

Since the language is regular, it has a regular expression and a FA.

Given a regular expression, check if there is any part that is not concatenated with \( ; \).

Specifically, use the following algorithm to determine if \( L = ; \) given a regular expression \( r \) for \( L \):

Method 1 (for deciding if a language \( L = ; \) given regular expression \( r \) for \( L \)):

Write \( r \) as \( r = r_1 + r_2 + \cdots + r_n \), where for each \( i = 1, 2, \ldots, n \), \( r_i = r_{i,1}r_{i,2}\cdots r_{i,j_i} \) for some \( j_i \neq 1 \); i.e., \( r \) is written as a “sum” of other regular expressions \( r_i, i = 1, 2, \ldots, n \), where each \( r_i \) is a concatenation of regular expressions.

It is always possible to write any regular expression \( r \) in this form.

If there exists some \( i = 1, 2, \ldots, n \) such that \( r_{i,j_i} \neq ; \) for all \( 1 \leq j \leq j_i \), then \( L \neq ; \). In other words, if one of the summands has none of its “factors” being \( ; \), then the language \( L \) is not empty.

If for each \( i = 1, 2, \ldots, n \), at least one of \( r_{i,1}, r_{i,2}, \ldots, r_{i,j_i} \) is \( ; \), then \( L = ; \). In other words, if each of the summands has at least one “factor” being \( ; \), then the language \( L \) is empty.

Example: The regular expression \( ;(b + a)_- + b \) has the last \( b \) not concatenated with \( ; \) so the language is not empty.

Example: The regular expression \( ;(b + a)_- + b \) has all parts concatenated with \( ; \) so the language is empty.

Remarks: The algorithm in the book for determining if \( L = ; \) given a regular expression for \( L \) is incorrect.

Method 2 (for deciding if a language \( L = ; \)): Given an FA, we check if there are any paths from \( - \) to some \( + \) state by using the “blue paint algorithm”:

1. Paint the start state blue.
2. From every blue state, follow each edge that leads out of it and paint the connecting state blue, then delete this edge from the machine.
3. Repeat Step 2 until no new state is painted blue, then stop.
4. When the procedure has stopped, if any of the final states are painted blue, then the machine accepts some words, and if not, the machine accepts no words.

Remarks on Method 2:

The above algorithm will iterate Step 2 at most \( N \) times, where \( N \) is the number of states in the machine.

Thus, it is a decision procedure.

Example:
Theorem 17 Let F be an FA with N states. Then if F accepts any strings at all, it accepts some string with N − 1 or fewer letters.

Proof.
Consider any string w that is accepted by F.
Let s = w and DONE = NO.
Do while (DONE == NO)

Trace path of s through F.
If no circuits in path, then set DONE = YES.
If there are circuits in the path, then
Eliminate first circuit in the path.
Let s be the string resulting from the new path.
Resulting path:
Starts in initial state.
Ends in a final state.
Has no circuits, so visits at most N states.
This corresponds to a string of at most N − 1 letters.
String is accepted by FA.

Method 3 (for deciding if a language L = ;): Test all words with N −1 or fewer letters by running them on the FA.

Theorem 18 There are effective procedures to decide whether:
1. A given FA accepts any words.

2. Two FA’s are equivalent; i.e., the two FA’s accept the same language.
3. Two regular expressions are equivalent; i.e., the two regular expressions generate the same language.

Remarks:
- We can establish part 3 of Theorem 18 by first converting the regular expressions into FA’s.
• We previously saw an effective procedure for doing this in the proof of Kleene's Theorem.
• Then we just developed an effective procedure to decide whether two FA's are equivalent.

Is L infinite?
Determining if a language L is infinite
If we have a regular expression for L, then all we need to do is check if the is applied to some part of the regular expression that is not _ nor

Theorem 19 Let F be an FA with N states. Then
If F accepts an input string w such that N _ length(w) < 2N
then F accepts an infinite language.
If F accepts infinitely many words, then F accepts some word w such
That N _ length(w) < 2N

Proof.
Assume that F accepts an input string w such that N _ length(w) < 2N
Since length(w) _ N, the second version of the pumping lemma (Theorem 14) implies
that there exist substrings x, y, and z such that y 6= _ and xynz, n = 0, 1, 2, . . . , are all accepted by F.
Thus, the FA accepts infinitely many words.

Assume that F accepts infinitely many words.
This implies that there exists some word u accepted by F that has a circuit (possibly more than one). Why?
Each circuit can consist of at most N states since F has only N states.
Iteratively eliminate the first circuit in the path until only one circuit left (as in the proof of Theorem 17).
Let v correspond to the word from this one-circuit path, and note that v is accepted by F.
We can write v as the concatenation of three strings x, y, and z, i.e., v = xyz, such that
x consists of the letters read before the circuit.
y consists of the letters read along the circuit.
z consists of the letters read after the circuit
We can show that 0 < length(y) _ N as follows:
Since we have eliminated all but the first circuit, the circuit starts and ends in the same state and all of the other states are unique. Thus, the circuit can visit at most N + 1 states (with at most one state repeated). This corresponds to reading at most N letters.

Also, since a circuit corresponds to at least one transition and each transition in an FA uses up exactly one letter, we see that length(y) > 0.
We can show that length(x) + length(z) < N as follows:
Since we constructed the string v by eliminating all but the first circuit, the paths followed by processing x and z have no circuits. Thus, all of the states visited along the paths followed by processing x and z are unique. Hence, the paths followed by processing x and z visit at most N states. This means that length(x) + length(z) _ N − 1 < N.
Thus, length(v) = length(x) + length(y) + length(z) _ N − 1 + N < 2N.
If v has at least N letters, then we are done.
If $v$ has less than $N$ letters, then we can pump up the cycle some number of times to obtain a word that has the desired characteristics since $0 < \text{length}(y) < N$.

Example:

- Consider the word $w = \text{abaaaababbab}$
- $\text{length}(w) = 13 > 2N = 12$.
- $w$ is accepted by the FA.
- Processing $w$ on FA takes the path
  $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6$
  which corresponds to the word $\text{abaaab}$, which has length 6.

Thus, Theorem 19 implies that the FA accepts an infinite language.

Consider the word $w = \text{bbaabb}$
- $\text{length}(w) = 6 = N$
- $w$ is accepted by the FA.

Processing $w$ on FA takes the path
$1 \rightarrow 3 \rightarrow \{z\} \rightarrow 2 \rightarrow 4 \rightarrow 6$

• Consider the word $w = \text{abaaaababbab}$
• $\text{length}(w) = 13 > 2N = 12$.
• $w$ is accepted by the FA.
• Processing $w$ on FA takes the path
  $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6$
  which corresponds to the word $\text{abaaab}$, which has length 6.

Thus, Theorem 19 implies that the FA accepts an infinite language.

Consider the word $w = \text{bbaabb}$
- $\text{length}(w) = 6 = N$
- $w$ is accepted by the FA.

Processing $w$ on FA takes the path
$1 \rightarrow 3 \rightarrow \{z\} \rightarrow 2 \rightarrow 4 \rightarrow 6$
Bypassing all but the first circuit yields the path
\[ 1 \rightarrow 3 \mid \{z\} \text{ circuit 1} \]
\[ 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6 \]
which corresponds to the word bbaab, which has length 5.

However, we can go around the circuit one more time, yielding the path
\[ 1 \rightarrow 3 \mid \{z\} \text{ circuit 1} \]
\[ 1 \rightarrow 3 \mid \{z\} \text{ circuit 1} \]
\[ 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6 \]
which corresponds to the word bbbaab, which has length 6.

Theorem 20 There is an effective procedure to decide whether a given FA accepts a finite or an infinite language.
Proof.
Suppose that the FA has \( N \) states.
Suppose that the alphabet consists of \( m \) letters.
Then by Theorem 19, we only need to check all strings \( w \) with \( N \leq \text{length}(w) < 2N \) to determine if FA accepts an infinite language.
If any of these are accepted, then the FA accepts an infinite language.
Otherwise, it accepts a finite language.
The number of strings \( w \) satisfying \( N \leq \text{length}(w) < 2N \) is \( mN + mN+1 + mN+2 + \cdots + m2N−1 \) which is finite.
• Thus, checking all of these strings is an effective procedure.

Definition: A Regular grammar, \( RG \), is composed of
1. \( \Sigma \): A finite non-empty set of symbols called terminals
2. \( N \): A finite set of symbols, called Non-terminals. \( \Sigma \) and \( N \) are disjoint
3. \( S \): A special start symbol in \( N \).
4. \( P \): A set of productions in one of the following two forms (not both):
   4a. Right RG: \( N_i \rightarrow \Sigma^*N_j \mid \Sigma^* \)
   4b. Left RG: \( N_i \rightarrow N_j \Sigma^* \mid \Sigma^* \)

Where \( N_i \) is a single non-terminal.
A regular grammar can be either Right Regular or Left Regular not both.

Example: A regular grammar for \( ab^* = \{ a, \ ab, \ abb, \ abbb, \ldots \} \)
A Right Regular Grammar for \( ab^* \)
\[ S \rightarrow aB \]
\[ B \rightarrow bB \mid b \]
A Left Regular Grammar for \( ab^* \)
\[ S \rightarrow Sb \mid a \]

We always represent non-terminals in upper case and terminals in lower case.
Other References: