Chapter 7

Kleene’s Theorem

7.1 Kleene’s Theorem

The following theorem is the most important and fundamental result in the theory of FA’s:

**Theorem 6** Any language that can be defined by either

- regular expression, or
- finite automata, or
- transition graph

can be defined by all three methods.

**Proof.** The proof has three parts:

**Part 1:** (FA $\Rightarrow$ TG) Every language that can be defined by an FA can also be defined by a transition graph.

**Part 2:** (TG $\Rightarrow$ RegExp) Every language that can be defined by a transition graph can also be defined by a regular expression.

**Part 3:** (RegExp $\Rightarrow$ FA) Every language that can be defined by a regular expression can also be defined by an FA.
7.2 Proof of Part 1: FA ⇒ TG

- We previously saw that every FA is also a transition graph.
- Hence, any language that has been defined by a FA can also be defined by a transition graph.

7.3 Proof of Part 2: TG ⇒ RegExp

- We will give a constructive algorithm for proving part 2.
- Thus, we will describe an algorithm to take any transition graph $T$ and form a regular expression corresponding to it.
- The algorithm will work for any transition graph $T$.
- The algorithm will finish in finite time.

An overview of the algorithm is as follows:

- Start with any transition graph $T$.
- First, transform it into an equivalent transition graph having only one start state and one final state.
- In each following step, eliminate either some states or some arcs by transforming the TG into another equivalent one.
- We do this by replacing the strings labelling arcs with regular expressions.
- We can traverse an arc labelled with a regular expression using any string that can be generated by the regular expression.
- End up with a TG having only two states, start and final, and one arc going from start to final.
- The final TG will have a regular expression on its one arc
- Note that in each step we eliminate some states or arcs.
- Since the original TG has a finite number of states and arcs, the algorithm will terminate in a finite number of iterations.
CHAPTER 7. KLEENE’S THEOREM

Algorithm:

1. If $T$ has more than one start state, add a new state and add arcs labeled \( \Lambda \) going to each of the original start states.

2. If $T$ has more than one final state, add a new state and add arcs labeled \( \Lambda \) going from each of the original final states to the new state. Need to make sure the final state is different than the start state.
3. Now we give an iterative procedure for eliminating states and arcs

(a) If $T$ has some state with $n > 1$ loops circling back to itself, where the loops are labeled with regular expressions $r_1, r_2, \ldots, r_n$, then replace the $n$ loops with a single loop labeled with the regular expression $r_1 + r_2 + \cdots + r_n$.

(b) If two states are connected by $n > 1$ direct arcs in the same direction, where the arcs are labelled with the regular expressions $r_1, r_2, \ldots, r_n$, then replace the $n$ arcs with a single arc labeled with the regular expression $r_1 + r_2 + \cdots + r_n$.
(c) Bypass operation:

i. If there are three states $x, y, z$ such that
   
   - there is an arc from $x$ to $y$ labelled with the regular expression $r_1$ and
   - an arc from $y$ to $z$ labelled with the regular expression $r_2$,

   then replace the two arcs and the state $y$ with a single arc from $x$ to $z$ labelled with the regular expression $r_1 r_2$. 

\[ x \xrightarrow{r_1} y \xrightarrow{r_2} z \Rightarrow x \xrightarrow{r_1 r_2} z \]

\[ x \xrightarrow{r_1} y \xrightarrow{r_2} z \xrightarrow{r_3} \Rightarrow x \xrightarrow{r_1 r_3 r^* r_2} z \]
ii. If there are
• $n+2$ states $x, y, z_1, z_2, \ldots, z_n$ such that there is an arc from $x$ to $y$ labelled with the regular expression $r_0$, and
• an arc from $y$ to $z_i$, $i = 1, 2, \ldots, n$, labelled with the regular expression $r_i$, and
• an arc from $y$ back to itself labelled with regular expression $r_{n+1}$,
then replace the $n + 1$ original arcs and the state $y$ with $n$ arcs from $x$ to $z_i$, $i = 1, 2, \ldots, n$, each labelled with the regular expression $r_0r_{n+1}r_i$.

iii. If any other arcs led directly to $y$, divert them directly to the $z_i$'s.
iv. Need to make sure that all paths possible in the original TG are still possible after the bypass operation.

- Example

\[
\begin{align*}
\text{w} & \xrightarrow{r_1} x & x & \xrightarrow{r_4} z \\
\text{w} & \xrightarrow{r_2} y & y & \xrightarrow{r_3} z \\
\text{w} & \xrightarrow{r_1 (r_2 r_3)^* r_4} z \\
\text{w} & \xrightarrow{r_1 (r_2 r_3)^* r_4 + r_1 r_2 (r_3 r_2)^* r_5} z
\end{align*}
\]
Example:

- Suppose we want to get rid of state $y$.
- Need to account for all paths that go through state $y$.
- There are arcs coming from $x$, $w$, and $z$ going into $y$.
- There are arcs from $y$ to $x$ and $z$.
- Thus, we need to account for each possible path from a state having an arc into $y$ (i.e., $x$, $w$, $z$) to each state having an arc from $y$ (i.e., $x$, $z$).
- Thus, we need to account for the paths from
  * $x$ to $y$ to $x$, which has regular expression $r_1 r_2^* r_5$
  * $x$ to $y$ to $z$, which has regular expression $r_1 r_2^* r_3$
  * $w$ to $y$ to $x$, which has regular expression $r_7 r_2^* r_5$
  * $w$ to $y$ to $z$, which has regular expression $r_7 r_2^* r_3$
  * $z$ to $y$ to $x$, which has regular expression $r_6 r_2^* r_5$
  * $z$ to $y$ to $z$, which has regular expression $r_6 r_2^* r_3$
- Thus, after eliminating state $y$, we get the following:

v. Never delete the unique start or final state.
Example:

$\begin{array}{c}
1 \xrightarrow{a} 2 \\
\quad \xleftarrow{ab} 3 \\
\quad \xrightarrow{ba} 4 \\
\quad \xleftarrow{abb} \Lambda \\
\quad \\
\quad \xrightarrow{\Lambda} 5 + \\
\quad \xleftarrow{a, b} \Lambda \\
\end{array}$

$\begin{array}{c}
1 - \xrightarrow{a} 1 - \\
\quad \xleftarrow{ab} abba \\
\quad \xrightarrow{abb} 5 + \\
\quad \xleftarrow{bb} a+b \\
\quad \xrightarrow{a+b} 5 + \\
\end{array}$

$\begin{array}{c}
1 - \xrightarrow{a} 1 - \\
\quad \xleftarrow{ab} a(abba+abb+bb)(a+b)* \\
\end{array}$
Example:
\[(\Lambda + b)((ab)bb^*(\Lambda + a(a+b)^*) + a(ba)^*a(a+b)^*)
\]

\[
\Rightarrow \quad (\Lambda + b)((ab)bb^*(\Lambda + a(a+b)^*) + a(ba)^*a(a+b)^*) + a(ba)^*a(a+b)^* + ab(ab)^*bb^*(\Lambda + a(a+b)^*)
\]
7.4 Proof of Part 3: RegExp $\Rightarrow$ FA

To show: every language that can be defined by a regular expression can also be defined by a FA.

We will do this by using a recursive definition and a constructive algorithm.

Recall

- every regular expression can be built up from the letters of the alphabet and $\Lambda$ and $\emptyset$.
- Also, given some existing regular expressions, we can build new regular expressions by applying the following operations:
  1. union (+)
  2. concatenation
  3. closure (Kleene star)
- We will not include $r^+$ in our discussion here, but this will not be a problem since $r^+ = rr^*$. 
Recall that we had the following recursive definition for regular expressions:

**Rule 1:** If $x \in \Sigma$, then $x$ is a regular expression. $\Lambda$ is a regular expression. $\emptyset$ is a regular expression.

**Rule 2:** If $r_1$ and $r_2$ are regular expressions, then $r_1 + r_2$ is a regular expression.

**Rule 3:** If $r_1$ and $r_2$ are regular expressions, then $r_1r_2$ is a regular expression.

**Rule 4:** If $r_1$ is a regular expression, then $r_1^*$ is a regular expression.

Based on the above recursive definition for regular expressions, we have the following recursive definition for FA’s associated with regular expressions:

**Rule 1:**

- There is an FA that accepts the language $L$ defined by the regular expression $x$; i.e., $L = \{x\}$, where $x \in \Sigma$, so language $L$ consists of only a single word and that word is the single letter $x$.
- There is an FA that accepts the language defined by regular expression $\Lambda$; i.e., the language $\{\Lambda\}$.
- There is an FA defined by the regular expression $\emptyset$; i.e., the language with no words, which is $\emptyset$.

**Rule 2:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$ and there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_2$, then there is an FA called $FA_3$ that accepts the language defined by the regular expression $r_1 + r_2$.

**Rule 3:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$ and there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_2$, then there is an FA called $FA_3$ that accepts the language defined by the regular expression $r_1r_2$, which is the concatenation.

**Rule 4:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$, then there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_1^*$. 
Let’s now show that each of the rules hold by construction:

**Rule 1:** There is an FA that accepts the language $L$ defined by the regular expression $x$; i.e., $L = \{x\}$, where $x \in \Sigma$. There is an FA that accepts language defined by the regular expression $\Lambda$. There is an FA that accepts the language defined by the regular expression $\emptyset$.

- If $x \in \Sigma$, then the following FA accepts the language $\{x\}$:

  ![Diagram](image)

- An FA that accepts the language $\{\Lambda\}$ is

  ![Diagram](image)

- An FA that accepts the language $\emptyset$ is

  ![Diagram](image)
Rule 2: If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$ and there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_2$, then there is an FA called $FA_3$ that accepts the language defined by the regular expression $r_1 + r_2$.

- Suppose regular expressions $r_1$ and $r_2$ are defined with respect to a common alphabet $\Sigma$.
- Let $L_1$ be the language generated by regular expression $r_1$.
- $L_1$ has finite automaton $FA_1$.
- Let $L_2$ be the language generated by regular expression $r_2$.
- $L_2$ has finite automaton $FA_2$.
- Regular expression $r_1 + r_2$ generates the language $L_1 + L_2$.
- Recall $L_1 + L_2 = \{w \in \Sigma^* : w \in L_1 \text{ or } w \in L_2\}$.
- Thus, $w \in L_1 + L_2$ if and only if $w$ is accepted by either $FA_1$ or $FA_2$ (or both).
- We need $FA_3$ to accept a string if the string is accepted by $FA_1$ or $FA_2$ or both.
- We do this by constructing a new machine $FA_3$ that simultaneously keeps track of where the input would be if it were running on $FA_1$ and where the input would be if it were running on $FA_2$.
- Suppose $FA_1$ has states $x_1, x_2, \ldots, x_m$, and $FA_2$ has states $y_1, y_2, \ldots, y_n$.
- Assume that $x_1$ is the start state of $FA_1$ and that $y_1$ is the start state of $FA_2$.
- We will create $FA_3$ with states of the form $(x_i, y_j)$.
- The number of states in $FA_3$ is at most $mn$, where $m$ is the number of states in $FA_1$ and $n$ is the number of states in $FA_2$.
- Each state in $FA_3$ corresponds to a state in $FA_1$ and a state in $FA_2$.
- $FA_3$ accepts string $w$ if and only if either $FA_1$ or $FA_2$ accepts $w$.
- So final states of $FA_3$ are those states $(x, y)$ such that $x$ is a final state of $FA_1$ or $y$ is a final state of $FA_2$. 
We use the following algorithm to construct $FA_3$ from $FA_1$ and $FA_2$.

- Suppose that $\Sigma$ is the alphabet for both $FA_1$ and $FA_2$.
- Given $FA_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$ with
  - Set of states $K_1 = \{x_1, x_2, \ldots, x_m\}$
  - $s_1 = x_1$ is the initial state
  - $F_1 \subseteq K_1$ is the set of final states of $FA_1$.
  - $\pi_1 : K_1 \times \Sigma \rightarrow K_1$ is the transition function for $FA_1$.
- Given $FA_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$ with
  - Set of states $K_2 = \{y_1, y_2, \ldots, y_n\}$
  - $s_2 = y_1$ is the initial state
  - $F_2 \subseteq K_2$ is the set of final states of $FA_2$.
  - $\pi_2 : K_2 \times \Sigma \rightarrow K_2$ is the transition function for $FA_2$.
- We then define $FA_3 = (K_3, \Sigma, \pi_3, s_3, F_3)$ with
  - Set of states $K_3 = K_1 \times K_2 = \{(x, y) : x \in K_1, y \in K_2\}$
  - The alphabet of $FA_3$ is $\Sigma$.
  - $FA_3$ has transition function $\pi_3 : K_3 \times \Sigma \rightarrow K_3$ with
    \[ \pi_3((x, y), \ell) = (\pi_1(x, \ell), \pi_2(y, \ell)). \]
  - The initial state $s_3 = (s_1, s_2)$.
  - The set of final states
    \[ F_3 = \{(x, y) \in K_1 \times K_2 : x \in F_1 \text{ or } y \in F_2\}. \]
- Since $K_3 = K_1 \times K_2$, the number of states in the new machine $FA_3$ is $|K_3| = |K_1| \cdot |K_2|$.
  - But we can leave out a state $(x, y) \in K_1 \times K_2$ from $K_3$ if $(x, y)$ is not reachable from $FA_3$’s initial state $(s_1, s_2)$.
  - This would result in fewer states in $K_3$, but still we have $|K_1| \cdot |K_2|$ as an upper bound for $|K_3|$; i.e., $|K_3| \leq |K_1| \cdot |K_2|$.
Example: $L_1 = \{ \text{words with } b \text{ as second letter} \}$
with regular expression $r_1 = (a + b)b(a + b)^*$
$L_2 = \{ \text{words with odd number of } a's \}$
with regular expression $r_2 = b^*a(b + ab^*a)^*$

FA1 for $L_1$: 

FA2 for $L_2$: 

FA3 for $L_1 + L_2$: 

Rule 3: If there is an FA called \( FA_1 \) that accepts the language defined by the regular expression \( r_1 \) and there is an FA called \( FA_2 \) that accepts the language defined by the regular expression \( r_2 \), then there is an FA called \( FA_3 \) that accepts the language defined by the regular expression \( r_1r_2 \).

For this part,

- we need \( FA_3 \) to accept a string if the string can be factored into two substrings, where the first factor is accepted by \( FA_1 \) and the second factor is accepted by \( FA_2 \).
- One problem is we don’t know when we reach the end of the first factor and the beginning of the second factor.

Example: \( L_1 = \{ \text{words that end with } aa \} \) with regular expression \( r_1 = (a+b)^*aa \)
\( L_2 = \{ \text{words with odd length} \} \) with regular expression \( r_2 = (a+b)((a+b)(a+b))^* \)

- Consider the string \( baaab \).
- If we factor it as \( (baa)(ab) \), then \( baa \in L_1 \) but \( ab \notin L_2 \).
- However, another factorization, \( (baaa)(b) \), shows that \( baaab \in L_1L_2 \) since \( baaa \in L_1 \) and \( b \in L_2 \).
• Basically idea of building $FA_3$ for $L_1L_2$ from $FA_1$ for $L_1$ and $FA_2$ for $L_2$:
  ■ Recall $L_1L_2 = \{w = w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$.
  ■ So a string $w$ is in $L_1L_2$ if and only if we can factor $w = w_1w_2$ such that $w_1$ is accepted by $FA_1$ and $w_2$ is accepted by $FA_2$.
  ■ $FA_3$ initially acts like $FA_1$.
  ■ When $FA_3$ hits a $\oplus$ state of $FA_1$,
    * Start a version of $FA_2$.
    * Keep processing on $FA_1$ and any previous versions of $FA_2$.
  ■ We need to keep processing on $FA_1$ because we don’t know where the first factor $w_1$ ends and the second factor $w_2$ begins
  ■ Final states of $FA_3$ are those states that have at least one final state from $FA_2$.

• More formally, we build machine $FA_3$ in following way:
  ■ Suppose that $FA_1$ and $FA_2$ have the same alphabet $\Sigma$.
  ■ Let $L_1$ be language generated by regular expression $r_1$ and having $FA_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$.
  ■ Let $L_2$ be language generated by regular expression $r_2$ and having $FA_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$.

  ■ Definition: For any set $S$, define $2^S$ to be the set of all possible subsets of $S$.

  Example: If $S = \{a, bb, ab\}$, then
  
  $$2^S = \{\emptyset, \{a\}, \{bb\}, \{ab\}, \{a, bb\}, \{a, ab\}, \{bb, ab\}, \{a, bb, ab\}\}.$$ 

  Fact: If $|S| < \infty$, then $|2^S| = 2^{|S|}$; i.e., there are $2^{|S|}$ different subsets of $S$.

  ■ Machine $FA_3 = (K_3, \Sigma, \pi_3, s_3, F_3)$ for $L_1L_2$ is as follows:
    * States
      $$K_3 = \{\{x\} + Y : x \in K_1, Y \in 2^{K_2}\};$$
      i.e., each state of $FA_3$ is a set of states, where exactly one of the states is from $FA_1$ and the rest (possibly none) are from $FA_2$.
    * Initial state $s_3 = \{s_1\}$; i.e., the initial state of $FA_3$ is the set consisting of only the initial state of $FA_1$. 

* Transition function $\pi_3 : K_3 \times \Sigma \to K_3$ is defined as

$$
\pi_3(\{x, y_1, \ldots, y_n\}, \ell) = \begin{cases} 
\{\pi_1(x, \ell), \pi_2(y_1, \ell), \ldots, \pi_n(y_n, \ell)\} & \text{if } \pi_1(x, \ell) \notin F_1, \\
\{\pi_1(x, \ell), \pi_2(y_1, \ell), \ldots, \pi_n(y_n, \ell), s_2\} & \text{if } \pi_1(x, \ell) \in F_1,
\end{cases}
$$

where $\{x, y_1, \ldots, y_n\} \in K_3$, $n \geq 0$, $x \in K_1$, $y_i \in K_2$ for $i = 1, \ldots, n$, and $\ell \in \Sigma$.

* Final states

$$F_3 = \{\{x, y_1, \ldots, y_n\} : n \geq 1, y_i \in F_2 \text{ for some } i = 1, \ldots, n\}.$$  

The number of states in $FA_3$ is

$$|K_3| = |K_1| \cdot |2^{K_2}| = |K_1| \cdot 2^{|K_2|}.$$  

* Actually, we can leave out from $K_3$ any states $\{x, y_1, \ldots, y_n\}$ that are not reachable from the initial state $s_3$.

* In this case, $|K_1| \cdot 2^{|K_2|}$ still provides an upper bound for $|K_3|$; i.e., $|K_3| \leq |K_1| \cdot 2^{|K_2|}$.
Example: \( L_1 = \{ \text{words that end with } aa \} \)
with regular expression \( r_1 = (a + b)^* aa \)
\( L_2 = \{ \text{words with odd length} \} \)
with regular expression \( r_2 = (a + b)((a + b)(a + b))^* \)
**Rule 4:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$, then there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_1^*$.

Basic idea of how to build machine $FA_2$:

- Each state of $FA_2$ corresponds to one or more states of $FA_1$.
- $FA_2$ initially acts like $FA_1$.
- when $FA_2$ hits a $\oplus$ state of $FA_1$, then $FA_2$ simultaneously keeps track of how the rest of the string would be processed on $FA_1$ from where it left off and how the rest of the string would be processed on $FA_1$ starting in the start state.
- Whenever $FA_2$ hits a $\oplus$ state of $FA_1$, we have to start a new process starting in the start state of $FA_1$ (if no version of $FA_1$ is currently in its start state.)
- The final states of $FA_2$ are those states which have a correspondence to some final state of $FA_1$.
- We need to be careful about making sure that $FA_2$ accepts $\Lambda$.
- To have $FA_2$ accept $\Lambda$, we make the start state of $FA_2$ also a final state.
- But we need to be careful when there are arcs going into the start state of $FA_1$. 

Formally, we build the machine $FA_2$ for $L_1^*$ as follows:

- Let $L_1$ be language generated by regular expression $r_1$ and having finite automaton $FA_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$.
- For now, assume that $FA_1$ does not have any arcs entering the initial state $s_1$.
- Know that language $L_1^*$ is generated by regular expression $r_1^*$.
- Define $FA_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$ for $L_1^*$ with
  - States $K_2 = 2^{K_1}$.
  - Initial state $s_2 = \{s_1\}$.
  - Transition function $\pi_2 : K_2 \times \Sigma \to K_2$ with
    
    \[
    \pi_2(\{x_1, \ldots, x_n\}, \ell) = \begin{cases}
    \{\pi_1(x_1, \ell), \ldots, \pi_1(x_n, \ell)\} & \text{if } \pi_1(x_k, \ell) \notin F_1 \text{ for all } k = 1, \ldots, n, \\
    \{\pi_1(x_1, \ell), \ldots, \pi_1(x_n, \ell), s_1\} & \text{if } \pi_1(x_k, \ell) \in F_1 \text{ for some } k = 1, \ldots, n,
    \end{cases}
    \]

    where $\{x_1, \ldots, x_n\} \in K_2$, $n \geq 1$, $x_i \in K_1$ for all $i = 1, \ldots, n$, and $\ell \in \Sigma$.
  - Final states
    
    $F_2 = \{s_1\} + \{\{x_1, \ldots, x_n\} : n \geq 1, x_i \in F_1 \text{ for some } i = 1, \ldots, n\}$.

- The number of states in $FA_2$ is
  
  $|K_2| = 2^{2^{K_1}} = 2^{2^{|K_1|}}$.

- Actually, we can leave out from $K_2$ any state $\{x_1, \ldots, x_n\}$ that is not reachable from the initial state $s_2$.
- In this case, $2^{2^{K_1}}$ still provides an upper bound for $|K_2|$; i.e., $|K_3| \leq 2^{2^{K_1}}$. 


Example: Consider language $L$ having regular expression

$$r = (a + bb*ab*a)((b + ab*a)b*a)^*$$

FA for $L$:

FA for $L^*$:
**Example:** Consider language $L$ having regular expression

$$(a + b)^* b$$

Need to be careful since we can return to the start state.

**FA for $L$:**

If we blindly applied previous method for constructing $FA$ for $L^*$, we get the following:

**Problem:**

- Note that start state is final state.
- But this $FA$ accepts $a \notin L^*$, and so this $FA$ is incorrect.
- Problem occurs because we can return to start state in original $FA$, and since we make the start state a final state in new $FA$. 
Solution:

- Given original FA $FA_1$ having arcs going into the initial state, create an equivalent FA $\tilde{FA}_1$ having no arcs going into the initial state by splitting the original start state $x_1$ of $FA_1$ into two states $x_{1.1}$ and $x_{1.2}$
  - $x_{1.1}$ is the new start state of $\tilde{FA}_1$ and is never visited again after the first letter of the input string is read.
  - $x_{1.2}$ in $\tilde{FA}_1$ corresponds to $x_1$ after the first letter of the input string is read.
- Then run algorithm to create FA for $L^*$ from the new FA $\tilde{FA}_1$.

new FA for $L$:

![Diagram of FA for L]

FA for $L^*$:

![Diagram of FA for L^*]
7.5 Nondeterministic Finite Automata

Definition: A nondeterministic finite automaton (NFA) is given by $M = (K, \Sigma, \Pi, s, F)$, where

1. $K$ is a finite set of states.
   - $s \in K$ is the initial state, which is denoted pictorially by ⊖, and there is exactly one initial state.
   - $F \subset K$ is a set of final states (possibly empty), where each final state is denoted pictorially by ⊕.

2. An alphabet $\Sigma$ of possible input letters.

3. $\Pi \subset K \times \Sigma \times K$ is a finite set of transitions, where each transition (arc) from one state to another state is labeled with a letter $\ell \in \Sigma$. (We do not allow for $\Lambda$ to be the label of an arc since $\Lambda$ is a string and not a letter of $\Sigma$.) We allow for the possibility of more than one edge with the same label from any state and there may be a state (or states) for which certain input letters have no edge leaving that state.
Example:

Note that

- definition of NFA is different from that of FA since
  - a FA must have from each state an arc labeled with each letter of alphabet, while NFA does not.
  - a FA is deterministic, while a NFA may be nondeterministic.
  - An NFA can have repeated labels from any single state.

- NFA allows for human choice to become a factor in selecting a way to process an input string.

- The definition of NFA is different from that of TG since
  - a TG can have arcs labeled with substrings of letters while a NFA has arcs labeled with only letters.
  - a TG can have arcs labeled with $\Lambda$ while a NFA cannot.
  - a TG can have more than one start state while a NFA can only have one.
• Can transform any NFA with repeated labels from any single state to an equivalent TG with no repeated labels from any single state.

\[
\begin{array}{c}
\text{1} \\
\text{b} \\
\text{a} \\
\text{b} \\
\text{b} \\
\text{2} \\
\text{3} \\
\text{4} \\
\text{5} \\
\text{b} \\
\text{a} \\
\text{b} \\
\text{b} \\
\text{\ldots} \\
\Rightarrow \\
\text{1} \\
\text{b} \\
\text{^} \\
\text{b} \\
\text{b} \\
\text{2} \\
\text{3} \\
\text{4} \\
\text{b} \\
\text{\ldots} \\
\text{5} \\
\text{b} \\
\text{\ldots}
\end{array}
\]

7.6 Properties of NFA

**Theorem 7** \( FA = NFA \); i.e., any language definable by a NFA is also definable by a deterministic FA and vice versa.

**Proof.** Note that

- Every FA is an NFA since we can consider an FA to be an NFA without the extra possible features.
- Every NFA is a TG.
- Kleene’s theorem states that every TG has an equivalent FA.

NFA useful because

- applications in artificial intelligence (AI).
• given two FA’s for two languages with regular expressions $r_1$ and $r_2$, it is easy to construct an NFA to accept language corresponding to regular expression $r_1 + r_2$. 
Example:

FA1:

FA2:

NFA$_{1+2}$

• This works when neither of the original FA’s has any arcs going into their original initial states.

• If one or both of the original FA’s has an arc going into its original initial state, the newly constructed FA for the language corresponding
to regular expression $r_1 + r_2$ may be incorrect. This is because the new FA may process part of the word on one of the original FA’s and then process the rest of the word on the other FA, and then incorrectly accept the word.