Griffith University

3515ICT Theory of Computation

Kleene’s Theorem

(Based loosely on slides by Harald Søndergaard of The University of Melbourne)
Kleene’s Theorem

**Theorem** For every language $L$ (over a finite alphabet $\Sigma$), the following statements are equivalent:

1. $L$ is defined by some regular expression $E$.
2. $L$ is accepted by some nondeterministic finite automaton $N$.
3. $L$ is accepted by some deterministic finite automaton $D$. 
Lemma For every NFA $N$, there exists a DFA $D$ s.t. $L(D) = L(N)$.

The proof is based on the so-called subset construction (Theorem 1.39, pp.55–58).

Given an NFA $N$, we construct a DFA $D$, each of whose states is a set of $N$-states.

If $N$ has $k$ states then $D$ may have up to $2^k$ states (but it will often have far fewer than that).
Consider the NFA

We can systematically construct an equivalent DFA.

Its start state is \( \{1, 3\} \).

From this state an \( a \) will take us back to \( \{1, 3\} \).

From \( \{1, 3\} \), \( b \) can only take us to \( \{2\} \).

Continuing similarly gives the DFA.

Any state \( S \) which contains an accept state from the NFA will be an accept state for the DFA.
Let \( N = (Q, \Sigma, \delta, q_0, F) \).

Let \( E(S) \) be the "\( \epsilon \)-closure" of \( S \subseteq Q \), i.e., \( S \) together with all states reachable from \( S \) using only \( \epsilon \)-transitions:

\[
E(S) = \bigcup_{s \in S} \{ s' \in Q \mid s \xrightarrow{\epsilon} s' \}
\]

We construct \( D = (Q', \Sigma, \delta', q'_0, F') \) as follows.

- \( Q' = 2^Q \)
- \( q'_0 = E(\{q_0\}) \).
- \( \delta'(S, a) = \bigcup_{s \in S} E(\delta(s, a)) \).
- \( F' = \{ S \in Q' \mid S \cap F \neq \emptyset \} \).

**Note** This construction may include some unreachable states.
### Example 1.41 (cont.)

**NFA:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>ϵ</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>*1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DFA:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>*13</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>123</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>∅</td>
</tr>
<tr>
<td>*123</td>
<td>123</td>
<td>23</td>
</tr>
<tr>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>
**Lemma** For every regular expression $E$, there exists an NFA $N$ s.t. $L(N) = L(E)$.

The proof is by **structural induction** on the form of $E$ (Closure under regular operations, pp.58–63, and Lemma 1.55, pp. 67–69)

- **Case** $E = \epsilon$: $\longrightarrow \bigcirc$
- **Case** $E = \emptyset$: $\longrightarrow \bigcirc$
- **Case** $E = a$: $\longrightarrow \bigcirc \xrightarrow{a} \bigcirc$

**Cases** $E = E_1 \cup E_2$, $E = E_1E_2$, or $E = E_1^*$: See following slides.
Case $E = E_1 + E_2$ (union):

By induction, suppose there are NFAs that accept $L(E_1)$ and $L(E_2)$. Then the following NFA accepts $L(E_1 + E_2)$.

Here $F_E = F_1 \cup F_2$. 
Case $E = E_1 E_2$ (concatenation):

By induction, suppose there are NFAs that accept $L(E_1)$ and $L(E_2)$

Then the following NFA accepts $L(E_1 E_2)$:

Here $F_E = F_2$. 
Case $E = E_1^*$ (repetition):

By induction, suppose there is an NFA that accepts $L(E_1)$.

Then the following NFA accepts $L(E_1^*)$:

Here, $F_E = \{q_0\} \cup F_1$. 
Example

Let us construct an NFA for \((a \cup b)^*bc\)

Start from innermost expressions and work out:

- \(a\)
- \(b\)

So \(a \cup b\) yields:

- \(a\)
- \(b\)

Diagram:

4-10
Then \((a \cup b)^*\) yields:

Finally \((a \cup b)^*bc\) yields:

Of course there are simpler, equivalent automata.
Lemma For every DFA $D$, there exists a regular expression $E$ s.t. $L(E) = L(D)$.

The proof is based on the so-called state elimination construction (Lemma 1.60, pp.69–76).

There are two main steps:

1. Transform the DFA $D$ into an equivalent GNFA $G$.

2. Repeatedly eliminate states from $G$ until only two states remain.

A generalised NFA (GNFA) is an NFA in which transitions are labelled by REs. A GNFA follows a transition labelled by the RE $E$ if the input has a prefix in $L(E)$. 
Given a DFA $D$, transform it to a GNFA $G$ as follows:

1. If the initial state of $D$ has any incoming transitions, add a new initial state $q_0$ with an $\epsilon$-transition to the initial state of $D$.

2. Add a new accepting state $q_A$ with an $\epsilon$-transition from every accepting state of $D$ to $q_A$. 

**DFAs $\Rightarrow$ GNFAs**
State elimination

Consider a state $q$ to be eliminated:

\[
\begin{array}{c}
q_1 \xrightarrow{E_1} \bullet \xrightarrow{E_2} q \xrightarrow{E_3} q_2
\end{array}
\]

Replace every pair of transitions from $q_1$ through $q$ to $q_2$ by a single transition with label $E_1E_2^*E_3$, and eliminate $q$. (Note that $q_1 = q_2$ is possible. Note also that $\epsilon E = E \epsilon = E$.)

If this results in two transitions from $q_1$ to $q_2$ with labels $E'_1$ and $E'_2$, they should be merged into a single transition with label $E'_1 + E'_2$.

Repeatedly eliminate intermediate states until no intermediate states remain.

Return the RE $E$ labelling the single transition from the initial state to the accepting state.
Example

Binary strings representing integers evenly divisible by 3 (after transforming to GNFA):

Eliminating state $c$: 
Eliminating state $b$:

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

Eliminating state $a$:

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

Final regular expression:

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