Lecture 4
State Reduction, Regular Expressions and CFL

CSc 135

Computing Theory and Programming Languages

Many slides by Ron Mak

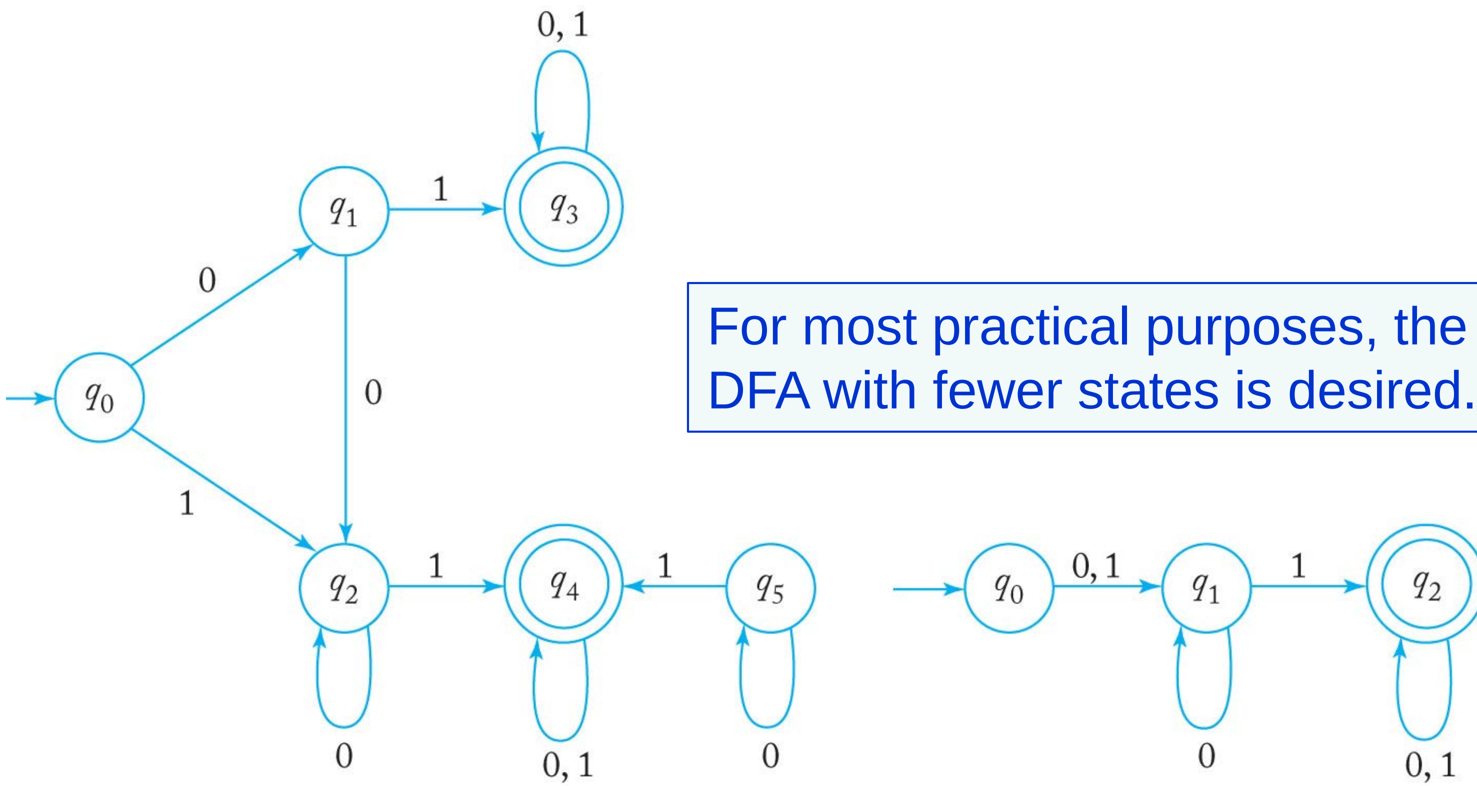
State Reduction

Equivalent DFAs

- A DFA defines a unique language.
- But a given language can have many DFAs that define it.
- Two DFAs can be equivalent and yet have a different number of states.
  - Equivalent DFAs define the same language.

Equivalent DFAs (cont.)

- These two DFAs are equivalent:



(a)

(b)

For most practical purposes, the DFA with fewer states is desired.

Indistinguishable States

- Consider two states p and q of a DFA and <u>all</u> strings w in Σ^*.

- If there <u>is</u> path from p to a final state implies there <u>is</u> a path from q to a final state, and

Not necessarily the same final state.

- If there is <u>no</u> path from p to a final state implies there is <u>no</u> path from q to a final state,

- Then states p and q are indistinguishable.

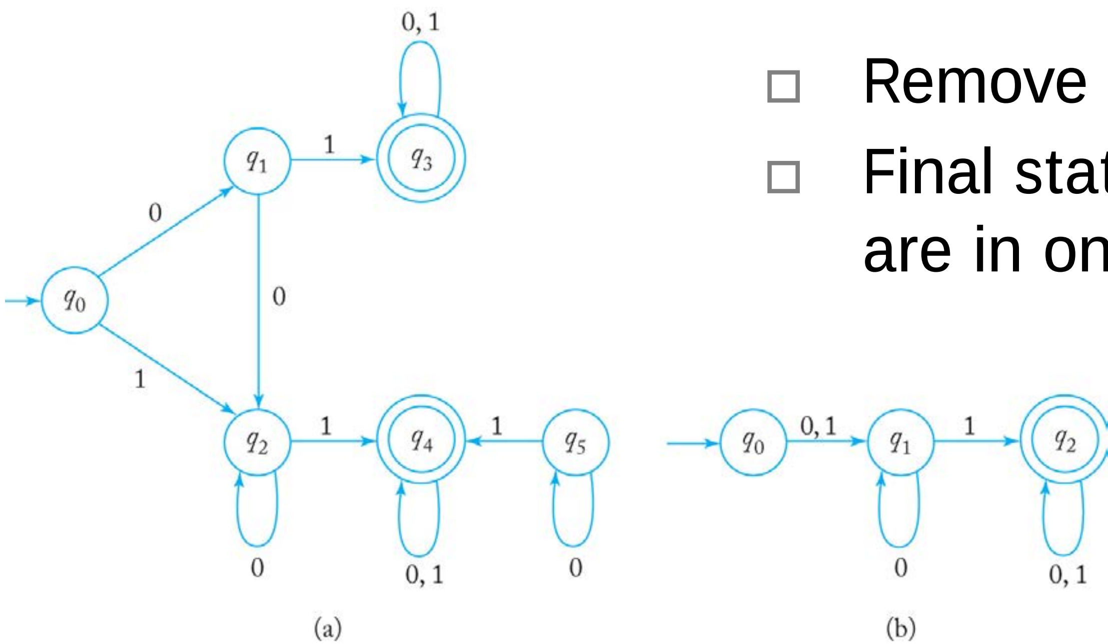
Distinguishable States

- However, if for <u>any one</u> string w there <u>is</u> a path from p to a final state but <u>no</u> path from q to a final state (or vice versa),
- Then the states p and q are distinguishable.

Reducing the Number of States

- Given a DFA, how can we simplify it by reducing the number of states?
  - Of course, we want the simplified DFA to be equivalent to the original one.
- One way:
  - Find and combine indistinguishable states.
- Plan:
  - First eliminate inaccessible states.
  - Then repeatedly partition the states into equivalence classes of indistinguishable states.

State Reduction Example #1



- Remove inaccessible state q5.
- Final states q_3 and q_4 are in one equivalence class:

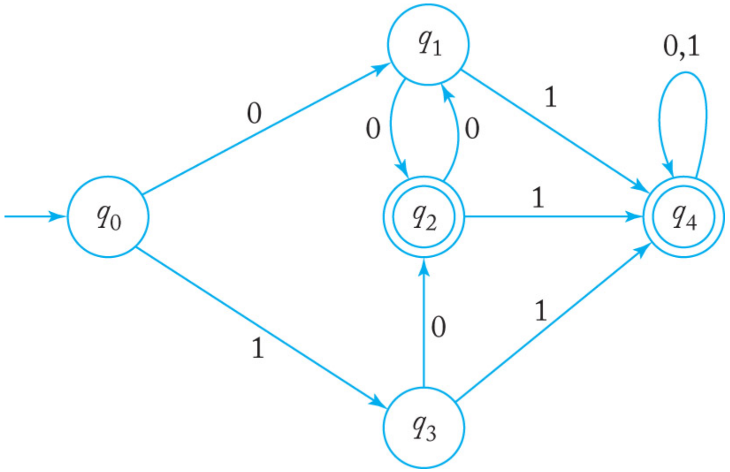
$$q_0\ q_1\ q_2 \mid q_3\ q_4$$

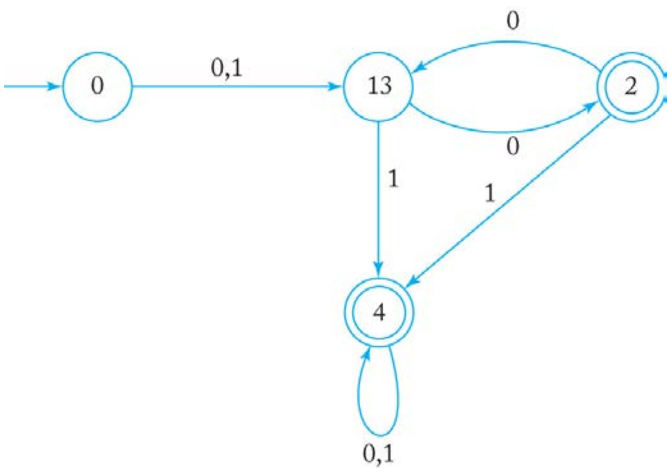
- From either q_1 or q_2 , input 1 and input 01 lead to a final state, so they're together in another equivalence class.

$$q_0 \mid q_1\ q_2 \mid q_3\ q_4$$

- We can't partition any further, so make new states out of each equivalence class.

State Reduction Example #2





- States q_2 and q_4 are final:
- From q_1 and q_3, strings 0 and 1 both lead to final states:
- $\delta(q_4, 0) = q_4$ but $\delta(q_2, 0) = q_1$:
- No further partitioning is possible.

0 1 3 | 2 4

0 | 1 3 | 2 4

0 | 1 3 | 2 | 4

JFLAP demo

Regular Expressions

Regular Languages and Automata

- A language L is called regular if and only if there exists a finite acceptor M such that $L = L(M)$.

- The finite acceptor can be a DFA or an NFA.

- Is there a more concise way to describe a regular language?

Regular Expressions

- A regular expression consists of strings of symbols from an alphabet Σ, parentheses, and the operators:
  - $+$ for union: $a + b$
  - $\bullet$ for concatenation: $a \bullet b$ which can also be written ab
  - $*$ for star-closure: a^*
- Example: $(a + (b \bullet c))^*$ is the star-closure of $\{a\} \cup \{bc\}$, which is the language $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, \dots\}$

Regular Expressions (cont.)

Let Σ be an alphabet. Then

① The primitive regular expressions are
$\varnothing$, λ, and $a \in \Sigma$.

② If r_1 and r_2 are regular expressions,
then $r_1 + r_2$, $r_1 \bullet r_2$, $r_1{}^*$, and (r_1) are also
regular expressions.

③ A string is a regular expression if and only if
it can be derived from the primitive regular
expressions by a finite number of applications
of the rules in (2).

Regular Expression Example

- Is $(a + b \bullet c)^* \bullet (c + \phi)$ a regular expression?

- Yes, since it is derived from the primitive regular expressions and repeated applications of the rules in (2) on the previous slide.

- But $(a + b +)$ is not.

Regular Expression Languages

- We can use a regular expression (RE) r to describe an associated language $L(r)$.

1. Ø is a RE denoting the empty set.
2. λ is a RE denoting $\{\lambda\}$.
3. For every $a \in \Sigma$, a is a RE denoting $\{a\}$.

terminating conditions

If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $L(r_1 \bullet r_2) = L(r_1)L(r_2)$
6. $L((r_1)) = L(r_1)$
7. $L(r_1^*) = L(r_1)^*$

recursive definitions

Regular Expression Language Example #1

- What language is defined by the RE $r = a^* \bullet (a + b)$?

$$
\begin{aligned}
L(r) &= L(a^* \bullet (a+b)) \\
&= L(a^*)L(a+b) \\
&= (L(a))^*(L(a) \cup L(b)) \\
&= \{\lambda, a, aa, aaa, \ldots\}(\{a\} \cup \{b\}) \\
&= \{\lambda, a, aa, aaa, \ldots\}\{a, b\} \\
&= \{a, aa, aaa, \ldots, b, ab, aab, \ldots\}
\end{aligned}
$$

Precedence Rules

- Consider the RE $a \bullet b + c$
  - If it's $(a \bullet b) + c$ then $L(a \bullet b + c) = \{ab, c\}$.
  - If it's $a \bullet (b + c)$ then $L(a \bullet b + c) = \{ab, ac\}$.

- To resolve this ambiguity, we use the precedence rules:
  - star-closure is the highest
  - concatenation is the next highest
  - union is the lowest

- Therefore, $a \bullet b + c$ is $(a \bullet b) + c$.

Regular Expression Language Example #2

- Let $\Sigma=\{0, 1\}$. Find regular expression r such that

  $L(r) = \{w \in \Sigma^* : w$ has at least one pair of consecutive zeros$\}$

- RE r must have 00 in it somewhere.

- What comes before or after the 00 is arbitrary.

- Therefore, $r = (0+1)^*00(0+1)^*$

Regular Expression Language Example #3

- Let $\Sigma = \{0, 1\}$. Find regular expression r such that

$$L(r) = \{w \in \Sigma^*: w \text{ has } \underline{\text{no pair }} \text{of consecutive zeros}\}$$

- Whenever there's a 0, it <u>must</u> be followed immediately by a 1.

- There <u>may</u> be any number of leading and trailing 1's.

- There <u>can</u> be a 0 at the very end.

- Therefore, $r = (1^*011^*)^*(0 + \lambda) + 1^*(0 + \lambda)$

Regular Expression Language Example #3 (cont.)

$$r = (1^*011^*)^*(0 + \lambda) + 1^*(0 + \lambda)$$

- Alternate view:
  The RE r can be a repetition of 1's and 01's,
  with a possible 0 at the end.
- Therefore, $r = (1 + 01)^*(0 + \lambda)$.
- Or, $r = 1^*\,(011^*)^*(0 + \lambda)$.
- There is more than one RE for a given language.
- Two REs are equivalent if they denote the same language.

Regular Expressions for Tokens

- Regular expressions can define the syntax of the tokens of a programming language.
  - Tokens are the low-level language elements, such as numbers, strings, and identifiers.

- Example: An identifier is a single letter optionally followed by letters and digits.
  - **`a`**
  - **`alpha`**
  - **`ab123c`**
  - But not: **`3abc`**

```
[a-z]([a-z]|[0-9])*
```

Regular Expressions for Tokens (cont.)

- An number token can be an unsigned integer constant:
  - **12 123 6789**
  - But not: **-12**

```
([0-9])+
```

- Or it can be an unsigned real constant:
  - **12.34 12e3 12e+45 0.123e4 123.45e-12**
  - But not: **+12.34 12. .34**

```
  ([0-9])+.([0-9])+
| ([0-9])+(e|E)([0-9])+
| ([0-9])+(e|E)(+|-)([0-9])+
| ([0-9])+.([0-9])+(e|E)([0-9])+
| ([0-9])+.([0-9])+(e|E)(+|-)([0-9])+
```

Regular Expressions for Tokens (cont.)

- Integer constant: $([0\text{-}9])^+$
- Real constant:

$$
\begin{aligned}
& ([0\text{-}9])^+.([0\text{-}9])^+ \\
| \ & ([0\text{-}9])^+(e|E)([0\text{-}9])^+ \\
| \ & ([0\text{-}9])^+(e|E)(+|-)([0\text{-}9])^+ \\
| \ & ([0\text{-}9])^+.([0\text{-}9])^+(e|E)([0\text{-}9])^+ \\
| \ & ([0\text{-}9])^+.([0\text{-}9])^+(e|E)(+|-)([0\text{-}9])^+
\end{aligned}
$$

```
TOKEN : {
    <INTEGER : (<DIGIT>)+>
  | <REAL1   : (<DIGIT>)+ "." (<DIGIT>)+>
  | <REAL2   : (<DIGIT>)+ <E> (<SIGN>)? (<DIGIT>)+>
  | <REAL3   : (<DIGIT>)+ "." (<DIGIT>)+ <E> (<SIGN>)? (<DIGIT>)+>

  | <#DIGIT : ["0"-"9"]>
  | <#SIGN  : ["+","-"]>
  | <#E     : ["e","E"]>
}
```

Regular Expressions and Regular Languages

- Regular expressions and regular languages are the same concept.

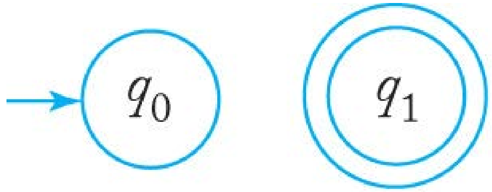
- For every regular expression r, there is a regular language $L = L(r)$.

Theorem 3.1

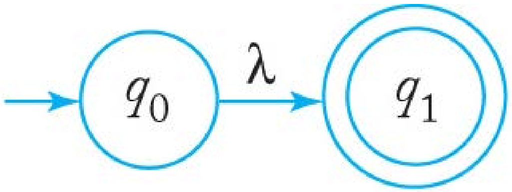
- The textbook proves this by constructing, for any regular expression r, an NFA that accepts $L(r)$.
  - Recall that any language accepted by an NFA or a DFA is regular.

Construct an NFA from a Regular Expression

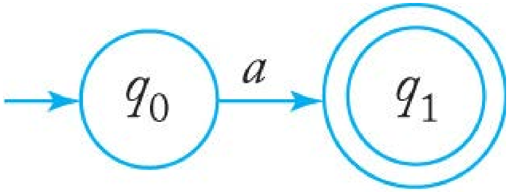
- NFA accepts ϕ



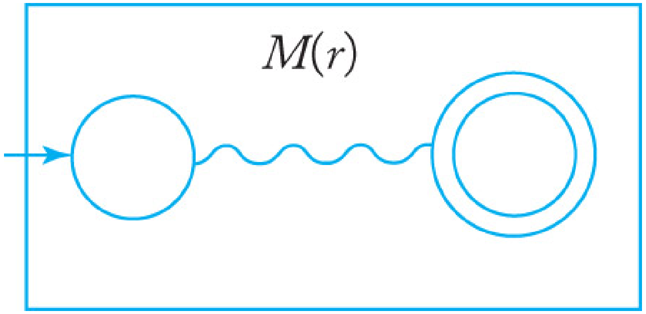
- NFA accepts $\{\lambda\}$



- NFA accepts $\{a\}$

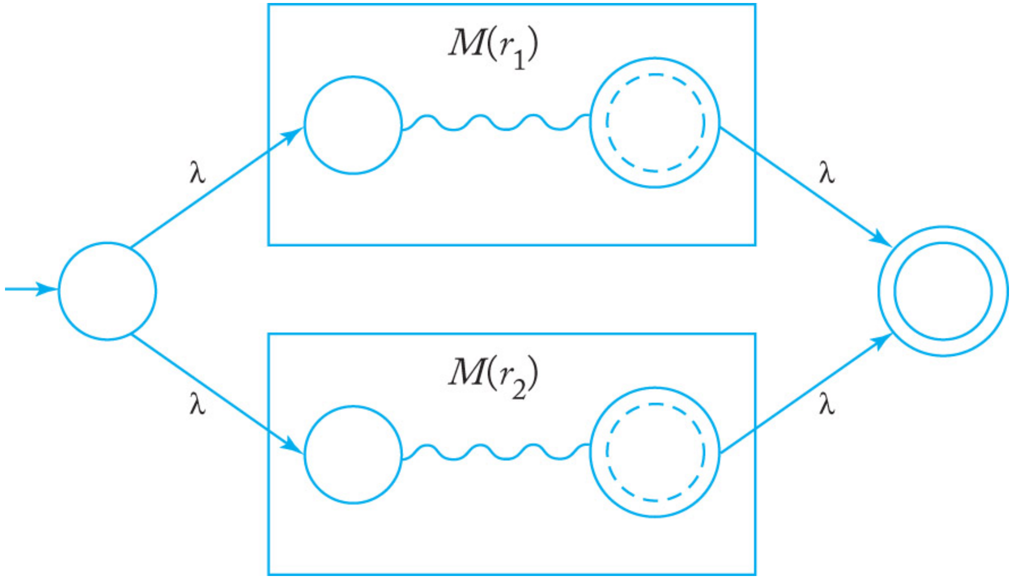


- NFA accepts $L(r)$

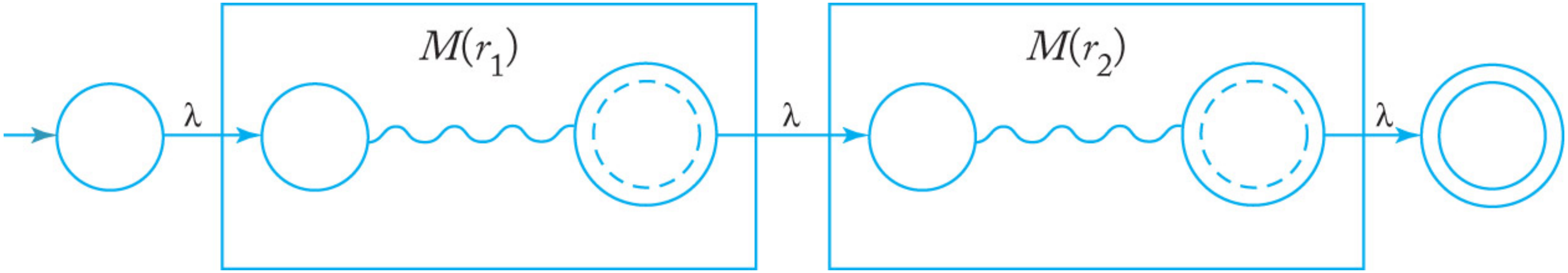


Construct an NFA from an RE (cont.)

- NFA accepts $L(r_1 + r_2)$

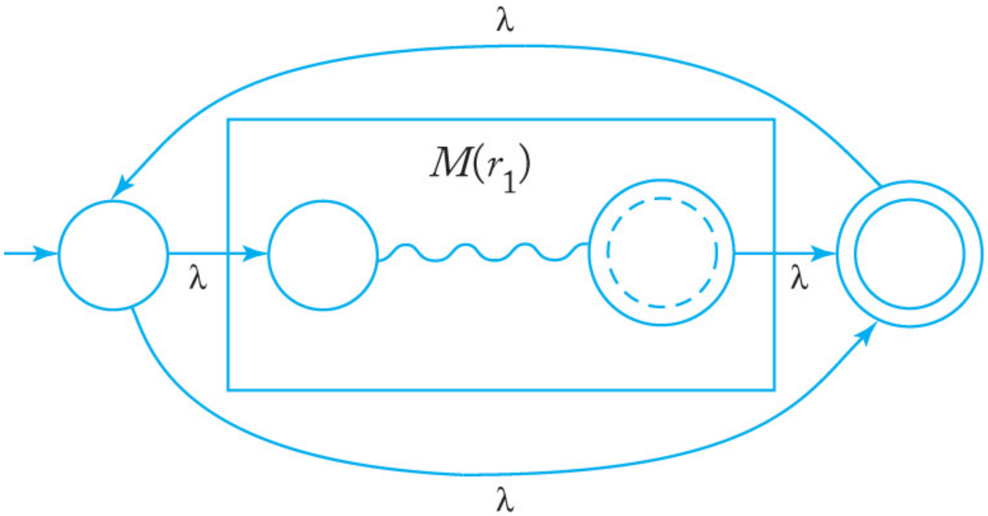


- NFA accepts $L(r_1 r_2)$



Construct an NFA from an RE (cont.)

- NFA accepts $L(r_1{}^*)$

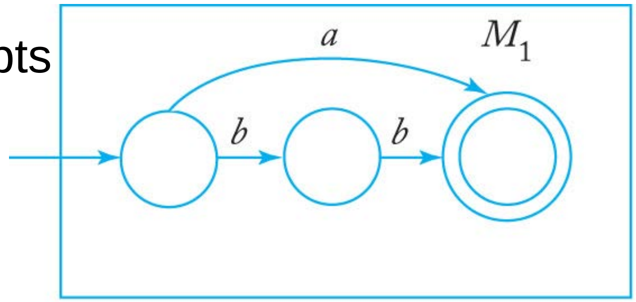


Example: Construct an NFA from an RE

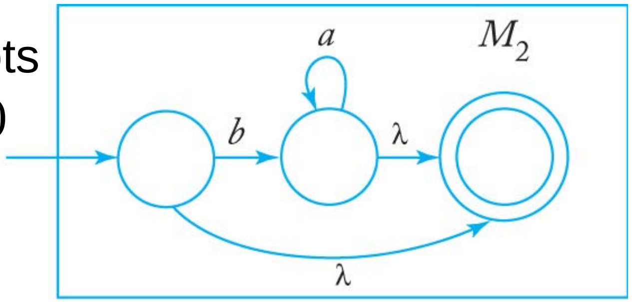
- Construct an NFA that accepts $L(r)$, where RE

$$r = (a + bb)^*(ba^* + \lambda)$$

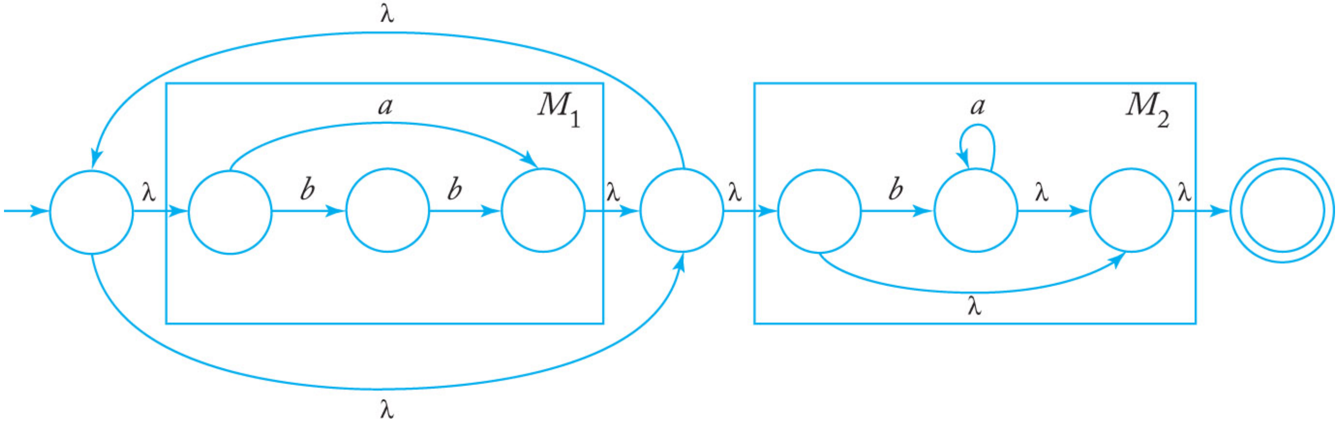
M_1 accepts $L(a + bb)$



M_2 accepts $L(ba^* + \lambda)$



Automaton accepts $L((a + bb)^*(ba^* + \lambda))$



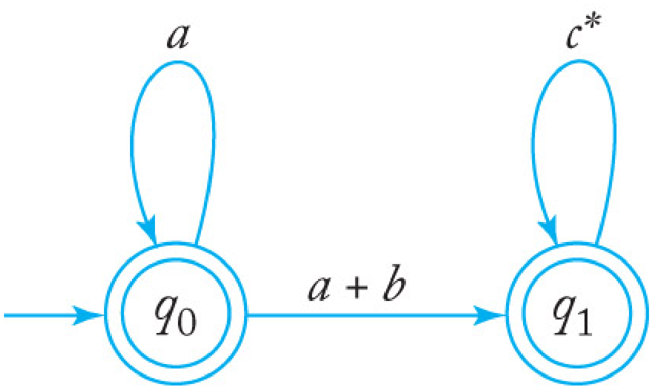
A Rough Algorithm

- Start with putting an initial and final state.

- Recursively, if you see
  - •: put a state,
  - +: put 4 states in a grid of 2x2, lambda transitions to the first two and out of the second two to the enclosing states,
  - *: put lambda transitions to and from the enclosing states,
  - primitive RE: create the NFA and connect to the enclosing states.

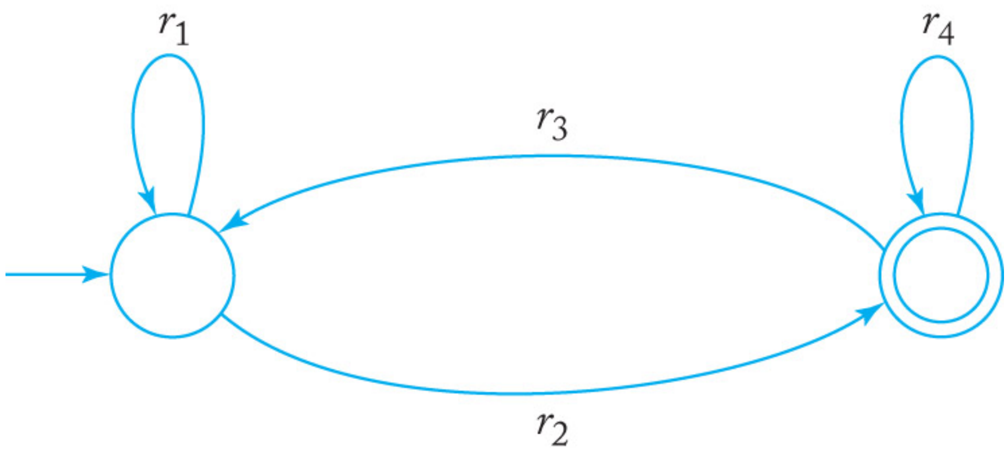
Generalized Transition Graph

- Generalized transition graph (GTG): A transition graph where the edges are labeled with regular expressions.
  - Example:



Generalized Transition Graph (cont.)

- The canonical form of a two-state GTG:



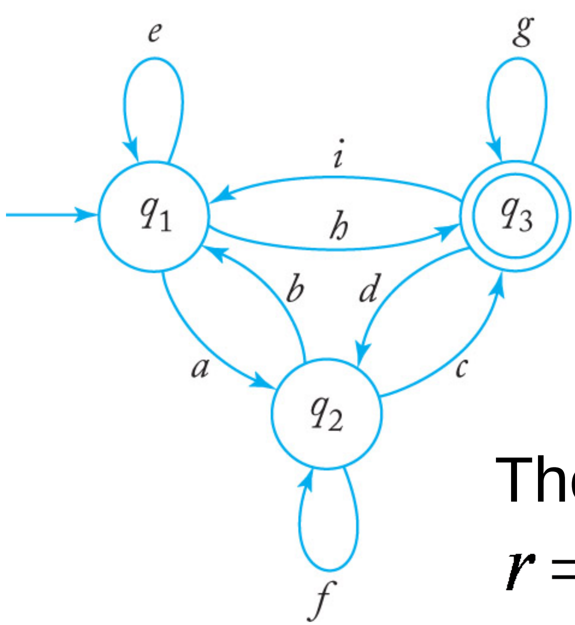
- The RE

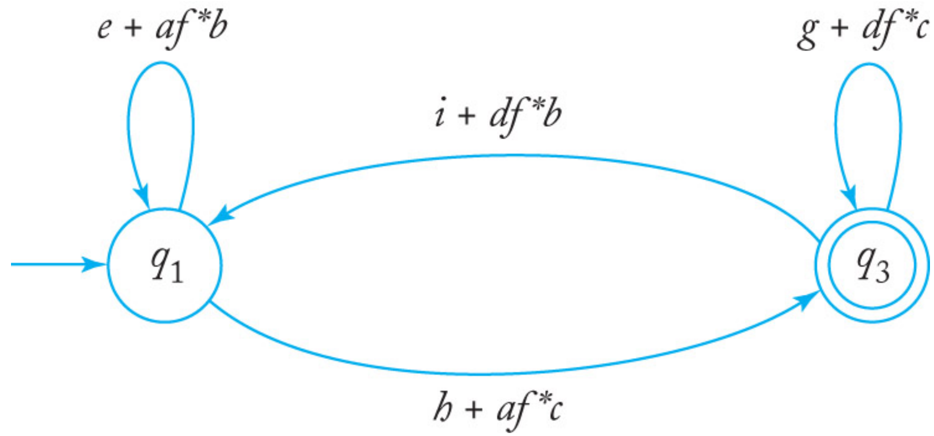
$$r = r_1 {}^* r_2 (r_4 + r_3 r_1 {}^* r_2)^*$$

covers all possible paths and is the graph's RE.

NFA to RE Conversion

- Convert the NFA to a GTG.
- If the GTG has more than two states, remove the extra states one at a time.
  - See the procedure in the textbook:





Then apply

$$r = r_1{}^* r_2 (r_4 + r_3 r_1{}^* r_2)^*$$

NFA to RE Conversion (cont.)

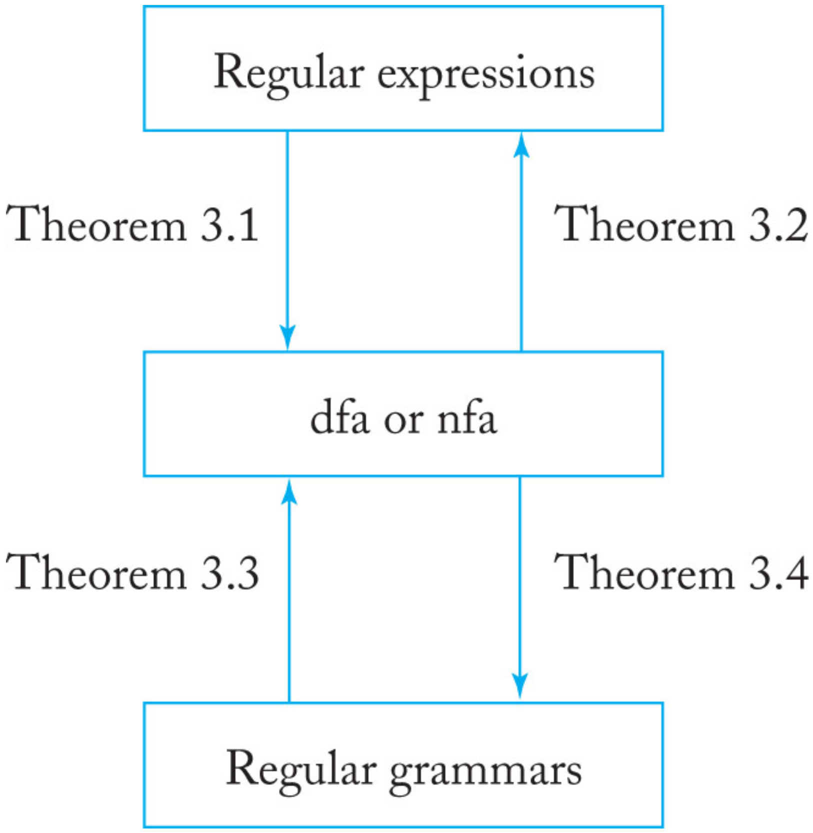
- From an NFA, we can construct a GTG.
  - Recall that any language accepted by an NFA or a DFA is regular.

- From a GTG, we can derive a regular expression.

- Therefore, for every regular language L, there is a regular expression r such that $L=L(r)$.

Theorem 3.2

Regular Expression, Acceptors and Regular Grammars



- Kleene's Theorem: Stephen Kleene proved in 1956 that regular expressions and finite automata are equivalent.

- There is an FA for a language if and only if there is an RE for the language.

Context-Free Languages

Context-Free Languages

- A context-free grammar $G = (V, T, S, P)$ has a more relaxed grammar than a regular grammar.

- All productions in P have the form

$$A \to x$$

  where $A \in V$ and $x \in (V \cup T)^*$
  - It's context-free because any time the variable on the left of a production appears in a sentential form, you can make the substitution.

- A language L is context-free if and only if there is a context-free grammar such that $L = L(G)$.

Context-Sensitive Languages

- A grammar $G = (V, T, S, P)$ is context-sensitive if all productions in P have the form

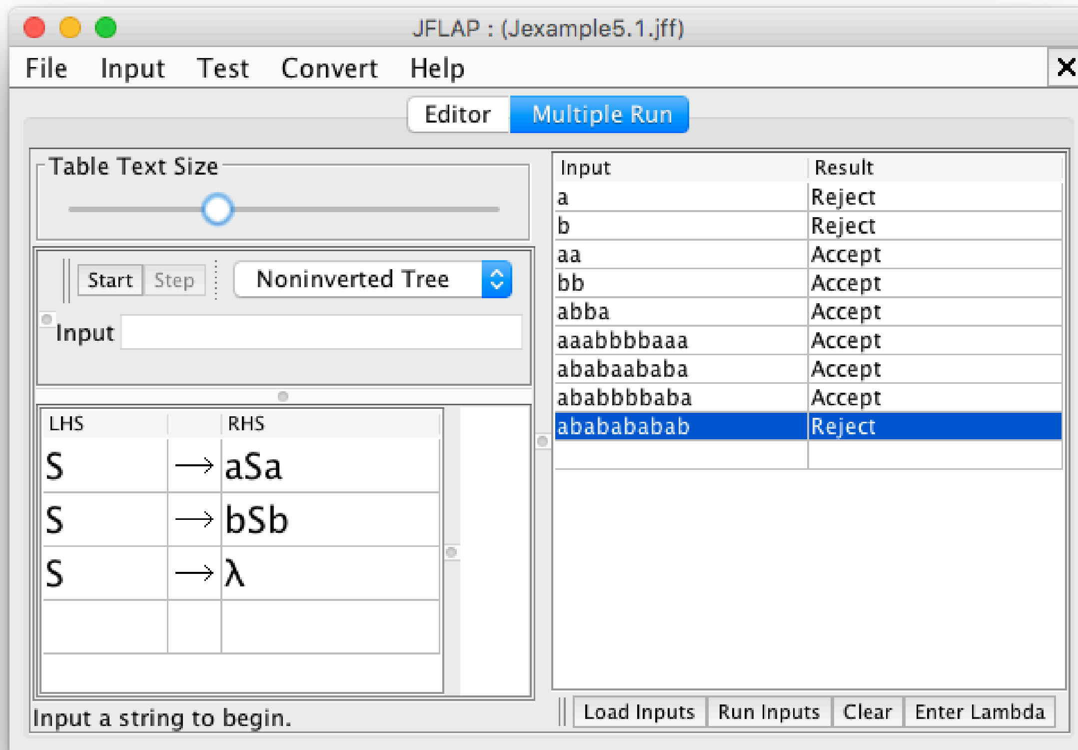
$$\alpha A \beta \rightarrow \alpha x \beta$$

where $A \in V$ and $\alpha, \beta \in (V \cup T)^*$ and $x \in (V \cup T)^+$.

- In other words, you can make the substitution $A \rightarrow x$ in a sentential form only within the context of α and β.

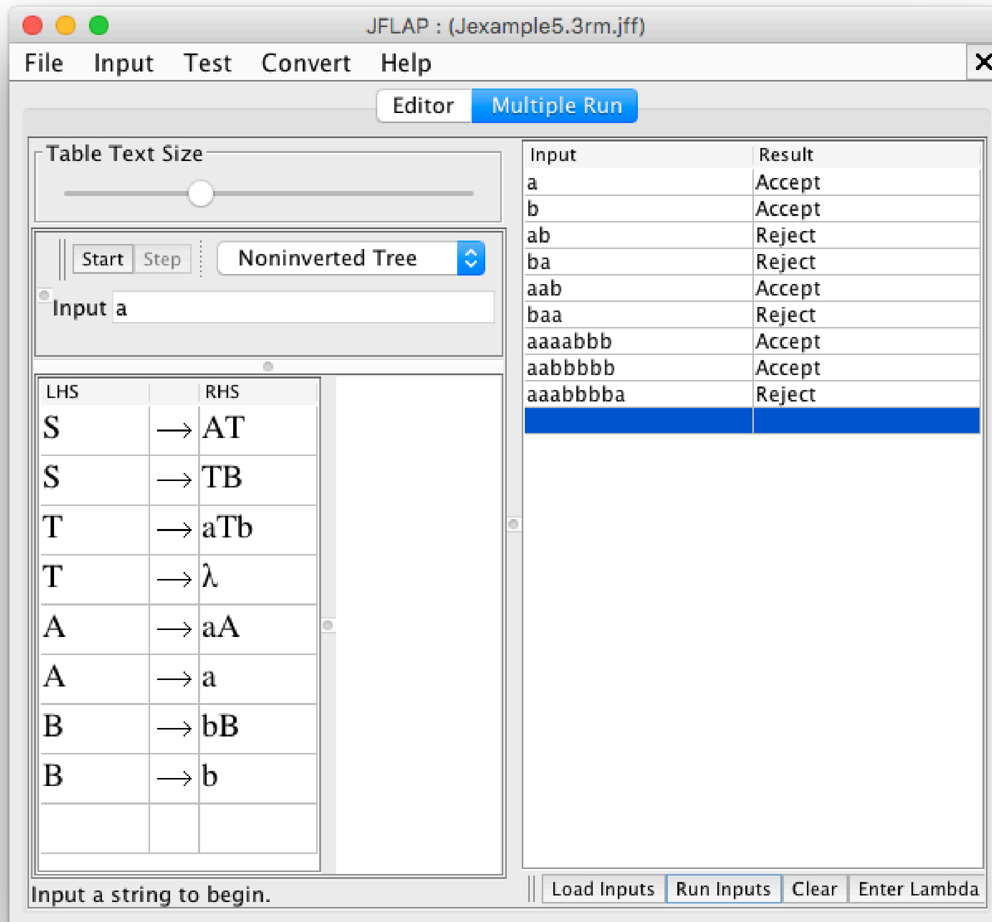
Context-Free Grammar Example #1

- Example 5.1
  - $L(G) = \{ww^R : w \in \{a,b\}^*\}$



Context-Free Grammar Example #2

- Example 5.3
  - $L(G) = \{a^n b^m : n \neq m\}$



Simplifying Context-Free Grammars

- We can convert a context-free grammar to an equivalent grammar that is somehow “simpler”.

- An equivalent but simpler grammar may have more restrictions and is easier to work with.

- Simpler does not necessarily mean fewer production rules.

λ-Free Grammars

- We want to study context-free languages that do not contain the empty string λ.
  - Let L be any context-free language.
  - Let $G = \{V, T, S, P\}$ be a context-free grammar for $L - \{\lambda\}$
  - Create a new grammar by adding the new start symbol S_0 to V and the new rules
    $$S_0 \to S \mid \lambda$$
  - The new grammar will generate L.
  - Therefore, any nontrivial conclusions made for $L - \{\lambda\}$ will also apply to L.

λ-Free Grammars (cont.)

- For any context-free grammar G, we can construct a grammar $\hat{G}$ such that $\hat{G} = L(G) - \{\lambda\}$

- Unless otherwise specified, we will discuss only λ-free context-free languages.

A Substitution Rule

- Let a context-free grammar G contain two different variables A and B.

- Suppose G contains a production of the form

$$A \to x_1 B x_2$$

and a production of the form

$$B \to y_1 \mid y_2 \mid \ldots \mid y_n$$

- Then for each B in the right side of a production, we can substitute each of B's right sides:

$$A \to x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid \ldots \mid x_1 y_n x_2$$

Remove Useless Productions

- A variable of a grammar is useless if:
  - It cannot be reached from the start variable, or
  - It cannot derive a terminal string.

- Example 1 :

$$S \to aSb \mid \lambda \mid A$$
$$A \to aA$$

  - Variable A is useless because it cannot derive a terminal string.

Remove Useless Productions (cont.)

- Example 2 :

$$S \to A$$
$$A \to aA \mid \lambda$$
$$B \to bA$$

  - Even though variable B can derive a terminal string ...
  - It's useless because it cannot be reached from the starting variable S.

Remove Useless Productions (cont.)

- Example 3: $S \to aS \mid A \mid C$

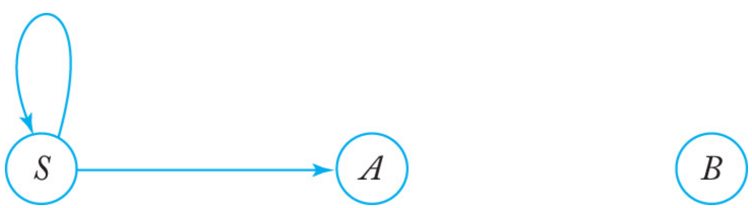
  $A \to a$

  $B \to aa$

  $C \to aCb$

  – C is useless since it cannot derive a terminal string.

  – Draw a dependency graph to show that B is useless since it cannot be reached from S:



  – Therefore: $S \to aS \mid A$

  $A \to a$

Remove λ Productions

- In a context-free grammar, a λ-production is

$$A \to \lambda$$

- Any variable A for which the derivation

$$A \overset{*}{\Rightarrow} \lambda$$

  is possible is nullable.

- To remove λ-productions from a grammar, add new productions where you replace all nullable variables in the right sides of productions with λ in every combination.

Example Removal of λ Productions

- Consider the productions

$$S \to ABaC$$
$$A \to BC$$
$$B \to b \mid \lambda$$
$$C \to D \mid \lambda$$
$$D \to d$$

- Variables A, B, and C are nullable.
  - Replace each of them with λ in every combination.
  - Example: Add to production A the rule with B replaced with λ and the rule with C replaced with λ:

$$A \to BC \mid B \mid C$$

Example Removal of λ Productions (cont.)

$$S \to ABaC$$
$$A \to BC$$
$$B \to b \mid \lambda$$
$$C \to D \mid \lambda$$
$$D \to d$$

- Similarly for production rule S, add rules where you replace A, B, and C in $ABaC$ with λ in every combination:
  - Replace A with λ to get BaC
  - Replace B with λ to get AaC
  - Replace C with λ to get ABa
  - Replace both A and B with λ to get aC, etc.

$$S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a$$
$$A \to BC \mid B \mid C$$
$$B \to b$$
$$C \to D$$
$$D \to d$$

Remove Unit Productions

- A unit production in a context-free grammar has the form $A \to B$ where A and B are variables.

- We also want to remove unit productions.

Example Removal of Unit Productions

- Consider the productions

$$S \to Aa \mid B$$
$$B \to A \mid bb$$
$$A \to a \mid bc \mid B$$

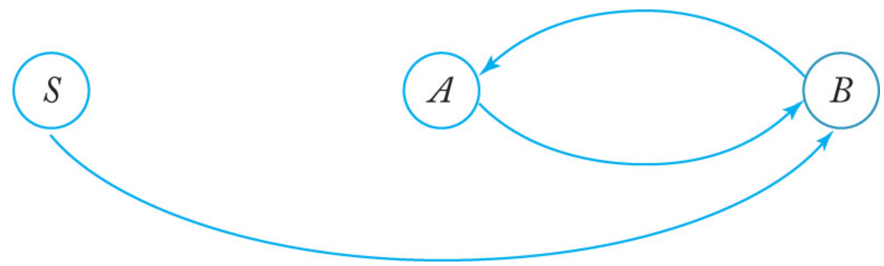
- Start with the non-unit productions

$$S \to Aa$$
$$A \to a \mid bc$$
$$B \to bb$$

Example Removal of Unit Productions (cont.)

$$S \to Aa \mid B$$
$$B \to A \mid bb$$
$$A \to a \mid bc \mid B$$

$$S \to Aa$$
$$A \to a \mid bc$$
$$B \to bb$$



- Draw the dependency graph for the unit productions to add new rules:

$$S \to a \mid bc \mid bb$$
$$A \to bb$$
$$B \to a \mid bc$$

- The equivalent grammar:
  - Note that B is now useless.

$$S \to a \mid bc \mid bb \mid Aa$$
$$A \to a \mid bb \mid bc$$
$$B \to a \mid bb \mid bc$$