Theory of Computation
Week 4

Much of the material on this slides comes from the recommended textbook by Elaine Rich

Announcement

Assignment 1 Released

- ❑ Due: 29/03/2020

Detailed content

Weekly program

- ✓ Week 1 – Background knowledge revision: logic, sets, proof techniques
- ✓ Week 2 – Languages and strings. Hierarchies. Computation. Closure properties
- ✓ Week 3 – Finite State Machines: non-determinism vs. determinism

Week 4 – Regular languages: expressions and grammars

- ❑ Week 5 – Non regular languages: pumping lemma. Closure
- ❑ Week 6 – Context-free languages: grammars and parse trees
- ❑ Week 7 – Pushdown automata
- ❑ Week 8 – Non context-free languages: pumping lemma and decidability. Closure
- ❑ Week 9 – Decidable languages: Turing Machines
- ❑ Week 10 – Church-Turing thesis and the unsolvability of the Halting Problem
- ❑ Week 11 – Decidable, semi-decidable and undecidable languages (and proofs)
- ❑ Week 12 – Revision of the hierarchy. Safety-critical systems
- ❑ Week 13 – Extra revision (if needed)

DETERMINISTIC FSM
Definition

- A Finite State Machine M is a quintuple

  $M = (K, \Sigma, \delta, s, A)$, where:

  – K is a finite set of states
  – Σ is an alphabet
  – δ is the transition function from $(K \times \Sigma)$ to K
  – $s \in K$ is the initial state, and
  – $A \subseteq K$ is the set of accepting states

NONDETERMINISTIC FSM
Definition

- A Finite State Machine M is a quintuple

  $M = (K, \Sigma, \Delta, s, A)$, where:

  – K is a finite set of states
  – Σ is an alphabet
  – Δ is the transition relation. It is a finite subset of $(K \times (\Sigma \cup \{\varepsilon\}) \times K$
  – $s \in K$ is the initial state, and
  – $A \subseteq K$ is the set of accepting states

SUMMARY

- A language is ***regular*** iff it is accepted by some FSM

- Given any DFSM M, there exists an algorithm *minDFSM* that constructs a minimal DFSM that also accepts $L(M)$.

- Given any NDFSM M, there exists an algorithm *ndfsmtodfsm* that constructs a DFSM that also accepts $L(M)$.

Week 04 Videos

You already know

- ❑ Regular Expression
  - ❑ 8 rules for forming regular expression
  - ❑ How to read a regular expression
  - ❑ Relation of RE with language
- ❑ Regular Grammar
  - ❑ Formal Definition of Regular Grammar
  - ❑ What is rule in a regular grammar
  - ❑ Condition for rules in regular grammar
  - ❑ How strings are generated from regular grammar

Videos to watch before lecture

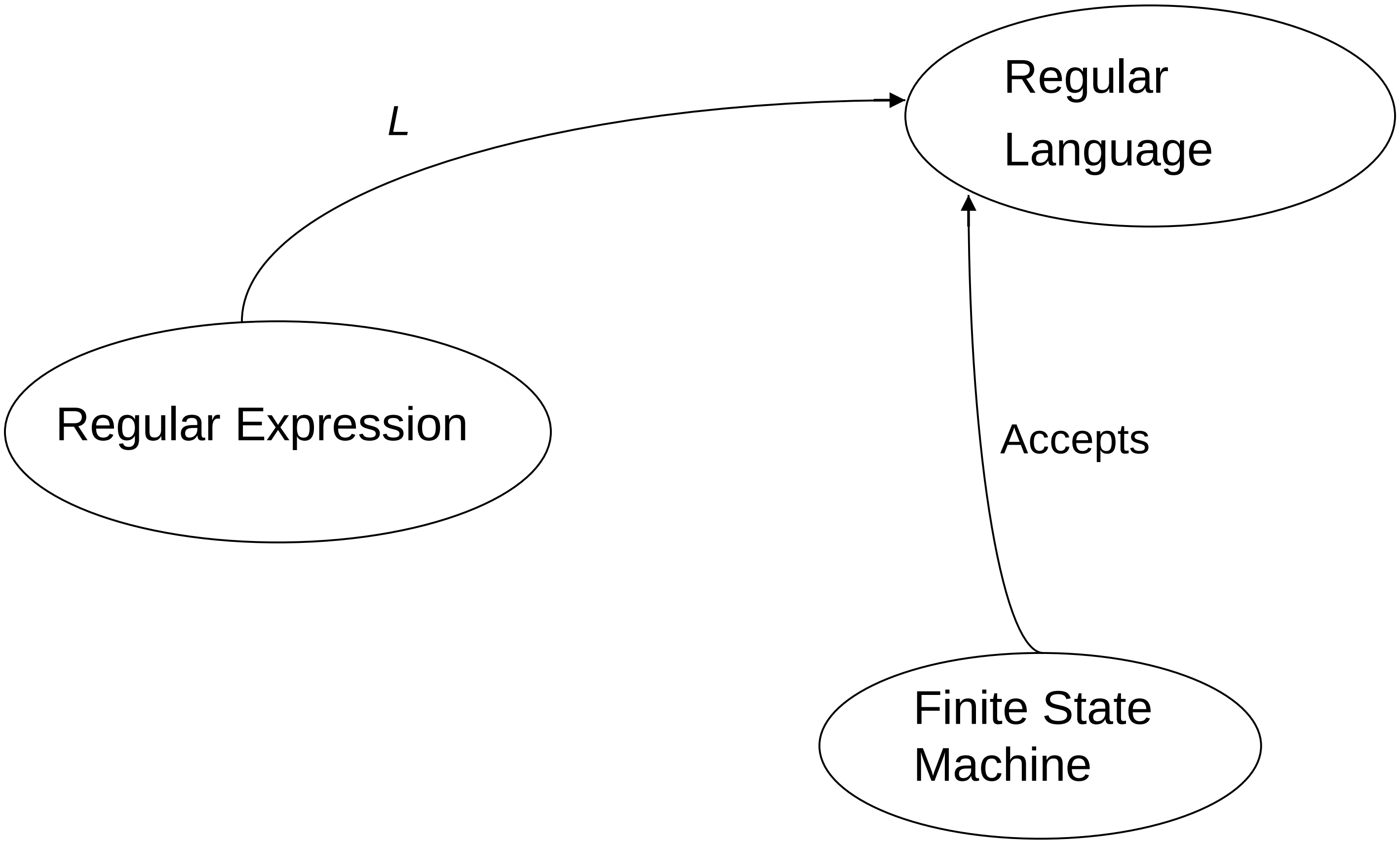
Additional videos to watch for this week

Week 04 Lecture Outline

Regular languages: expressions and grammars

- ❑ Regular Expression
- ❑ Kleene’s theorem
- ❑ Regular Grammar
- ❑ Conversion between Regular Grammar and FSM
- ❑ Simplifying Regular Expression
- ❑ More on regular expression

REGULAR LANGUAGES



REGULAR EXPRESSIONS

The regular expressions over an alphabet Σ are all and only the strings that can be obtained as follows:

1. $\varnothing$ is a regular expression.
2. ε is a regular expression.
3. Every element of Σ is a regular expression.
4. If α , β are regular expressions, then so is $\alpha\beta$.
5. If α , β are regular expressions, then so is $\alpha\cup\beta$.
6. If α is a regular expression, then so is α^*.
7. α is a regular expression, then so is α^+.
8. If α is a regular expression, then so is (α).

REGULAR EXPRESSIONS
Examples

If $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, the following are regular expressions:

$\emptyset$

ε

$\mathtt{a}$

$(\mathtt{a} \cup \mathtt{b})^*$

$\mathtt{abba} \cup \varepsilon$

REGULAR EXPRESSIONS
Rules

Define L, a **semantic interpretation function** for regular expressions:

1. $L(\varnothing) = \varnothing$.
2. $L(\varepsilon) = \{\varepsilon\}$.
3. $L(c)$, where $c \in \Sigma = \{c\}$.
4. $L(\alpha\beta) = L(\alpha)\ L(\beta)$.
5. $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$.
6. $L(\alpha^*) = (L(\alpha))^*$.
7. $L(\alpha^+) = L(\alpha\alpha^*) = L(\alpha)\ (L(\alpha))^*$.
   If $L(\alpha)$ is equal to $\varnothing$, then $L(\alpha^+)$ is also equal to $\varnothing$. Otherwise $L(\alpha^+)$ is the language that is formed by concatenating together one or more strings drawn from $L(\alpha)$.
8. $L((\alpha)) = L(\alpha)$.

REGULAR EXPRESSIONS
Rules

- Rules 1, 3, 4, 5, and 6 give the language its power to define sets.
- Rule 8 has as its only role grouping other operators.
- Rules 2 and 7 appear to add functionality to the regular expression language, but they don't.

2. ε is a regular expression.

7. α is a regular expression, then so is α^+.

REGULAR EXPRESSIONS
Example

$$L((\mathtt{a} \cup \mathtt{b})^*\mathtt{b}) = L((\mathtt{a} \cup \mathtt{b})^*)\ L(\mathtt{b})$$

$$= (L((\mathtt{a} \cup \mathtt{b})))^*\ L(\mathtt{b})$$

$$= (L(\mathtt{a}) \cup L(\mathtt{b}))^*\ L(\mathtt{b})$$

$$= (\{\mathtt{a}\} \cup \{\mathtt{b}\})^*\ \{\mathtt{b}\}$$

$$= \{\mathtt{a}, \mathtt{b}\}^*\ \{\mathtt{b}\}.$$

Go through a loop zero or more times, picking a single a or b each time. Then concatenate b.

REGULAR EXPRESSIONS
More examples

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^*: |w| \text{ is even}\}$

REGULAR EXPRESSIONS
More examples

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^*: |w| \text{ is even}\}$

$$((\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b}))^*$$

$$(\mathtt{aa} \cup \mathtt{ab} \cup \mathtt{ba} \cup \mathtt{bb})^*$$

REGULAR EXPRESSIONS
More examples

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^*: |w| \text{ is even}\}$

$((\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b}))^*$

$(\mathtt{aa} \cup \mathtt{ab} \cup \mathtt{ba} \cup \mathtt{bb})^*$

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^*: w \text{ contains an odd number of } \mathtt{a}\text{'s}\}$

REGULAR EXPRESSIONS
More examples

$L = \{w \in \{\texttt{a}, \texttt{b}\}^*: |w| \text{ is even}\}$

$((\texttt{a} \cup \texttt{b})\ (\texttt{a} \cup \texttt{b}))^*$

$(\texttt{aa} \cup \texttt{ab} \cup \texttt{ba} \cup \texttt{bb})^*$

$L = \{w \in \{\texttt{a}, \texttt{b}\}^*: w \text{ contains an odd number of } \texttt{a}\text{'s}\}$

$\texttt{b}^*\ (\texttt{ab}^*\texttt{ab}^*)^*\ \texttt{a}\ \texttt{b}^*$

$\texttt{b}^*\ \texttt{a}\ \texttt{b}^*\ (\texttt{ab}^*\texttt{ab}^*)^*$

REGULAR EXPRESSIONS
Idioms

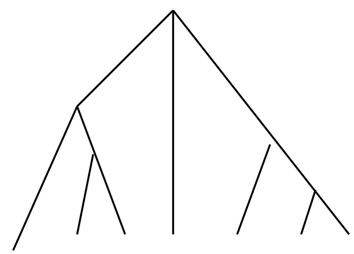
$(\alpha \cup \varepsilon)$ → is frequently read as "optional α"

$(\mathtt{a} \cup \mathtt{b})^*$ → describes all possible strings composed of characters $\mathtt{a}$ and $\mathtt{b}$

NOTE: $(\mathtt{a} \cup \mathtt{b})^* \neq \mathtt{a}^* \cup \mathtt{b}^*$ and $(\mathtt{ab})^* \neq \mathtt{a}^*\mathtt{b}^*$

REGULAR EXPRESSIONS
Operator precedence

	Regular Expressions	Arithmetic Expressions
Highest	Kleene star	exponentiation
	concatenation	multiplication
Lowest	union	addition



$a\ b^* \cup c\ d^*$

$x\ y^2 + i\ j^2$

REs vs. FSMs
Kleene's theorem

Finite state machines and regular expressions define the same class of languages. To prove this, we must show:

Theorem 1*:* Any language that can be defined with a regular expression can be accepted by some FSM and so is regular.

Theorem 2: Every regular language (i.e., every language that can be accepted by some DFSM) can be defined with a regular expression.

REs vs. FSMs
Kleene's theorem

***Theorem 1**:* Any language that can be defined with a regular expression can be accepted by some FSM and so is regular.

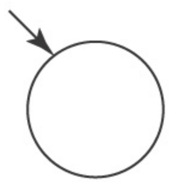
Proof: By construction. We will show that given any regular expression α we can construct a FSM *M* which accepts the same language, i.e. $L(\alpha) = L(M)$

We must show how to build FSMs to accept languages defined by regular expressions

REEs vs. FSMs
Kleene's theorem

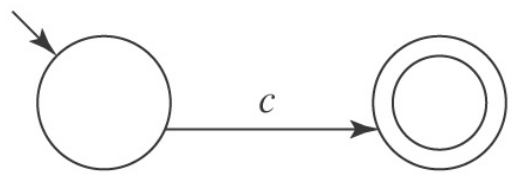
***Theorem 1**:* Any language that can be defined with a regular expression can be accepted by some FSM and so is regular.

If $\alpha = \varnothing$:



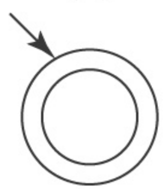
$L(\varnothing) = \varnothing$

If $\alpha = c \in \Sigma$:



$L(c)$, where $c \in \Sigma = \{c\}$

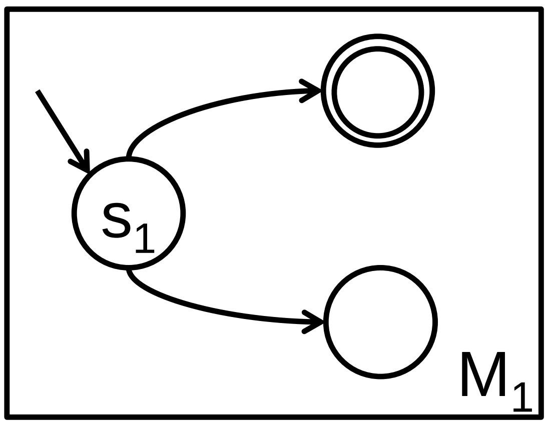
If $\alpha = \varepsilon$

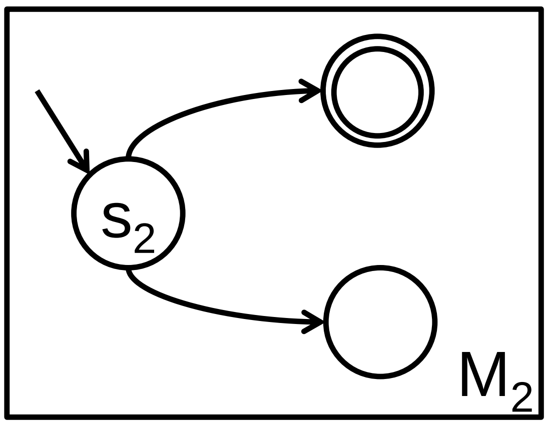


$L(\varepsilon) = \{\varepsilon\}$.

REs vs. FSMs
Kleene's theorem

- β and γ be regular expression over Σ
- accepted by FSM M_1 and M_2



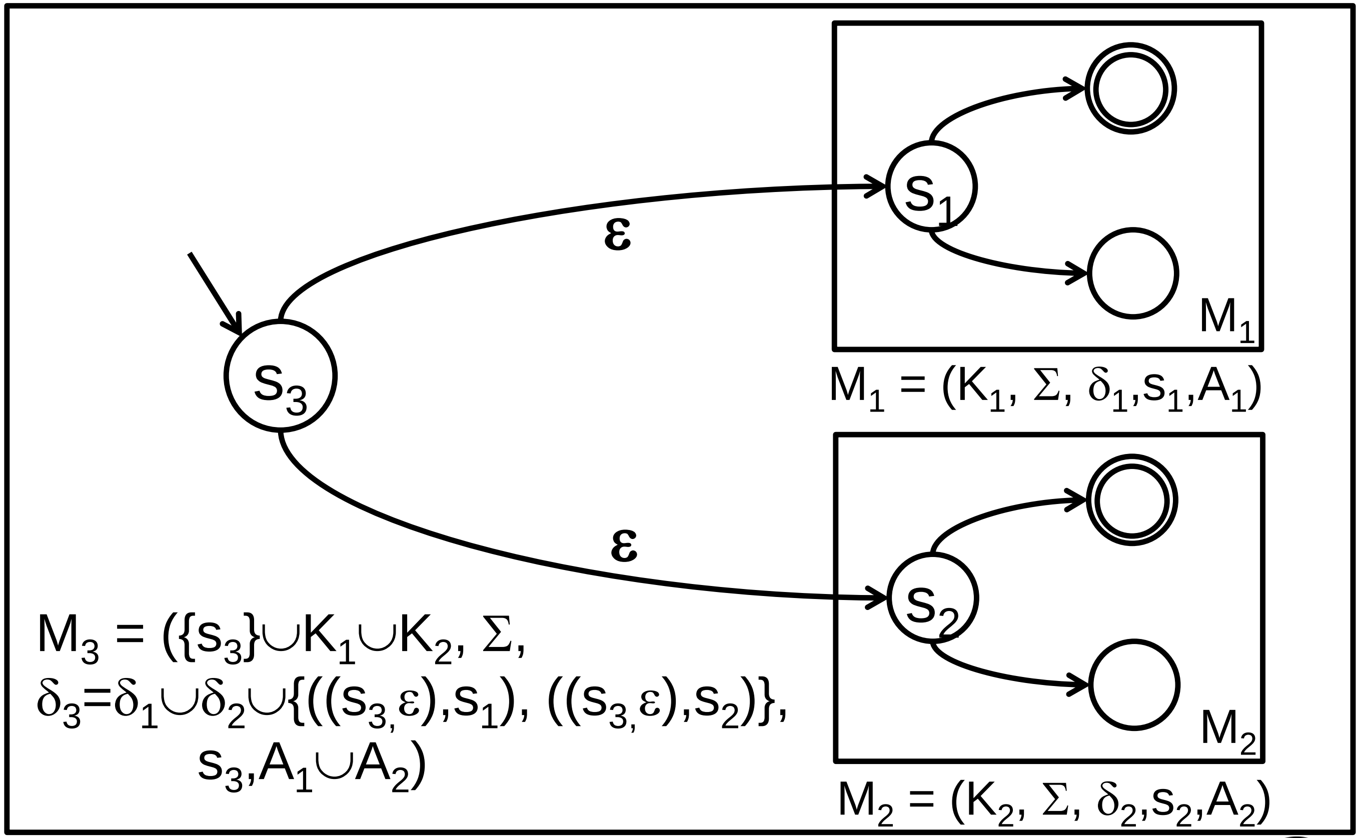


$M1 = (K_1, \Sigma, \delta_1, s_1, A_1)$

$M2 = (K_2, \Sigma, \delta_2, s_2, A_2)$

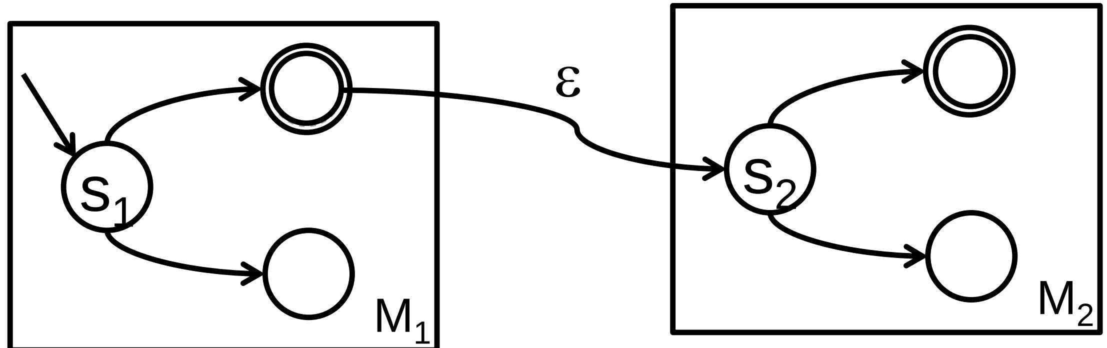
REEs vs. FSMs
Kleene's theorem

If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct M_3 such that $L(M_3)=L(\alpha)=L(\beta) \cup L(\gamma)$:



REEs vs. FSMs Kleene's theorem

If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct M_3 such that $L(M_3)=L(\alpha)=L(\beta)\, L(\gamma)$:



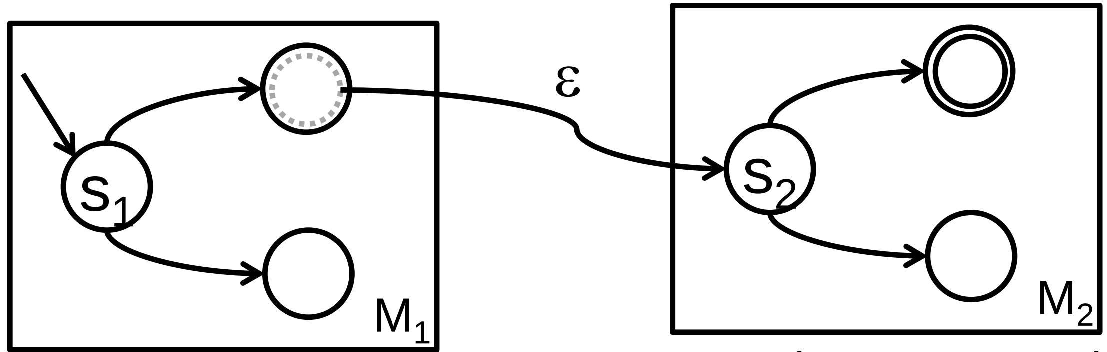
$M_1 = (K_1, \Sigma, \delta_1, s_1, A_1)$

$M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$

$$M_3 = (K_1 \cup K_2, \Sigma, \delta_3 = \delta_1 \cup \delta_2 \cup \{((q, \varepsilon), s_2) : q \in A_1\}, s_1, A_2)$$

REs vs. FSMs
Kleene's theorem

If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct M_3 such that $L(M_3)=L(\alpha)=L(\beta)\ L(\gamma)$:



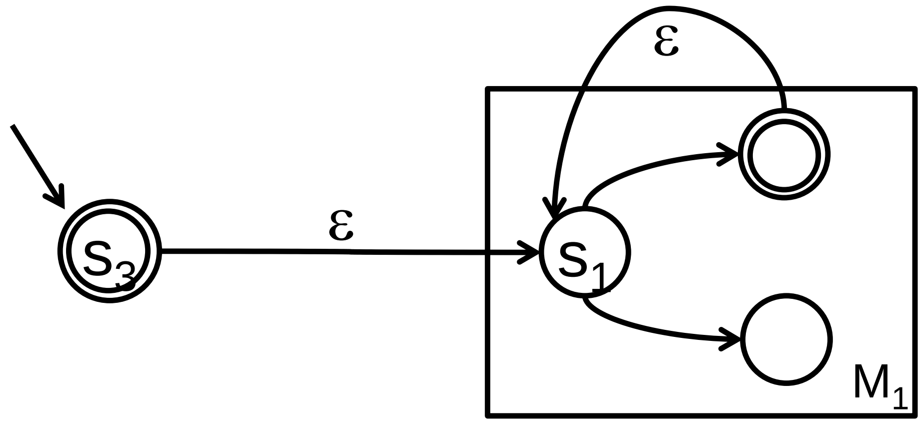
$M_1 = (K_1, \Sigma, \delta_1, s_1, A_1)$

$M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$

$$M_3 = (K_1 \cup K_2, \Sigma, \delta_3 = \delta_1 \cup \delta_2 \cup \{((q, \varepsilon), s_2) : q \in A_1\}, s_1, A_2)$$

REEs vs. FSMs
Kleene's theorem

If α is the regular expression β^* and $L(\beta)$ is regular, then we construct M_3 such that $L(M_3)=L(\alpha)=L(\beta)^*$:



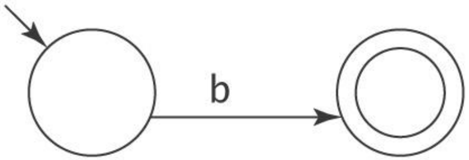
$M_1 = (K_1, \Sigma, \delta_1, s_1, A_1)$

$M_3 = (\{s_3\} \cup K_1, \Sigma, \delta_3 = \delta_1 \cup \{((s_3, \varepsilon), s_1), ((q, \varepsilon), s_1) : q \in A_1\}, s_3, \{s_3\} \cup A_1)$

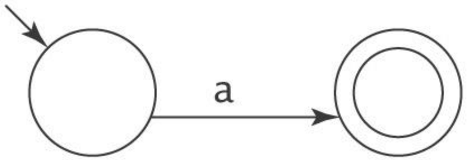
KLEENE'S THEOREM
Example

(b ∪ ab)*

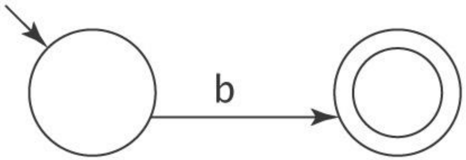
An FSM for b



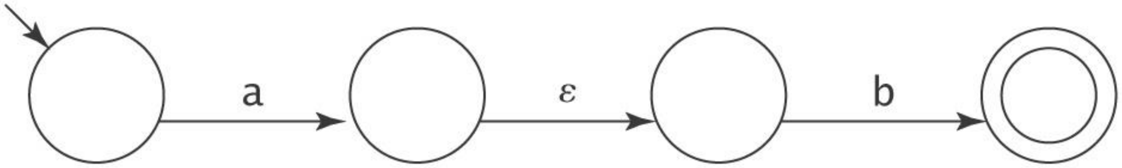
An FSM for a



An FSM for b



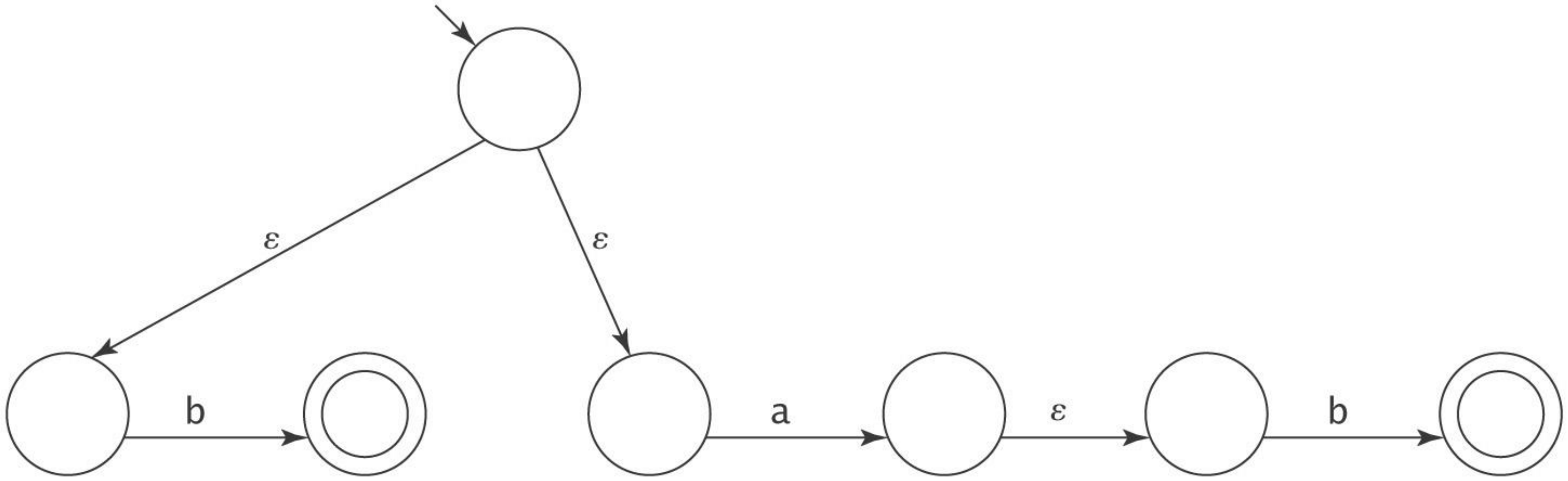
An FSM for ab:



KLEENE'S THEOREM
Example

(b ∪ ab)*

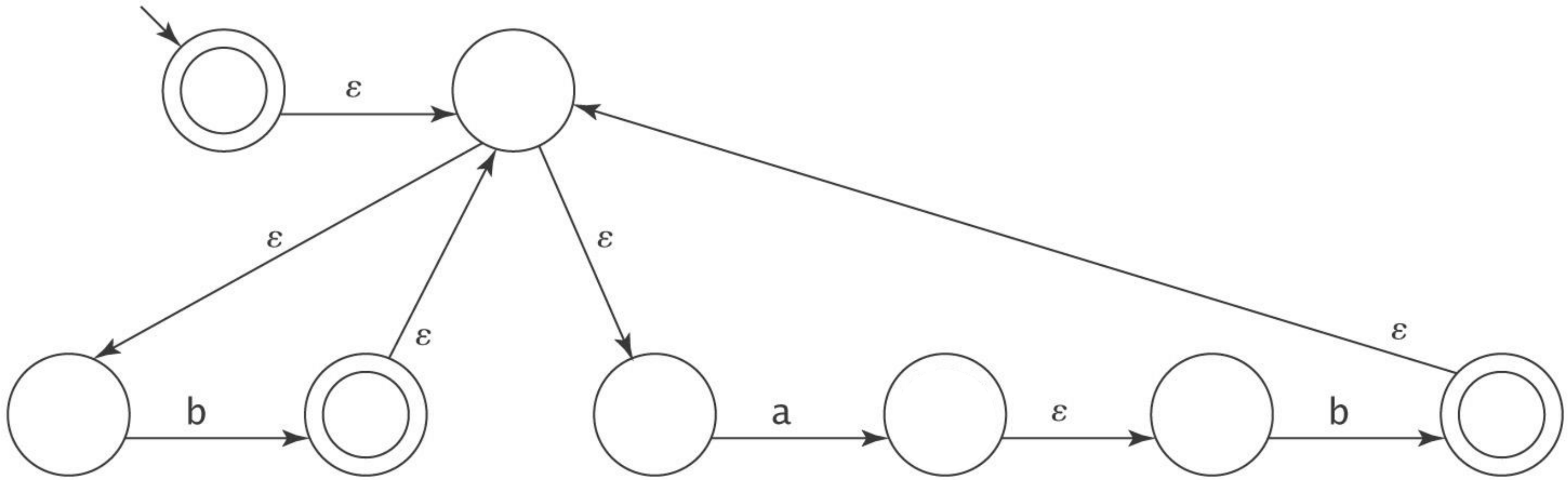
An FSM for (b ∪ ab):



KLEENE'S THEOREM
Example

$(b \cup ab)^*$

An FSM for $(b \cup ab)^*$:



REs vs. FSMs
Kleene's theorem

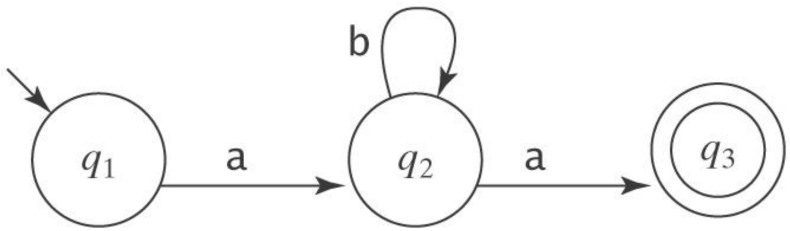
Theorem 2: Every regular language can be defined with a regular expression.

Proof: This is equivalent to showing that for every FSM there is a corresponding regular express. We'll show this construction.

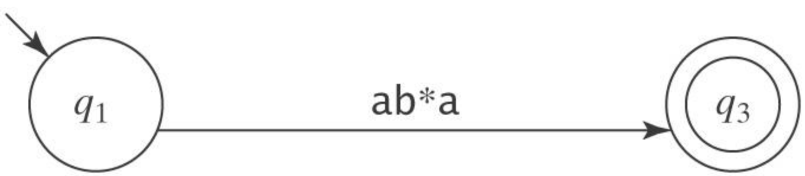
The key idea is that we'll allow arbitrary regular expressions to label the transitions of an FSM.

KLEENE'S THEOREM
Tricks

Let *M* be:



Suppose we rip out state 2:



fsmtoregexheuristic

fsmtoregexheuristic(*M*: FSM) =

1. Remove unreachable states from *M*.
2. If *M* has no accepting states then return $\varnothing$.
3. If the start state of *M* is part of a loop, create a new start state *s* and connect *s* to *M*'s start state via an ε-transition.
4. If there is more than one accepting state of *M* or there are any transitions out of any of them, create a new accepting state and connect each of *M*'s accepting states to it via an ε-transition. The old accepting states no longer accept.
5. If *M* has only one state then return ε.
6. Until only the start state and the accepting state remain do:
   - 6.1 Select *rip* (not *s* or an accepting state).
   - 6.2 Remove *rip* from *M*.
   - 6.3 *Modify the transitions among the remaining states so *M* accepts the same strings.
7. Return the regular expression that labels the one remaining transition from the start state to the accepting state.

fsmtoregex

fsmtoregex(*M*: FSM) =

1. M' = *standardize*(*M*: FSM).
2. Return *buildregex*(M').

standardize(*M*: FSM) =

1. Remove unreachable states from *M*.
2. If necessary, create a new start state.
3. If necessary, create a new accepting state.
4. If there is more than one transition between states *p* and *q*, collapse them.
5. If any transitions are missing, create them with label $\varnothing$.

fsmtoregex

buildregex(*M*: FSM) =

1. If *M* has no accepting states then return $\varnothing$.
2. If *M* has only one state, then return ε.
3. Until only the start and accepting states remain do:
   - 3.1 Select some state *rip* of *M*.
   - 3.2 For every transition from *p* to *q*, if both *p* and *q* are not *rip* then do
     - Compute the new label R' for the transition from *p* to *q*:

$$R'(p, q) = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q)$$

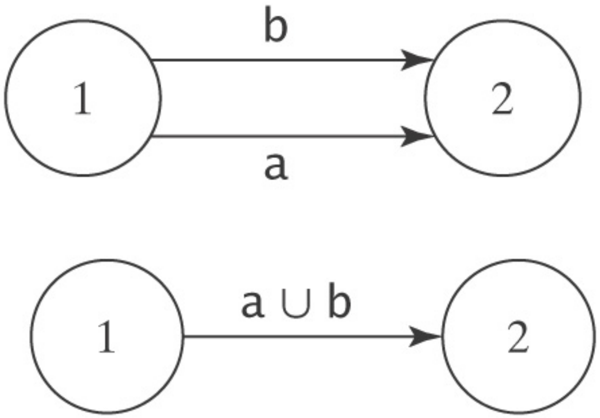
   - 3.3 Remove *rip* and all transitions into and out of it.
4. Return the regular expression that labels the transition from the start state to the accepting state.

MODIFICATIONS TO *M* BEFORE WE START

We require that, from every state other than the accepting state there must be exactly one transition to every state (including itself) except the start state. And into every state other than the start state there must be exactly one transition from every state (including itself) except the accepting state.

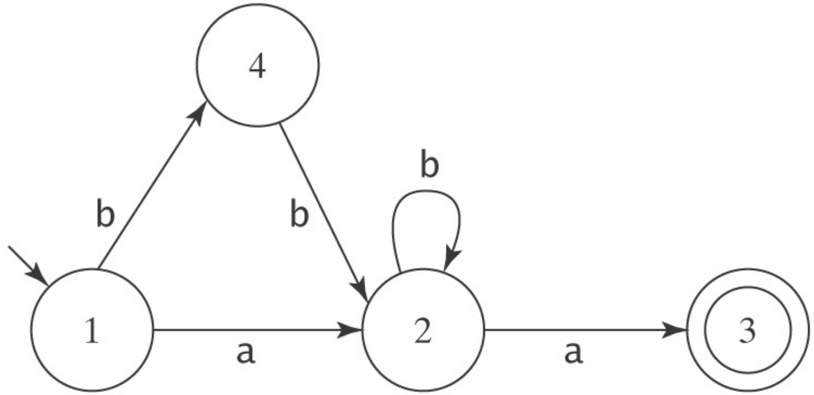
1. If there is more than one transition between states *p* and *q*, collapse them into a single transition:

becomes:

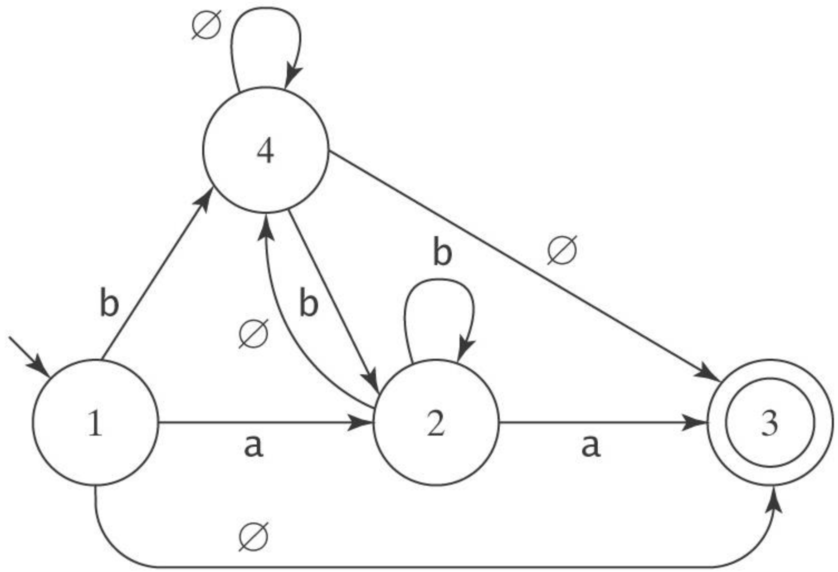


MODIFICATIONS TO *M* BEFORE WE START

2. If any of the required transitions are missing, add them:

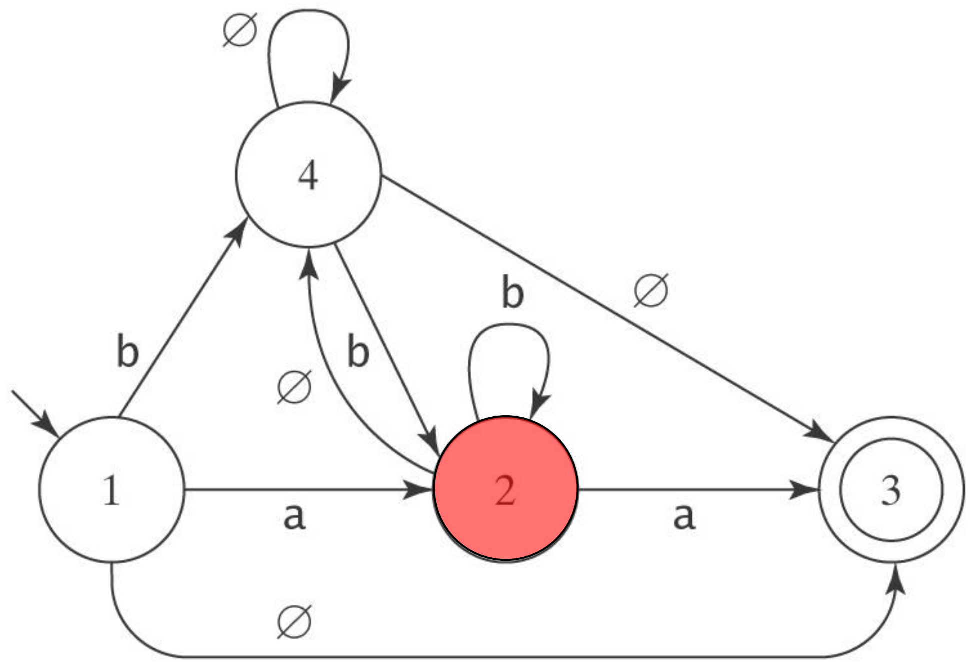


becomes:



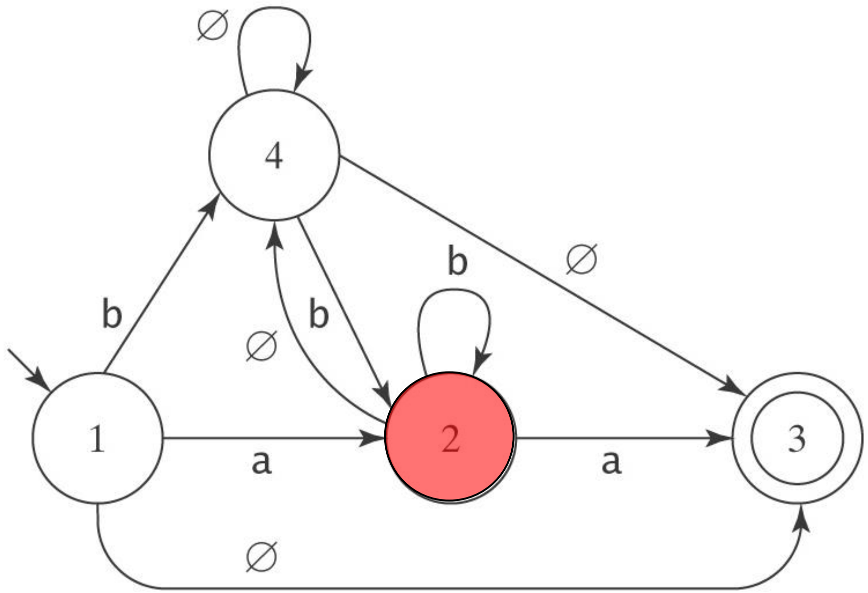
MODIFICATIONS TO *M* BEFORE WE START

3. Choose a state. Rip it out. Restore functionality.



Suppose we rip state 2.

MODIFICATIONS TO *M* BEFORE WE START



Consider any pair of states *p* and *q*. Once we remove *rip*, how can *M* get from *p* to *q*?

- It can still take the transition that went directly from *p* to *q*, or
- It can take the transition from *p* to *rip*. Then, it can take the transition from *rip* back to itself zero or more times. Then it can take the transition from *rip* to *q*.

DEFINITION OF $R(p, q)$

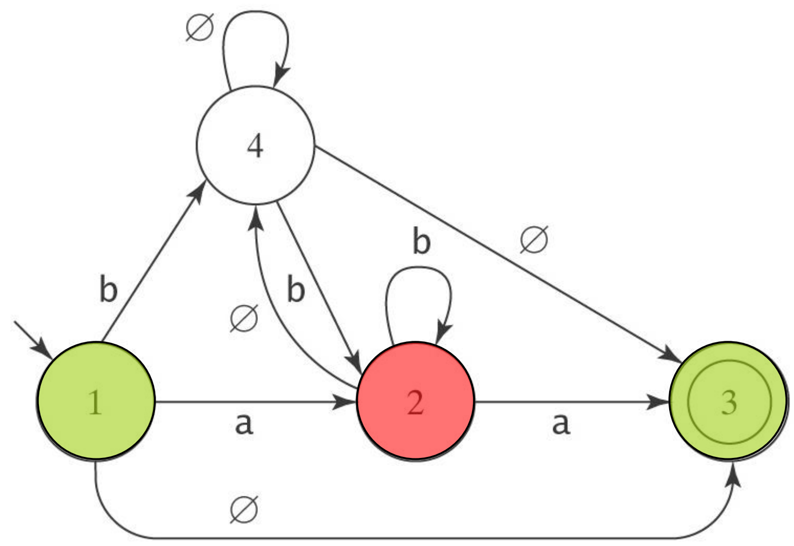
After removing *rip*, the new regular expression that should label the transition from p to q is:

$R(p, q)$		/* Go directly from p to q
	$\cup$	/* or
$R(p, rip)$		/* Go from p to *rip*, then
$R(rip, rip)^*$		/* Go from *rip* back to itself any number of times, then
$R(rip, q)$		/* Go from *rip* to q

Without the comments, we have:

$$R' = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q)$$

KLEENE'S THEOREM
The harder example

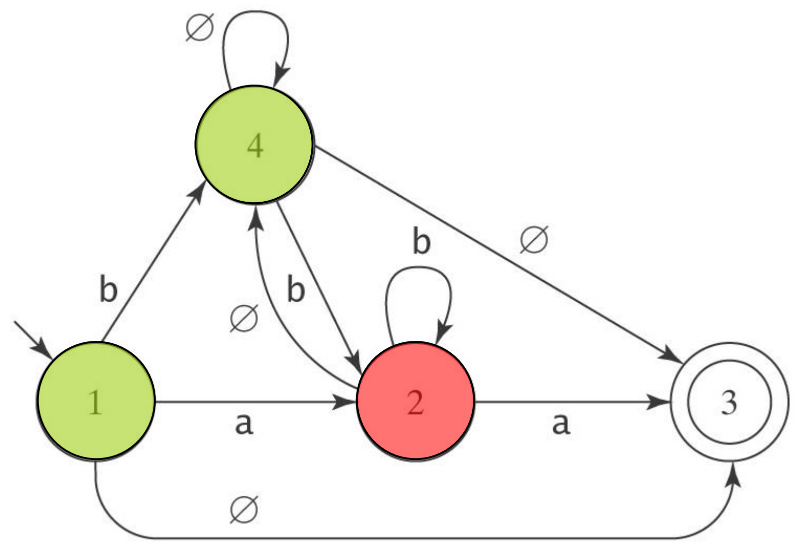


$R' = R(p, q) \cup R(p, rip)\ R(rip, rip)^*\ R(rip, q)$

Let *rip* be state 2. Then:

$$
\begin{aligned}
R'(1, 3) &= R(1, 3) \cup R(1, rip)R(rip, rip)^*R(rip, 3) \\
&= R(1, 3) \cup R(1, 2)R(2, 2)^*R(2, 3) \\
&= \quad \varnothing \quad \cup \quad \mathtt{a} \qquad \mathtt{b}^* \qquad \mathtt{a} \\
&= \mathtt{ab^*a}
\end{aligned}
$$

KLEENE'S THEOREM
The harder example

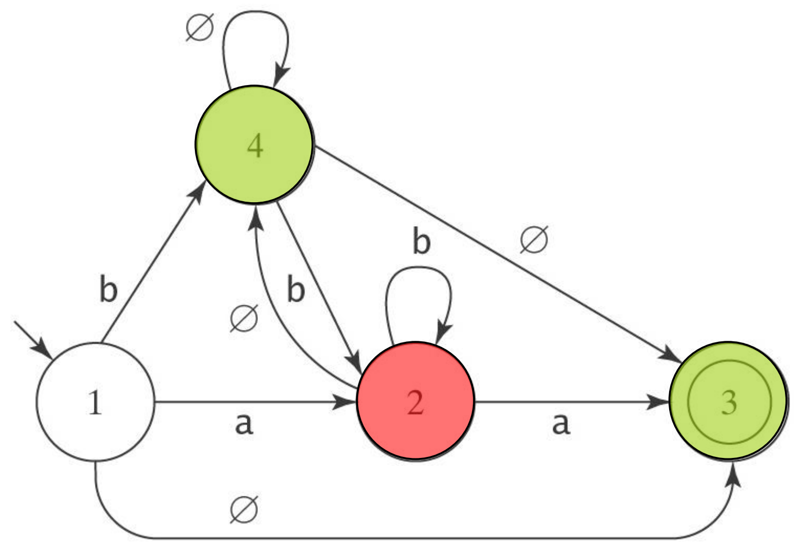


$$R' = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q)$$

Let *rip* be state 2. Then:

$$\begin{aligned} R'(1, 4) &= R(1, 4) \cup R(1, rip)R(rip, rip)^*R(rip, 4) \\ &= R(1, 4) \cup R(1, 2)R(2, 2)^*R(2, 4) \\ &= \mathtt{b} \cup \mathtt{a}\ \mathtt{b}^*\ \varnothing \\ &= \mathtt{b} \end{aligned}$$

KLEENE'S THEOREM
The harder example



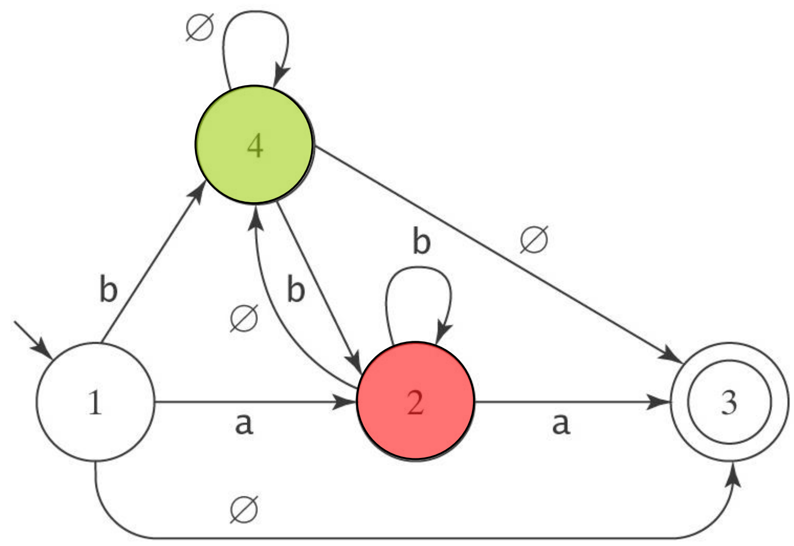
$R' = R(p, q) \cup R(p, rip)\ R(rip, rip)^*\ R(rip, q)$

Let *rip* be state 2. Then:

$$
\begin{aligned}
R'(4, 3) &= R(4, 3) \cup R(4, rip)R(rip, rip)^*R(rip, 3) \\
&= R(4, 3) \cup R(4, 2)R(2, 2)^*R(2, 3) \\
&= \quad \varnothing \quad \cup \quad \mathtt{b} \qquad \mathtt{b}^* \qquad \mathtt{a} \\
&= \mathtt{bb^*a}
\end{aligned}
$$

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

KLEENE'S THEOREM
The harder example



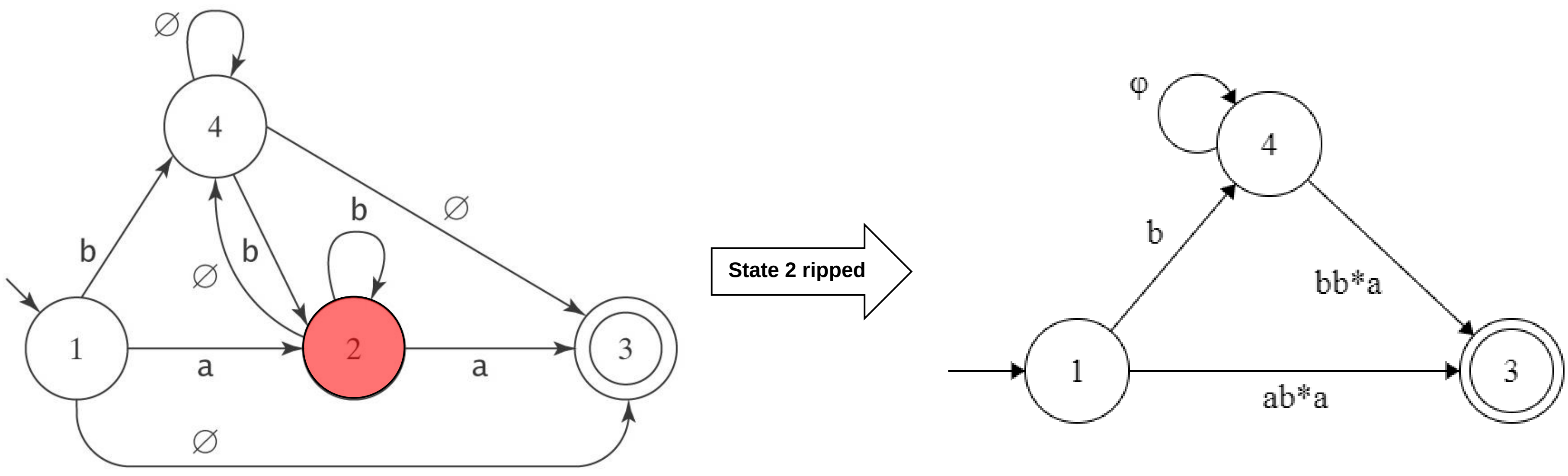
$R' = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q)$

Let *rip* be state 2. Then:

$$
\begin{aligned}
R'(4, 4) \quad &= R(4, 4) \cup R(4, rip)R(rip, rip)^*R(rip, 4) \\
&= R(4, 4) \cup R(4, 2)R(2, 2)^*R(2, 4) \\
&= \quad \emptyset \quad \cup \quad \mathtt{b} \quad \mathtt{b}^* \quad \emptyset \\
&= \quad \emptyset
\end{aligned}
$$

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

KLEENE'S THEOREM
The harder example



Kleene's Theorem, RE and FSM

- No difference in formal power of RE and FSM
- Practical difference in their effectiveness as problem solving tools
  - RE can specify the order in which a sequence of symbols must appear
    - e.g. phone number / email address
  - When order doesn't matter FSM is more effective
    - e.g. vending machine / parity checking

SIMPLIFYING REGULAR EXPRESSIONS

Regex's describe sets:

- Union is commutative: $\alpha \cup \beta = \beta \cup \alpha$.
- Union is associative: $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$.
- $\emptyset$ is the identity for union: $\alpha \cup \emptyset = \emptyset \cup \alpha = \alpha$.
- Union is idempotent: $\alpha \cup \alpha = \alpha$.

Concatenation:

- Concatenation is associative: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.
- ε is the identity for concatenation: $\alpha \varepsilon = \varepsilon \alpha = \alpha$.
- $\emptyset$ is a zero for concatenation: $\alpha \emptyset = \emptyset \alpha = \emptyset$.

Concatenation distributes over union:

- $(\alpha \cup \beta) \gamma = (\alpha \gamma) \cup (\beta \gamma)$.
- $\gamma (\alpha \cup \beta) = (\gamma \alpha) \cup (\gamma \beta)$.

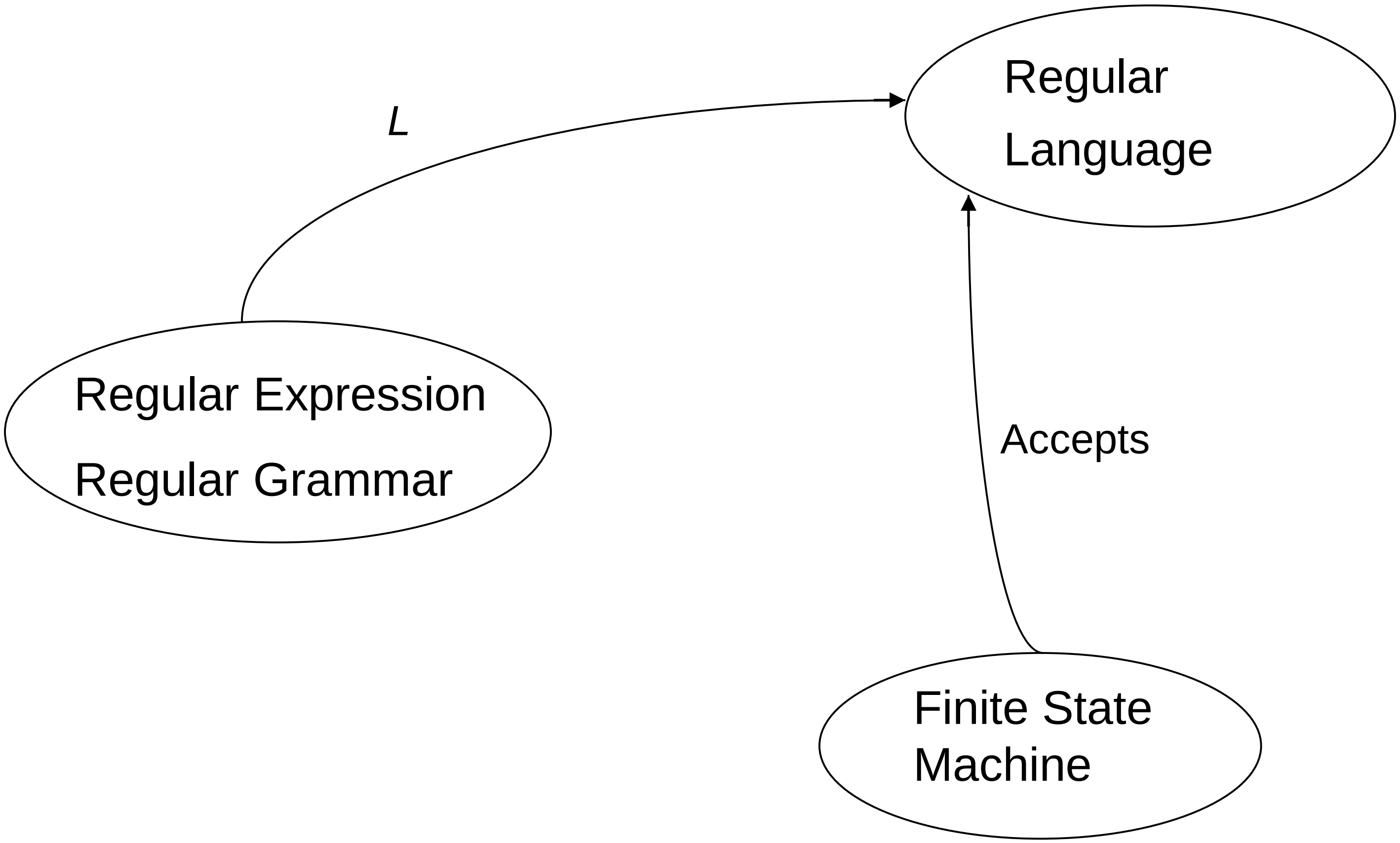
Kleene star:

- $\emptyset^* = \varepsilon$.
- $\varepsilon^* = \varepsilon$.
- $(\alpha^*)^* = \alpha^*$.
- $\alpha^*\alpha^* = \alpha^*$.
- $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$.

SUMMARY

- Regular Expressions (REs) are useful to define patterns
- For every RE there is an equivalent FSM
- For every FSM there is a equivalent RE
- The class of languages that can be defined with REs is exactly the class of regular languages
- REs can be simplified and in the simplified form they represent the language

REGULAR LANGUAGES



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

REGULAR GRAMMARS

A ***regular grammar*** G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains **nonterminals** and **terminals**,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite set of rules of the form:

$$X \rightarrow Y,$$

- S (the start symbol) is a nonterminal.

REGULAR GRAMMARS

In a regular grammar, all rules in R must:

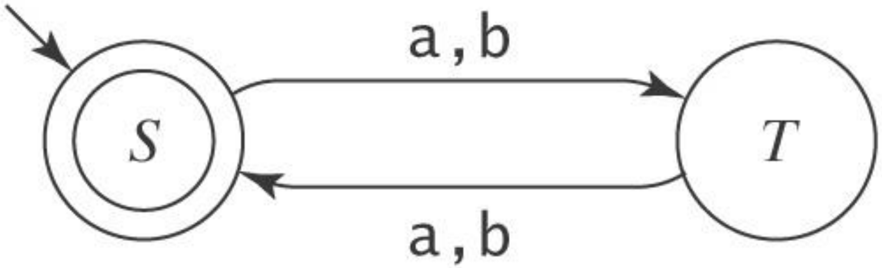
- have a left hand side that is a single nonterminal
- have a right hand side that is:
  - ε, or
  - a single terminal, or
  - a single terminal followed by a single nonterminal.

Legal: $S \rightarrow \mathtt{a}$, $S \rightarrow \varepsilon$, and $T \rightarrow \mathtt{a}S$

Not legal: $S \rightarrow \mathtt{a}S\mathtt{a}$ and $\mathtt{a}S\mathtt{a} \rightarrow T$

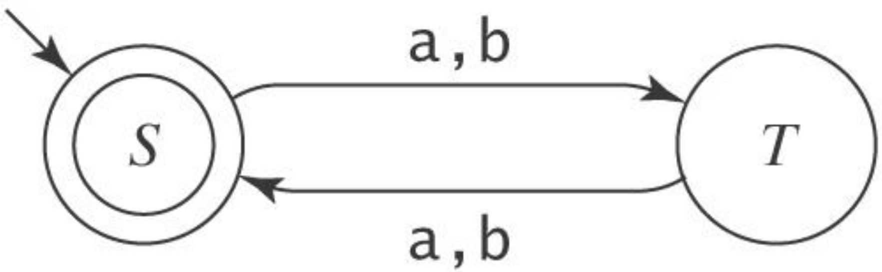
REGULAR GRAMMARS
Example

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : |w| \text{ is even}\}$ $((\mathtt{aa}) \cup (\mathtt{ab}) \cup (\mathtt{ba}) \cup (\mathtt{bb}))^*$



REGULAR GRAMMARS
Example

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : |w| \text{ is even}\}$ $((\mathtt{aa}) \cup (\mathtt{ab}) \cup (\mathtt{ba}) \cup (\mathtt{bb}))^*$



$S \to \varepsilon$
$S \to \mathtt{a}T$
$S \to \mathtt{b}T$
$T \to \mathtt{a}S$
$T \to \mathtt{b}S$

REGULAR GRAMMARS

Theorem: The class of languages that can be defined with regular grammars is exactly the regular languages.

Proof: By two constructions.

REGULAR GRAMMARS

Regular grammar → FSM:

grammartofsm(G = (V, Σ, R, S)) =

1. Create in M a separate state for each nonterminal in V.
2. Start state is the state corresponding to S .
3. If there are any rules in R of the form $X \rightarrow a$, for some $a \in \Sigma$, create a new state labeled #.
4. For each rule of the form $X \rightarrow a\ Y$, add a transition from X to Y labeled a.
5. For each rule of the form $X \rightarrow a$, add a transition from X to # labeled a.
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X as accepting.
7. Mark state # as accepting.
8. Complete M if incomplete (see the textbook for details).

FSM → Regular grammar: Similarly.

REGULAR GRAMMARS
Example - Even Length Strings

$$S \to \varepsilon$$
$$S \to \mathtt{a}T$$
$$S \to \mathtt{b}T$$

$$T \to \mathtt{a}$$
$$T \to \mathtt{b}$$
$$T \to \mathtt{a}S$$
$$T \to \mathtt{b}S$$

REGULAR GRAMMARS
Example – Strings that End in `aaaa`

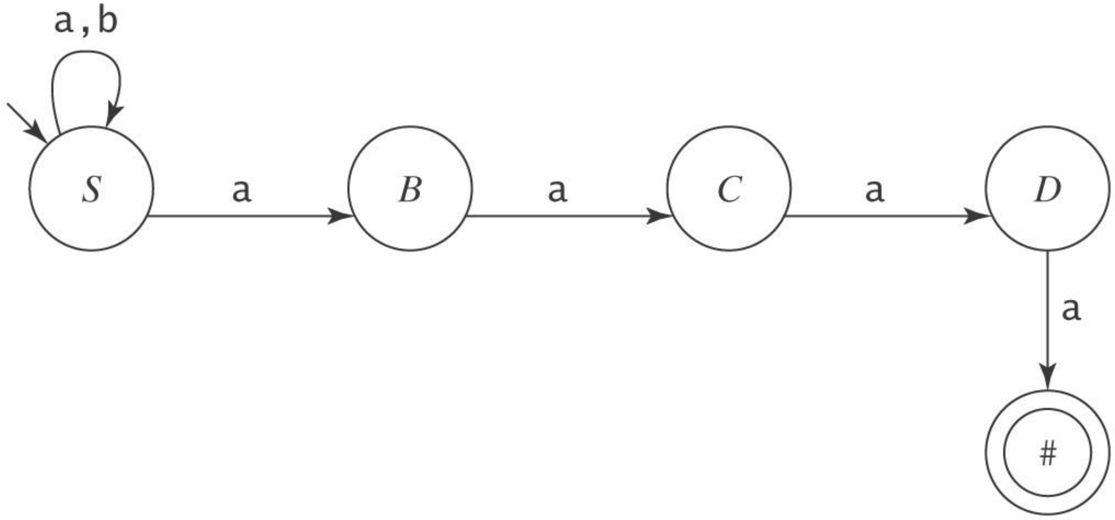
$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ ends with the pattern $\mathtt{aaaa}\}$.

$$S \rightarrow \mathtt{a}S$$
$$S \rightarrow \mathtt{b}S$$
$$S \rightarrow \mathtt{a}B$$
$$B \rightarrow \mathtt{a}C$$
$$C \rightarrow \mathtt{a}D$$
$$D \rightarrow \mathtt{a}$$

REGULAR GRAMMARS
Example – Strings that End in aaaa

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ ends with the pattern $\mathtt{aaaa}\}$.

$S \rightarrow \mathtt{a}S$
$S \rightarrow \mathtt{b}S$
$S \rightarrow \mathtt{a}B$
$B \rightarrow \mathtt{a}C$
$C \rightarrow \mathtt{a}D$
$D \rightarrow \mathtt{a}$



REGULAR GRAMMARS
Example – One character missing

$S \to \varepsilon$
$S \to \mathtt{a}B$
$S \to \mathtt{a}C$
$S \to \mathtt{b}A$
$S \to \mathtt{b}C$
$S \to \mathtt{c}A$
$S \to \mathtt{c}B$

$A \to \mathtt{b}A$
$A \to \mathtt{c}A$
$A \to \varepsilon$
$B \to \mathtt{a}B$
$B \to \mathtt{c}B$
$B \to \varepsilon$

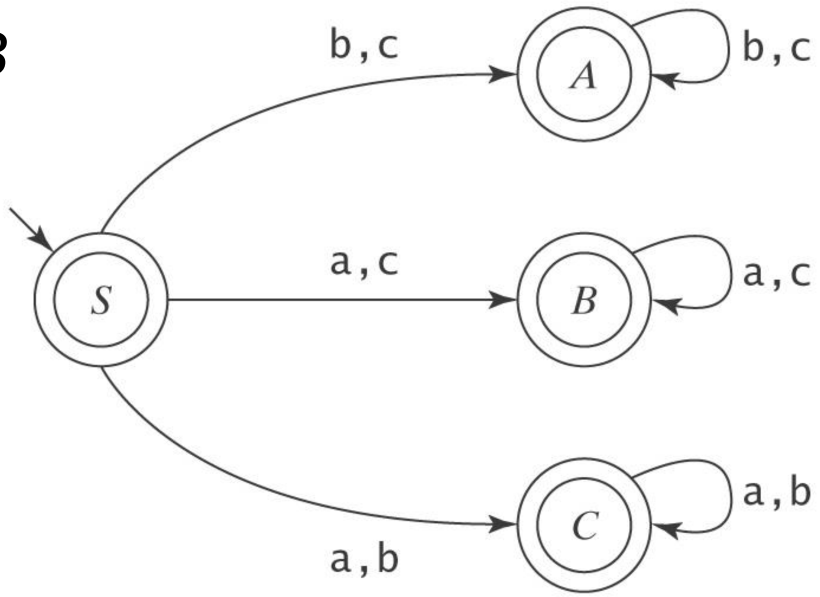
$C \to \mathtt{a}C$
$C \to \mathtt{b}C$
$C \to \varepsilon$

REGULAR GRAMMARS
Example – One character missing

$S \to \varepsilon$
$S \to \mathtt{a}B$
$S \to \mathtt{a}C$
$S \to \mathtt{b}A$
$S \to \mathtt{b}C$
$S \to \mathtt{c}A$
$S \to \mathtt{c}B$

$A \to \mathtt{b}A$
$A \to \mathtt{c}A$
$A \to \varepsilon$
$B \to \mathtt{a}B$
$B \to \mathtt{c}B$
$B \to \varepsilon$

$C \to \mathtt{a}C$
$C \to \mathtt{b}C$
$C \to \varepsilon$



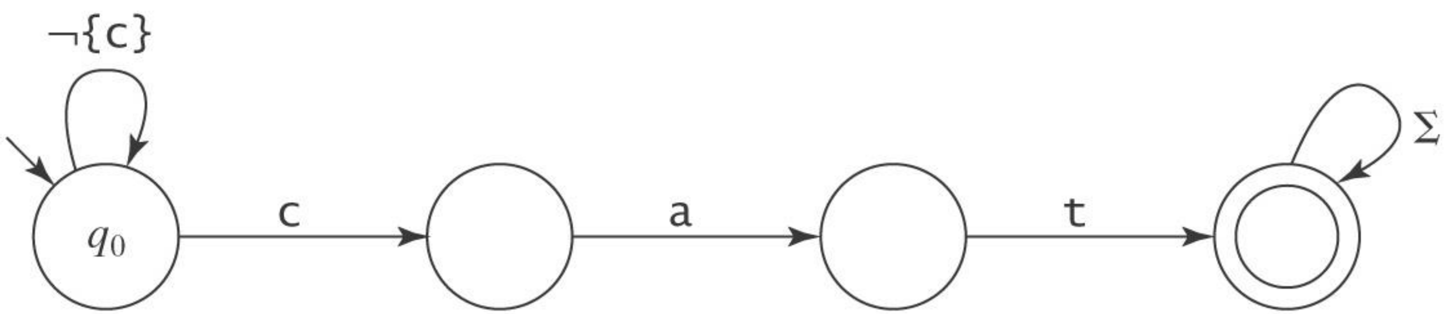
REGULAR GRAMMARS, REs and FSM

- Regular grammars define regular languages
- So equivalent to REs and FSM
- Regular grammars are used less frequently
  - FSM closely mirrors the structure of a regular language
  - Regular expressions are another useful representation

Building DFSM is sometimes easy

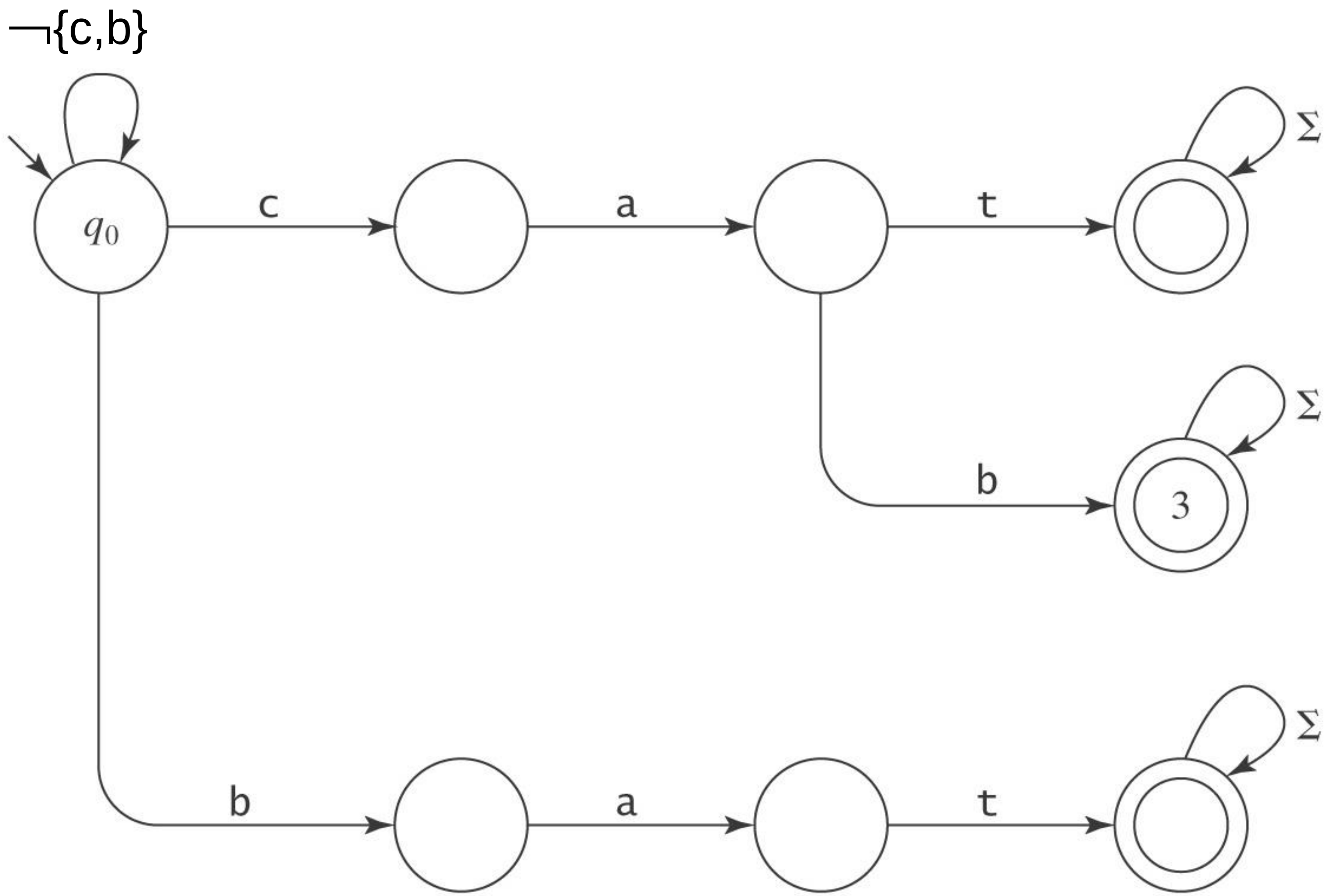
{cat, bat, cab}

The single keyword `cat`:



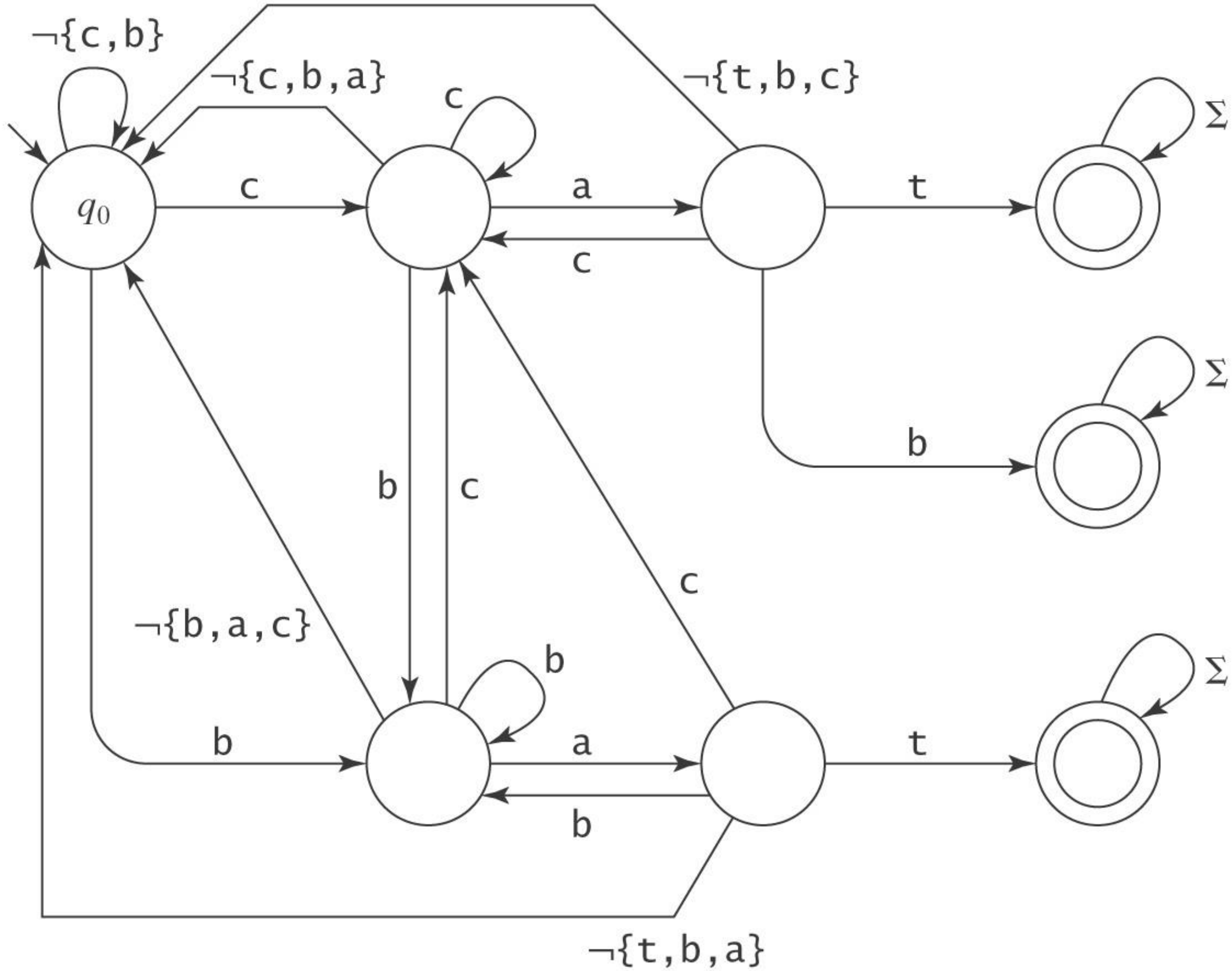
Building DFSM is sometimes easy {cat, bat, cab}

Adding `bat` and `cab`:



Building DFSM is sometimes easy
{cat, bat, cab}

All transitions in:



Building DFSM is sometimes easy

buildkeywordFSM(*K*:set of keywords) =

1. Create a start state q_0.
2. For each element *k* of *K* do:

    Create a branch corresponding to *k*.

3. Create a set of transitions that describe what to do when a branch dies:
    I. Because the complete pattern has been found OR
    II. Because the next char is not correct to continue
4. Make the states at the end of each branch accepting

A Biology Example – BLAST

Given a protein or DNA sequence, find others that are likely to be evolutionarily close to it.

<u>ESGH</u>D<u>TTTYY</u>NKNRY<u>PAGWNN</u>HHDQM<u>FFW</u>V

Build a DFSM that can examine thousands of other sequences and find those that match any of the selected patterns.

Ref: Section K.3.1 (Page 971)

Regular Expressions in Perl

Syntax	*Name*	*Description*
abc	Concatenation	Matches *a*, then *b*, then *c*, where *a*, *b*, and *c* are any regexs
a | *b* | *c*	Union (Or)	Matches *a* or *b* or *c*, where *a*, *b*, and *c* are any regexs
*a**	Kleene star	Matches 0 or more *a*'s, where *a* is any regex
a+	At least one	Matches 1 or more *a*'s, where *a* is any regex
a?		Matches 0 or 1 *a*'s, where *a* is any regex
a{*n*, *m*}	Replication	Matches at least *n* but no more than *m* *a*'s, where *a* is any regex
*a**?	Parsimonious	Turns off greedy matching so the shortest match is selected
a+?	″	″
.	Wild card	Matches any character except newline
^	Left anchor	Anchors the match to the beginning of a line or string
$	Right anchor	Anchors the match to the end of a line or string
`[a-z]`		Assuming a collating sequence, matches any single character in range
`[^a-z]`		Assuming a collating sequence, matches any single character not in range
`\d`	Digit	Matches any single digit, i.e., string in `[0-9]`
`\D`	Nondigit	Matches any single nondigit character, i.e., `[^0-9]`
`\w`	Alphanumeric	Matches any single "word" character, i.e., `[a-zA-Z0-9]`
`\W`	Nonalphanumeric	Matches any character in `[^a-zA-Z0-9]`
`\s`	White space	Matches any character in [space, tab, newline, etc.]

Regular Expressions in Perl

Syntax	*Name*	*Description*
`\S`	Nonwhite space	Matches any character not matched by `\s`
`\n`	Newline	Matches newline
`\r`	Return	Matches return
`\t`	Tab	Matches tab
`\f`	Formfeed	Matches formfeed
`\b`	Backspace	Matches backspace inside []
`\b`	Word boundary	Matches a word boundary outside []
`\B`	Nonword boundary	Matches a non-word boundary
`\0`	Null	Matches a null character
nnn	Octal	Matches an ASCII character with octal value *nnn*
`\x`*nn*	Hexadecimal	Matches an ASCII character with hexadecimal value *nn*
`\c`*X*	Control	Matches an ASCII control character
char	Quote	Matches *char*; used to quote symbols such as . and \
(*a*)	Store	Matches *a*, where *a* is any regex, and stores the matched string in the next variable
\1	Variable	Matches whatever the first parenthesized expression matched
\2		Matches whatever the second parenthesized expression matched
…		For all remaining variables

References

- **Automata, Computability and Complexity. Theory and Applications**

  - By Elaine Rich
- Chapter 6, 7:
  - Page : 127-151, 155-161.