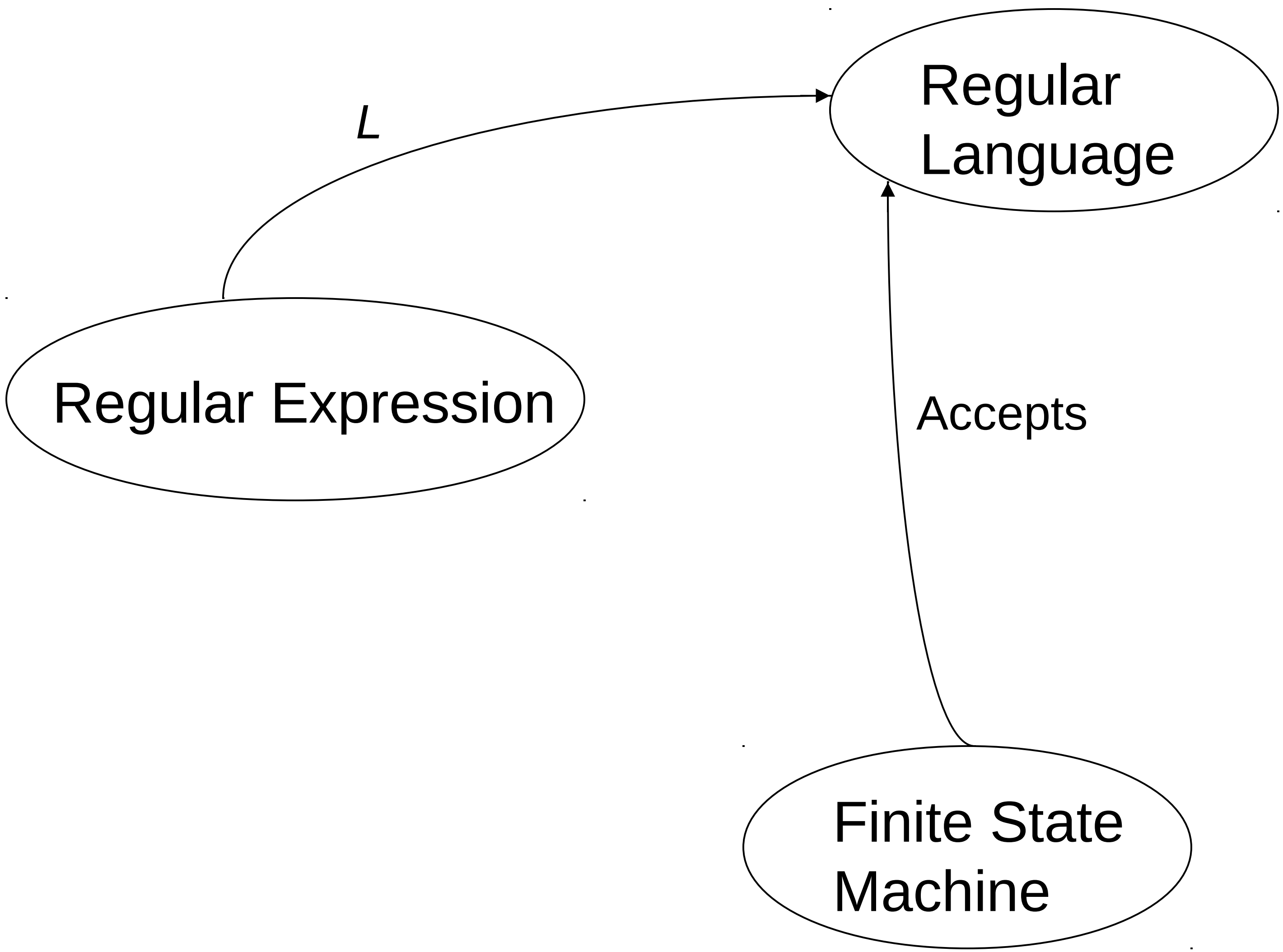
Regular Expressions

Chapter 6

Regular Languages



Regular Expressions

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

1. $\emptyset$ is a regular expression.
2. ε is a regular expression.
3. Every element $a \in \Sigma$ 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).

Sometimes union '$\cup$' is also denoted as '+' or '|'.

Regular Expression Examples

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

$\varnothing$

ε

a

$(a \cup b)^*$

$abba \cup \varepsilon$

Regular Expressions Define Languages

Semantic interpretation: the **language $L(\alpha)$** expressed by a regular expression α:

1. $L(\varnothing) = \varnothing$.
2. $L(\varepsilon) = \{\varepsilon\}$.
3. $L(c) = \{c\}$, where $c \in \Sigma$.
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. $L((\alpha)) = L(\alpha)$.

Example

$$L((a \cup b)^*b) = L((a \cup b)^*) \; L(b)$$

$$= (L((a \cup b)))^* \; L(b)$$

$$= (L(a) \cup L(b))^* \; L(b)$$

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

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

Examples

$L(\ a^*b^*\ ) =$

$L(\ (a \cup b)^*\ ) =$

$L(\ (a \cup b)^*a^*b^*\ ) =$

$L(\ (a \cup b)^*abba(a \cup b)^*\ ) =$

Going the Other Way

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

Going the Other Way

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

$((a \cup b)(a \cup b))^*$

$(aa \cup ab \cup ba \cup bb)^*$

Going the Other Way

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

$((a \cup b)(a \cup b))^*$

$(aa \cup ab \cup ba \cup bb)^*$

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

Going the Other Way

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

$((a \cup b) (a \cup b))^*$

$(aa \cup ab \cup ba \cup bb)^*$

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

$b^* (ab^*ab^*)^* a b^*$

$b^* a b^* (ab^*ab^*)^*$

More Regular Expression Examples

L($(aa^*) \cup \varepsilon$) =

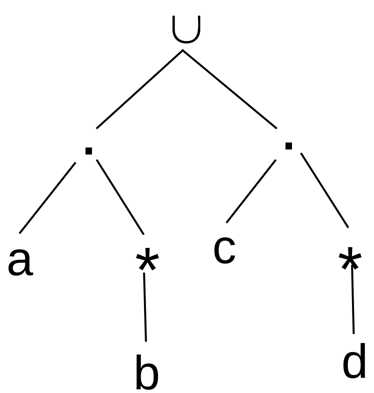
L($(a \cup \varepsilon)^*$) =

$L = \{w \in \{a, b\}^*$: there is no more than one b in $w\}$

$L = \{w \in \{a, b\}^*$: no two consecutive letters in w are the same$\}$

Operator Precedence in Regular Expressions

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



$ab^* \cup cd^*$

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

The Details Matter

$a^* \cup b^* \neq (a \cup b)^*$

$(ab)^* \neq a^*b^*$

The Details Matter

$L_1 = \{w \in \{a, b\}^* :$ every a is immediately followed a b$\}$

A regular expression for L_1:

A FSM for L_1:

$L_2 = \{w \in \{a, b\}^* :$ every a has a matching b somewhere$\}$

A regular expression for L_2:

A FSM for L_2:

Kleene’s Theorem

Finite state machines and regular expressions define the same class of languages.

Theorem 6.3 (Kleene): The class of languages that can be defined with regular expressions is exactly the class of regular languages.

For Every Regular Expression There is a Corresponding FSM

Proof: By construction (Thompson's construction: simpler than the one in textbook).

We shall build a NDFSM for each regular expression such that:
- its starting state has no incoming edges
- it has only one accepting state that has no outgoing edges

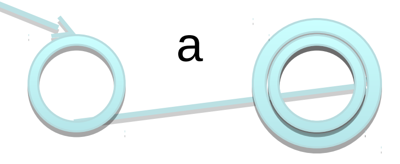
We use structural induction:

For Every Regular Expression There is a Corresponding FSM

NDFSM's for the basic blocks:

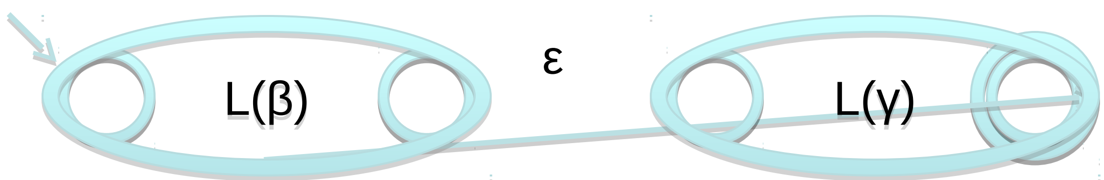
∅:

A single element $a \in \Sigma$:



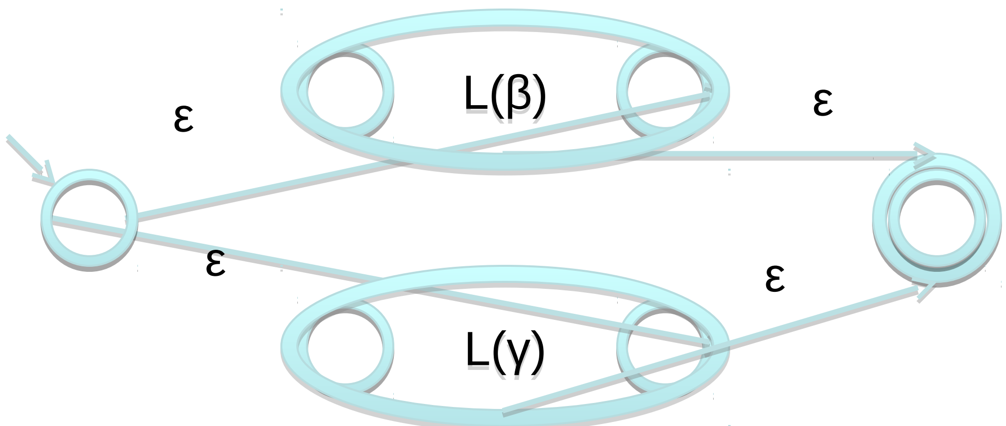
Concatenation

$\alpha = \beta\ \gamma$: build a NDFSM for L(α)



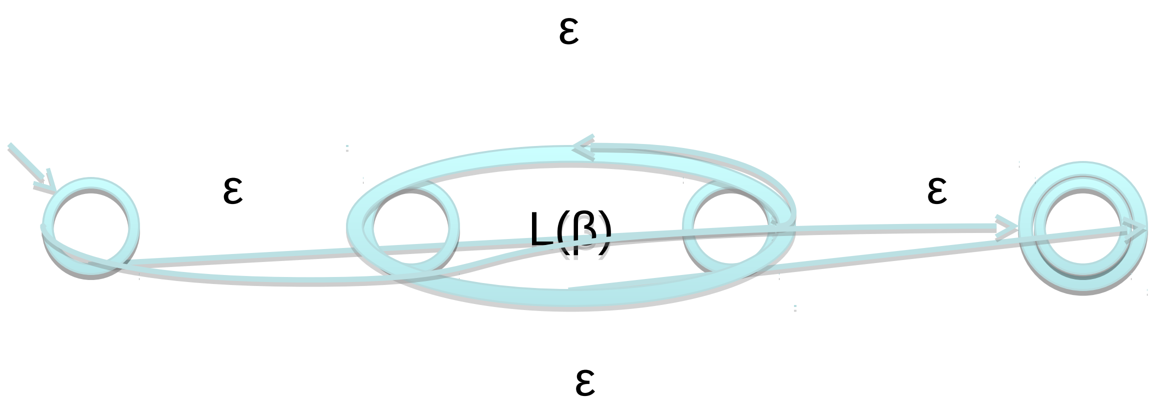
Union

$\alpha = \beta \cup \gamma$: build a NDFSM for $L(\alpha)$



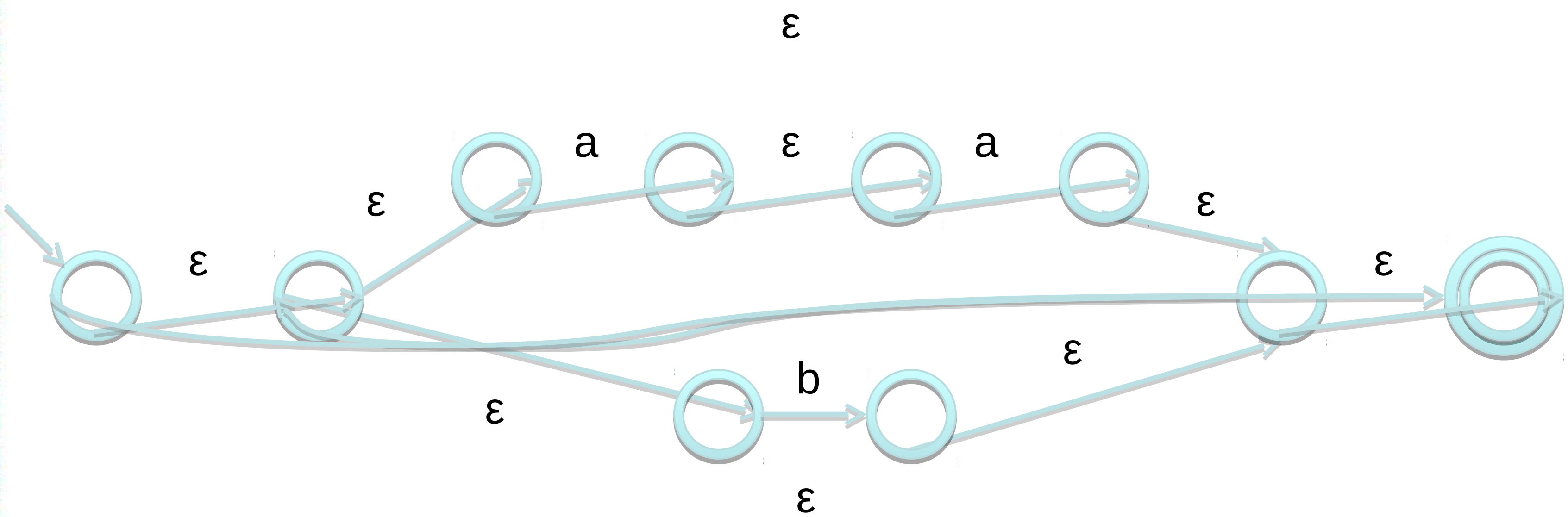
Kleene *

$\alpha = \beta^*$: build a NDFSM for $L(\alpha)$



Example

$\alpha = (aa \cup b)^*$



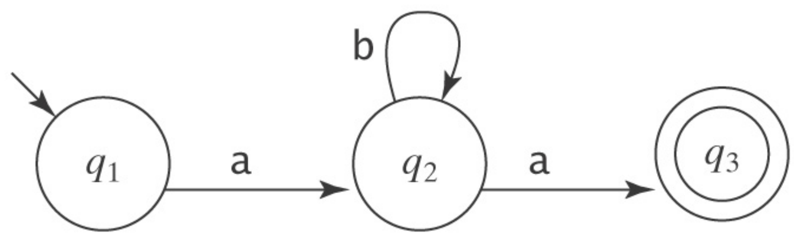
For Every FSM There is a Corresponding Regular Expression

We'll show this by construction.

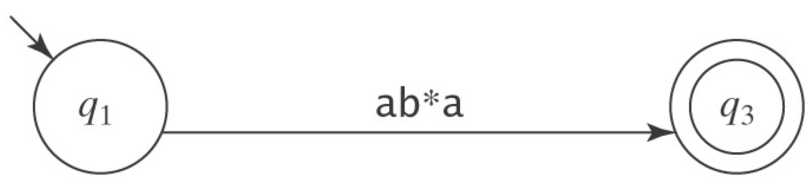
The key idea is that we'll allow arbitrary regular expressions to label the transitions of an FSM.

A Simple Example

Let *M* be:



Suppose we rip out state 2:

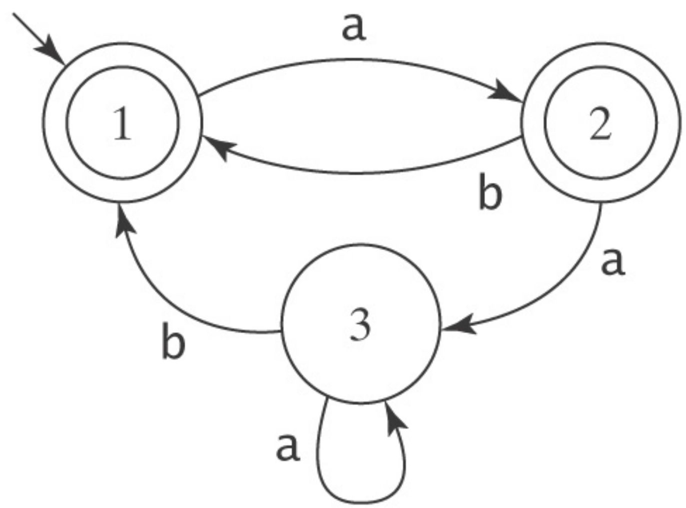


The Algorithm *fsmtoregexheuristic*

fsmtoregexheuristic(*M*: FSM) =

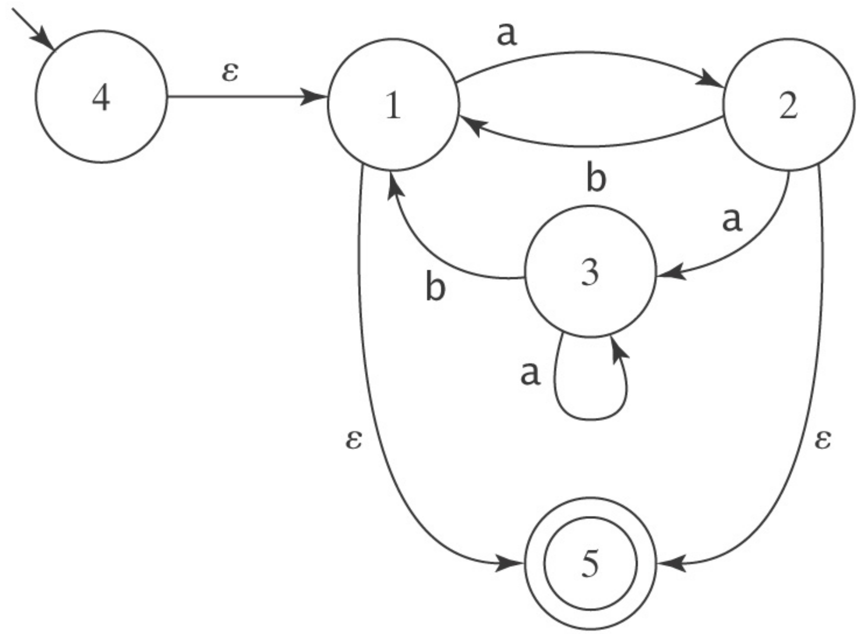
1. Remove unreachable states from *M*.
2. If *M* has no accepting states then return $\varnothing$.
3. If the start state of *M* is part of a loop, create a new start state *s* and connect *s* to *M*'s start state via an ε-transition.
4. If there is more than one accepting state of *M* or there are any 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.

An Example



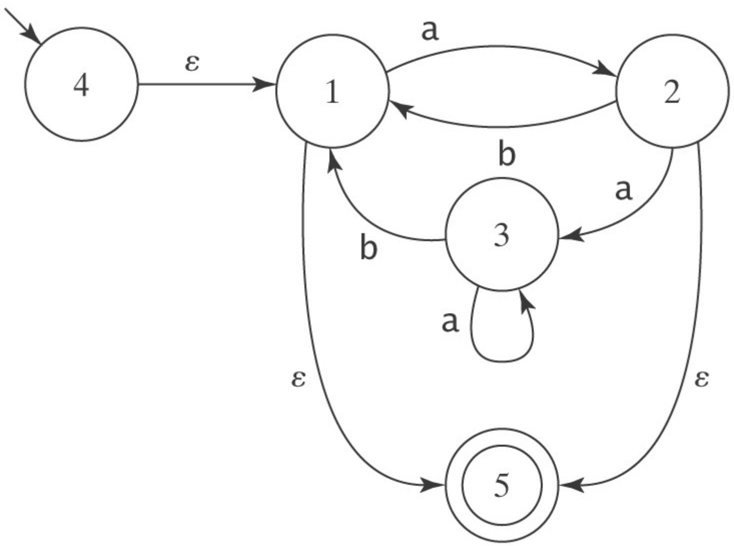
1. Create a new initial state and a new, unique accepting state, neither of which is part of a loop.

An Example, Continued

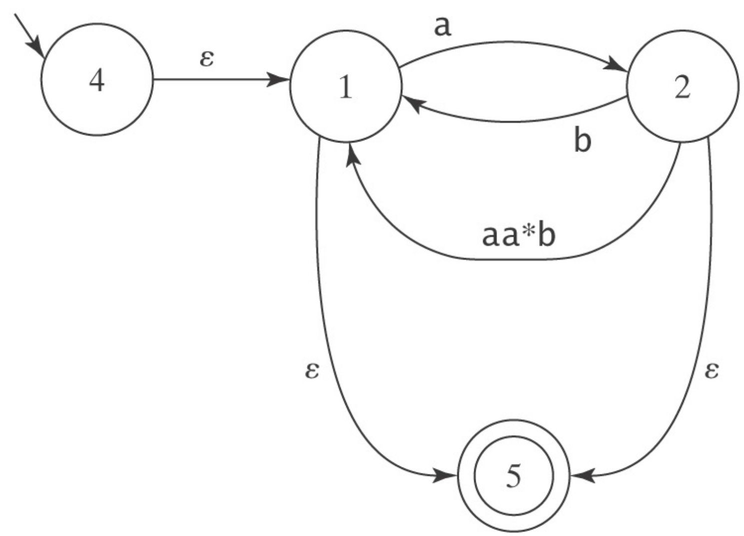


2. Remove states and arcs and replace with arcs labelled with larger and larger regular expressions.

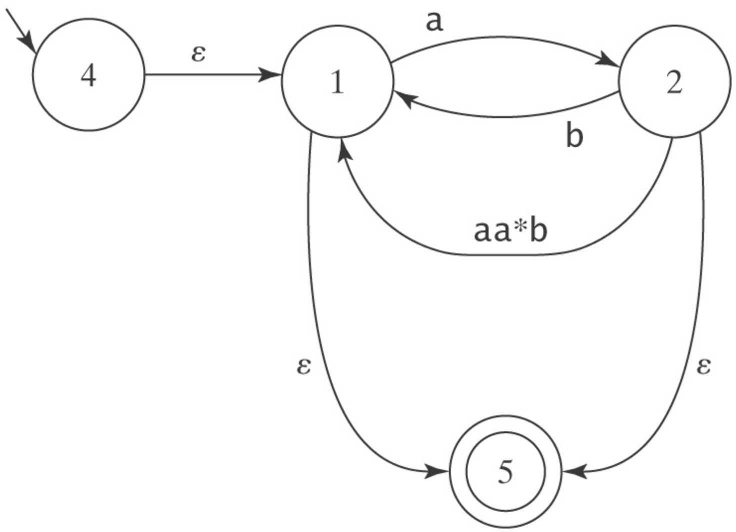
An Example, Continued



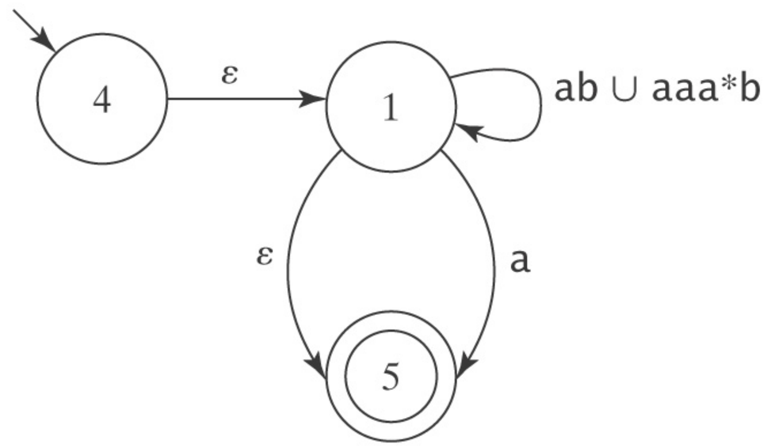
Remove state 3:



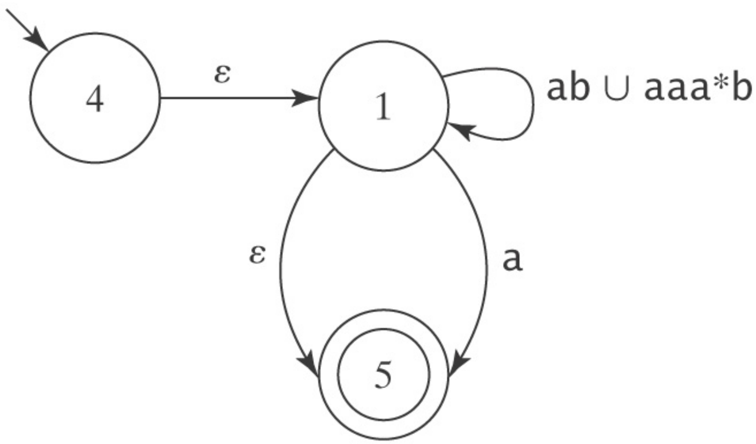
An Example, Continued



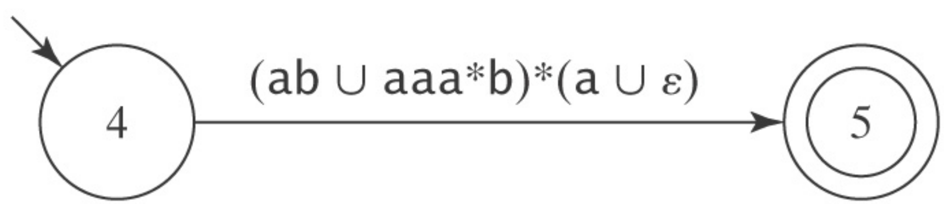
Remove state 2:



An Example, Continued



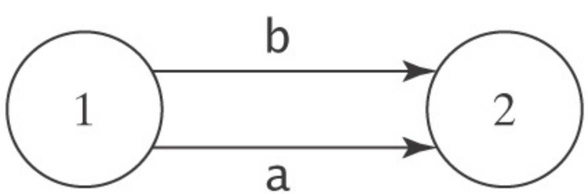
Remove state 1:



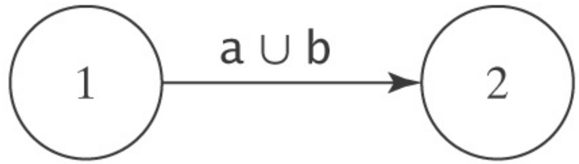
Further Modifications to *M* Before We Start

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:

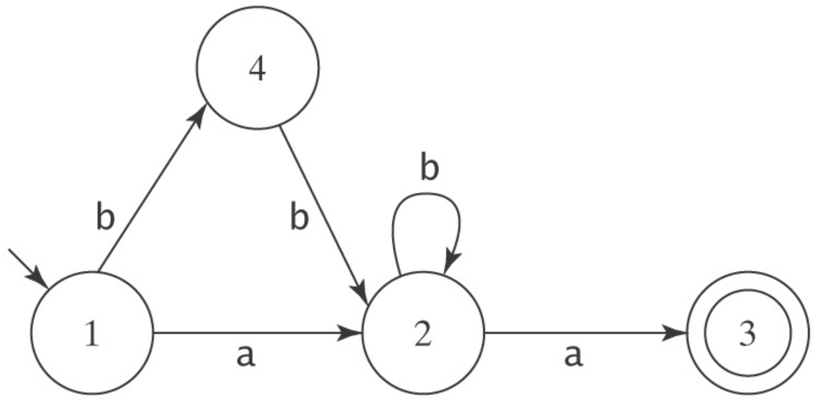


becomes:

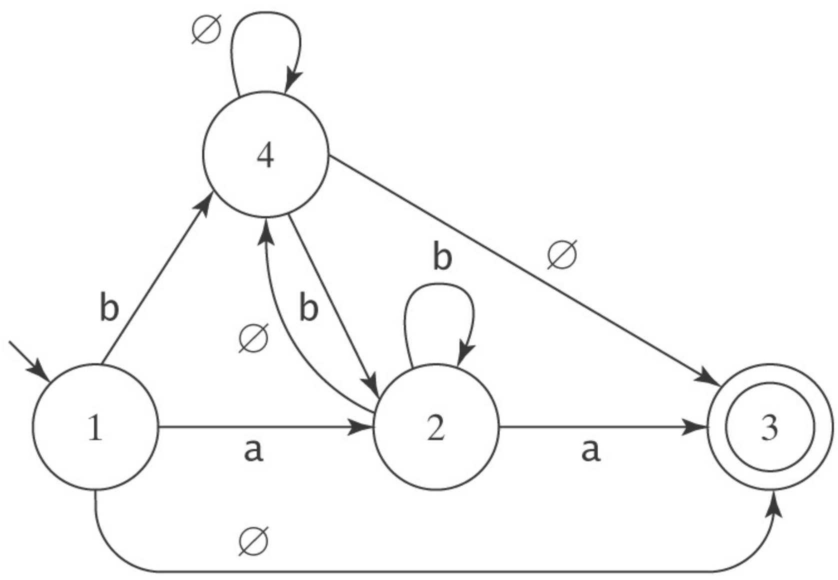


Further Modifications to *M* Before We Start

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

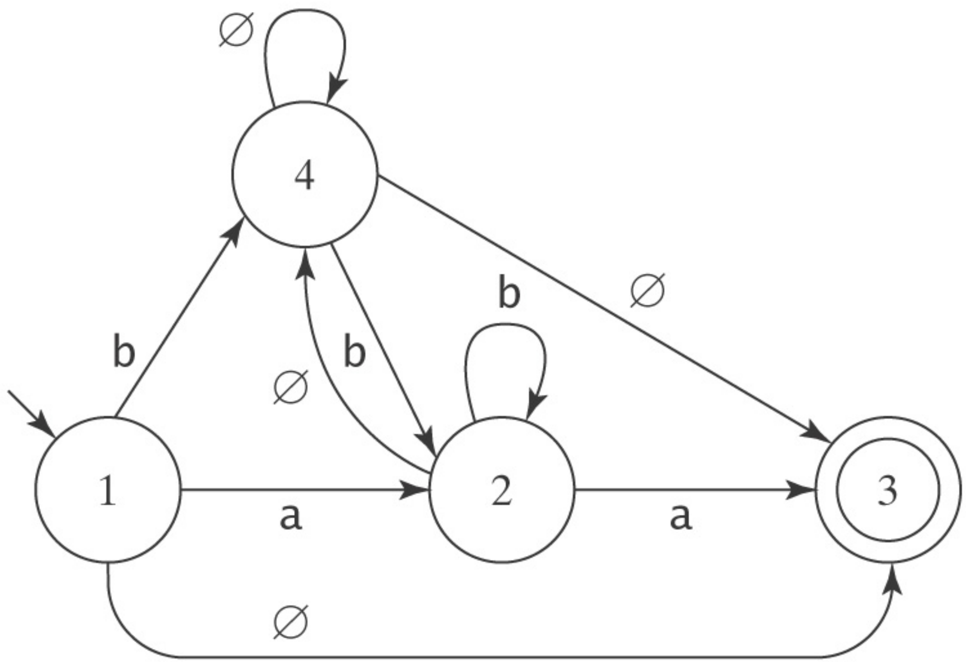


becomes:



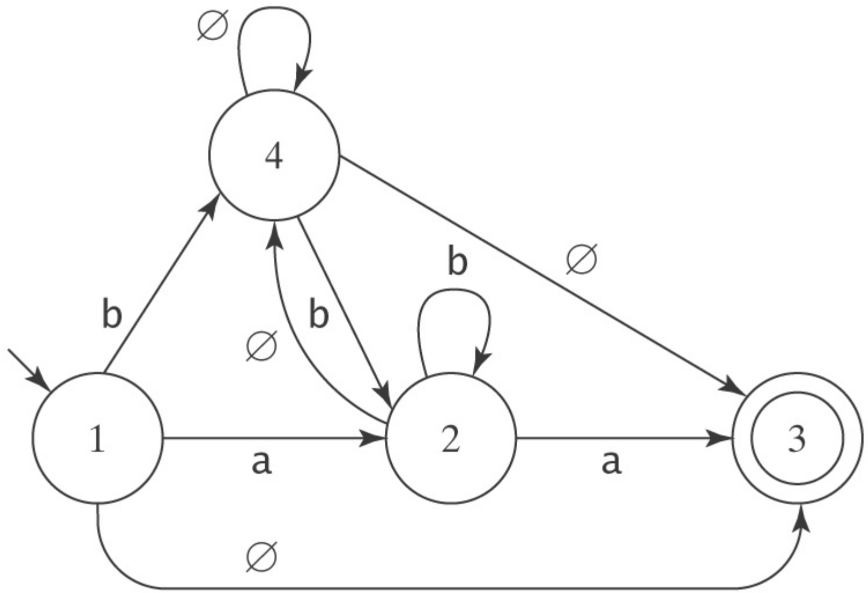
Ripping Out States

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



Suppose we rip state 2.

What Happens When We Rip?



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*.

Defining *R*(*p*, *q*)

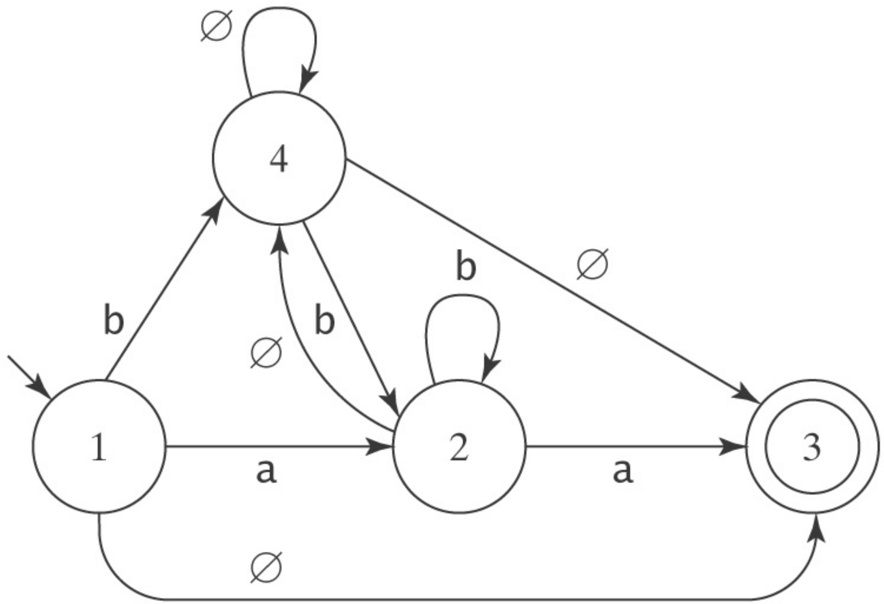
After removing *rip*, the new regular expression that should label the transition from *p* to *q* is:

$R(p, q)$		/* 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:

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

Returning to Our Example



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

Let *rip* = 2. Then:

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

The Algorithm *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 $\varnothing$.

The Algorithm *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.

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

Using Regular Expressions in the Real World

Matching numbers:

-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

Matching ip addresses:

([0-9]{1,3} (\. [0-9] {1,3}){3})

Finding doubled words:

([A-Za-z]+) \s+ \1

From Friedl, J., Mastering Regular Expressions, O’Reilly,1997.

Simplifying Regular Expressions

Regex's describe sets:

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

Concatenation:

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

Concatenation distributes over union:

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

Kleene star:

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

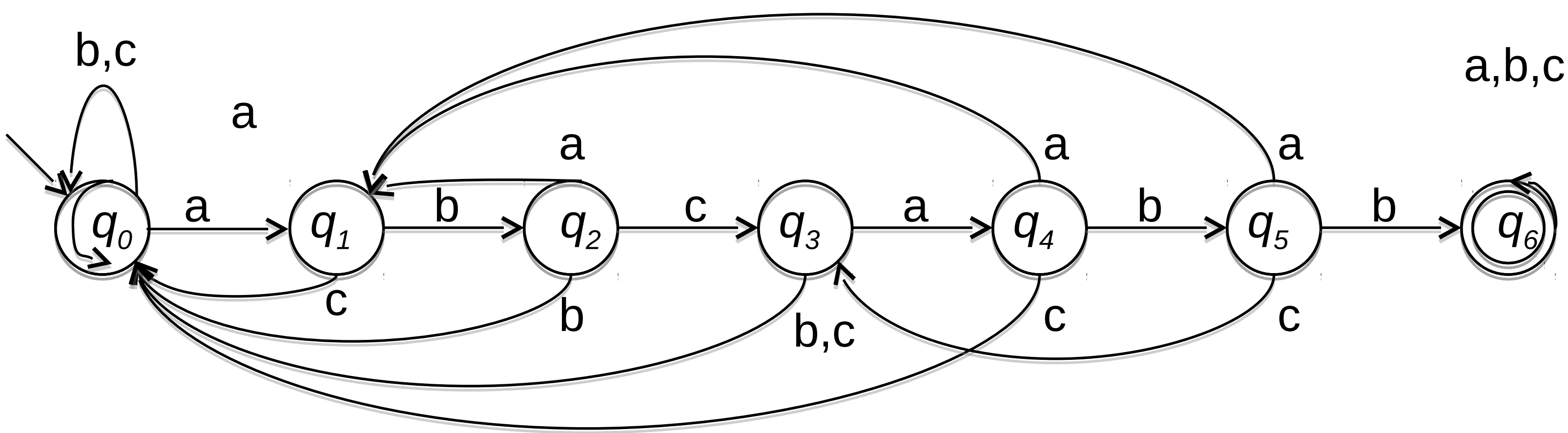
Pattern Matching

Given a pattern *p* and a text *T*, does *p* occur as a substring of *T*?

Solution: Construct a DFSM *M* for L($\Sigma^* p\ \Sigma^*$); *p* occurs in *T* iff *M* accepts *T.*

Example: p = abcabb

Below is the minimal DFSM that accepts L(Σ^* abcabb Σ^*)



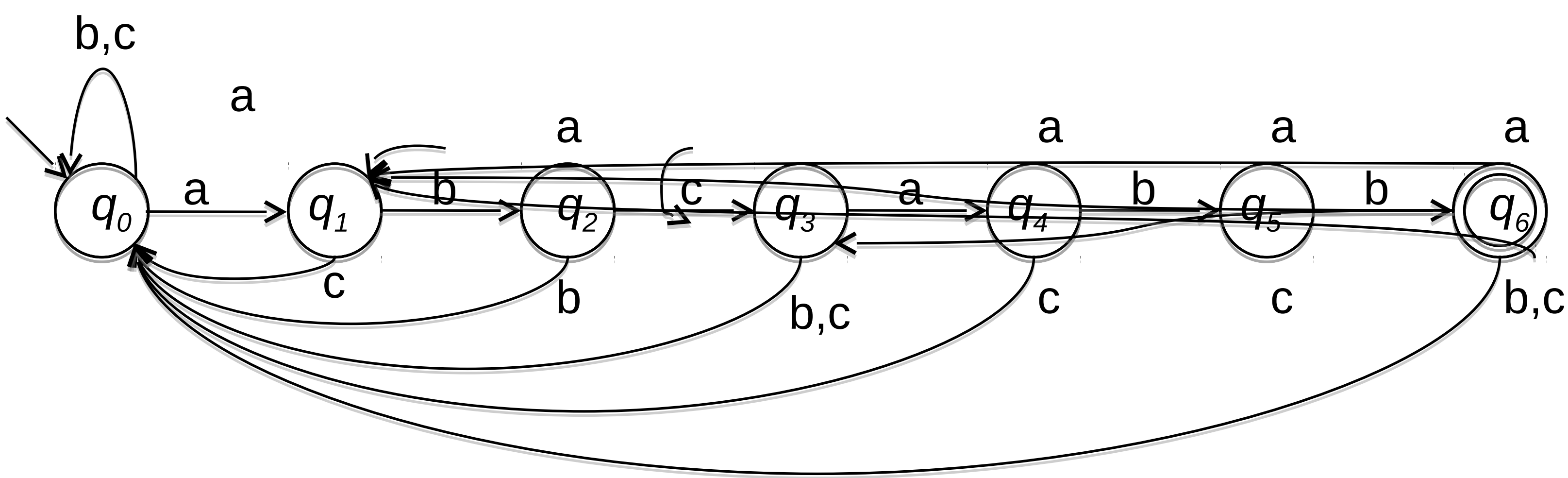
Pattern Searching

Given a pattern *p* and a text *T*, find all occurrences of *p* as a substring of *T*.

Solution: Construct a DFSM *M* for L(Σ^**p)* and run *M* on *T*; an accepting state marks the end of an occurrence of *p*.

Example: p = abcabb

Below is the minimal DFSM that accepts L(Σ^*abcabb)



Multiple Patterns

Multi-Pattern matching

Given several patterns $p_1, p_2, \ldots, p_k$, and a text T, does any of the patterns occur as a substring of T?

Solution: DFSM for $L(\Sigma^*(p_1 \cup p_2 \cup \ldots \cup p_k)\ \Sigma^*)$.

Multi-Pattern searching

Given several patterns $p_1, p_2, \ldots, p_k$, and a text T, find all occurrences of all patterns as substrings of T.

Solution: DFSM for $L(\Sigma^*(p_1 \cup p_2 \cup \ldots \cup p_k))$.