Regular Expressions

Regular Expression (RE)

Regular expression: An algebraic way to describe regular languages.

Many of today's programming languages use regular expressions to match patterns in strings.
E.g., awk, flex, lex, java, javascript, perl, python

Used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect.

Few Web search engines may allow the use of Regular Expressions

Recursive Definition

Primitive regular expressions: $\emptyset, \quad \lambda, \quad \alpha$

Given regular expressions r_1 and r_2

$$\left.\begin{array}{l} r_1 + r_2 \\ r_1 \cdot r_2 \\ r_1 * \\ (r_1) \end{array}\right\}$$ Are regular expressions

Examples

A regular expression: $(a + b \cdot c)^* \cdot (c + \emptyset)$

Not a regular expression: $(a + b +)$

Regular Expressions

A regular expression: $(a+b\cdot c)^*\cdot(c+\varnothing)$

Operator Precedence:

Highest: Kleene Closure

Then: Concatenation

Lowest: Union

Definition

For primitive regular expressions:

$$L(\varnothing) = \varnothing$$

$$L(\lambda) = \{\lambda\}$$

$$L(a) = \{a\}$$

Definition (continued)

For regular expressions r_1 and r_2

$$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

$$L(r_1 \cdot r_2) = L(r_1)\, L(r_2)$$

$$L(r_1 *) = (L(r_1))*$$

$$L((r_1)) = L(r_1)$$

Languages of Regular Expressions

$L(r)$: language of regular expression r

Example

$$L((a+b\cdot c)^*)=\{\lambda, a, bc, aa, abc, bca, \ldots\}$$

Example

Regular expression: $(a+b)\cdot a^*$

$$
\begin{aligned}
L((a+b)\cdot a^*) &= L((a+b))\,L(a^*) \\
&= L(a+b)\,L(a^*) \\
&= (L(a)\cup L(b))\,(L(a))^* \\
&= (\{a\}\cup\{b\})\,(\{a\})^* \\
&= \{a,b\}\,\{\lambda,a,aa,aaa,\ldots\} \\
&= \{a,aa,aaa,\ldots,b,ba,baa,\ldots\}
\end{aligned}
$$

Example

Regular expression $r = (a+b)^*(a+bb)$

$$L(r) = \{a, bb, aa, abb, ba, bbb, \ldots\}$$

Example

Regular expression $r = (aa)^*(bb)^*b$

$$L(r) = \{a^{2n}b^{2m}b : \quad n, m \geq 0\}$$

Example

Regular expression $r = (0+1)^* 00 (0+1)^*$

$L(r)$ = { all strings containing substring 00 }

Example

Regular expression $r = (1 + 01)^* (0 + \lambda)$

$L(r)$ = { all strings without substring 00 }

Regular Expressions

EXAMPLE 2.1 The expression $0(0+1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

EXAMPLE 2.2 The expression $0+1+0(0+1)^*0+1(0+1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit. Note the inclusion of the strings 0 and 1 as special cases.

EXAMPLE 2.3 The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1's, respectively.

EXAMPLE 2.4 The expressions $(0+1)^*1(0+1)^*1(0+1)^*$, $(0+1)^*10^*1(0+1)^*$, $0^*10^*1(0+1)^*$, and $(0+1)^*10^*10^*$ all represent the same set of strings that contain at least two 1's.

Equivalent Regular Expressions

Definition:

Regular expressions r_1 and r_2

are **equivalent** if $L(r_1) = L(r_2)$

Example

L = { all strings without substring 00 }

$$r_1 = (1 + 01)^*(0 + \lambda)$$

$$r_2 = (1^*011^*)^*(0 + \lambda) + 1^*(0 + \lambda)$$

$$L(r_1) = L(r_2) = L \Longrightarrow$$

r_1 and r_2 are equivalent regular expressions

Kleene algebra

Another benefit: regular expressions can be manipulated using algebraic laws (Kleene algebra). For example:

$$
\begin{array}{rclcrcl}
\alpha + (\beta + \gamma) & \equiv & (\alpha + \beta) + \gamma & \quad & \alpha + \beta & \equiv & \beta + \alpha \\
\alpha + \emptyset & \equiv & \alpha & & \alpha + \alpha & \equiv & \alpha \\
\alpha(\beta\gamma) & \equiv & (\alpha\beta)\gamma & & \epsilon\alpha & \equiv & \alpha\epsilon \ \equiv \ \alpha \\
\alpha(\beta + \gamma) & \equiv & \alpha\beta + \alpha\gamma & & (\alpha + \beta)\gamma & \equiv & \alpha\gamma + \beta\gamma \\
\emptyset\alpha & \equiv & \alpha\emptyset \ \equiv \ \emptyset & & \epsilon + \alpha\alpha^* & \equiv & \epsilon + \alpha^*\alpha \equiv \alpha^*
\end{array}
$$

Often these can be used to simplify regular expressions down to more pleasant ones.

Regular Expression: The IEEE POSIX standard

Character	Meaning	Examples
[ ]	alternatives	/[aeiou]/, /m[ae]n/
-	range	/[a-z]/
[^]	not	/[^pbm]/, /[^ox]s/
?	optionality	/Kath?mandu/
*	zero or more	/baa*!/
+	one or more	/ba+!/
.	any character	/cat.[aeiou]/
^, $	start, end of line	
\	not special character	\.\?\^
|	alternate strings	/cat|dog/
( )	substring	/cit(y|ies)/

etc.

Regular Expressions

Valid Email Addresses

Valid IP Addresses

Valid Dates

Floating Point Numbers

Variables

Integers

Numeric Values

Naming Regular Expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum    ::= letter | digit
```

Grammar-like notation for named RE's: a regular grammar

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed,
  unlike full context-free grammars

Specifying Tokens

Identifiers

```
ident       ::= letter (letter | digit)*
```

Integer constants

```
integer     ::= digit+
sign        ::= + | -
signed_int  ::= [sign] integer
```

Real number constants

```
real        ::= signed_int
                [fraction] [exponent]
fraction    ::= . digit+
exponent    ::= (E|e) signed_int
```

RE specification of initial MiniJava lexical structure

```
Program        ::= (Token | Whitespace)*

Token          ::= ID | Integer | ReservedWord |
                   Operator | Delimiter
ID             ::= Letter (Letter | Digit)*
Letter         ::= a | ... | z | A | ... | Z
Digit          ::= 0 | ... | 9
Integer        ::= Digit+
ReservedWord::= class | public | static |
                   extends | void | int |
                   boolean | if | else |
                   while | return | true | false |
                   this | new | String | main |
                   System.out.println
Operator       ::= + | - | * | / | < | <= | >= |
                   > | == | != | && | !
Delimiter      ::= ; | . | , | = |
                   ( | ) | { | } | [ | ]

Whitespace     ::= <space> | <tab> | <newline>
```

Regular Expressions and Regular Languages

Theorem

$$\left\{\begin{array}{l}\text{Languages}\\ \text{Generated by}\\ \text{Regular Expressions}\end{array}\right\} = \left\{\begin{array}{l}\text{Regular}\\ \text{Languages}\end{array}\right\}$$

Theorem (Kleene 1956):

We say that a language $L \subseteq \Sigma^\star$ is regular *if there exists a regular expression r such that $L = L(r)$. In this case, we also say that r* represents *the language L.*

Proof:

$$\left\{\begin{array}{l}\text{Languages}\\ \text{Generated by}\\ \text{Regular Expressions}\end{array}\right\} \subseteq \left\{\begin{array}{l}\text{Regular}\\ \text{Languages}\end{array}\right\}$$

$$\left\{\begin{array}{l}\text{Languages}\\ \text{Generated by}\\ \text{Regular Expressions}\end{array}\right\} \supseteq \left\{\begin{array}{l}\text{Regular}\\ \text{Languages}\end{array}\right\}$$

Proof - Part 1

$$\left\{\begin{array}{l}\text{Languages}\\ \text{Generated by}\\ \text{Regular Expressions}\end{array}\right\} \subseteq \left\{\begin{array}{l}\text{Regular}\\ \text{Languages}\end{array}\right\}$$

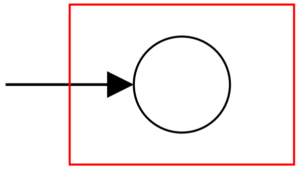
For any regular expression r
the language $L(r)$ is regular

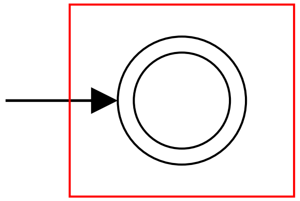
Proof by induction on the size of r

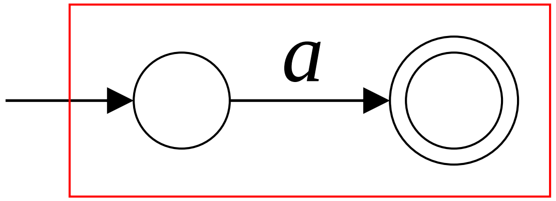
Induction Basis

Primitive Regular Expressions: $\varnothing, \quad \lambda, \quad \alpha$

Corresponding NFAs







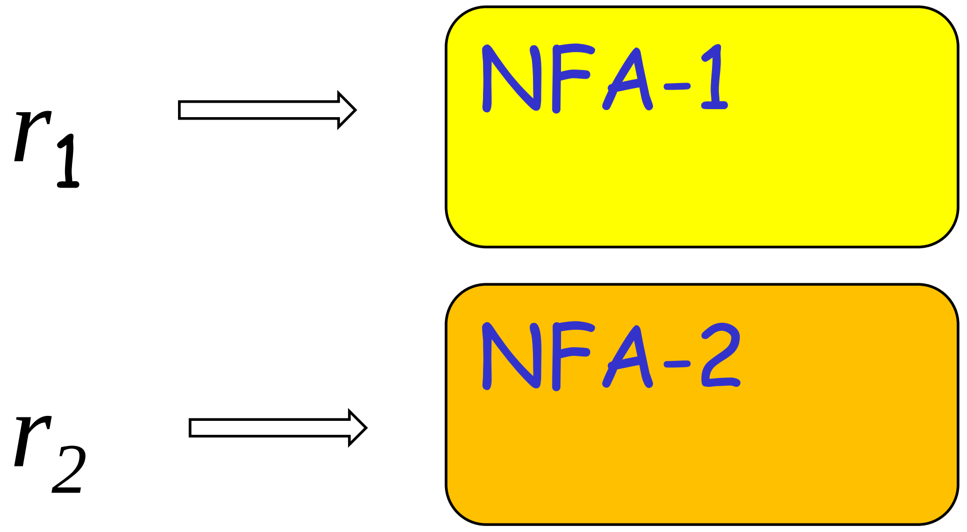
$$\left.\begin{array}{l} L(M_1)=\varnothing=L(\varnothing)\\[2em] L(M_2)=\{\lambda\}=L(\lambda)\\[2em] L(M_3)=\{a\}=L(a)\end{array}\right\}\text{regular languages}$$

Inductive Hypothesis

Suppose

that for regular expressions r_1 and r_2,

$L(r_1)$ and $L(r_2)$ are regular languages



Inductive Step

By inductive hypothesis we know:

$L(r_1)$ and $L(r_2)$ are regular languages

We will prove: There exits NFA for regular expressions

$$r_1 + r_2$$

$$r_1 \cdot r_2$$

$$r_1 *$$

$$(r_1)$$

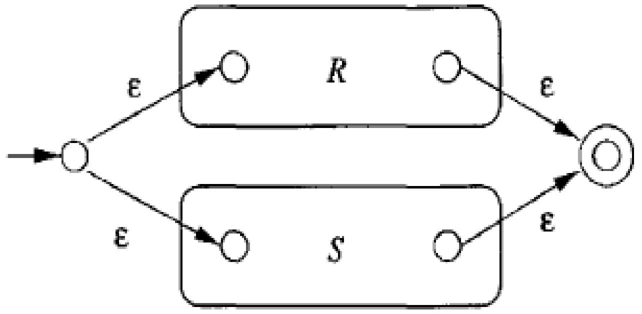
Inductive Step

If we can, then:

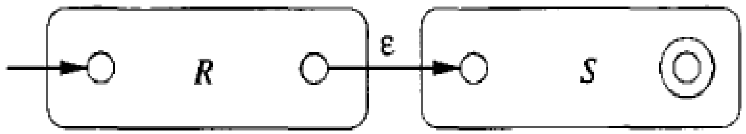
$$\left.\begin{array}{l} L(r_1 + r_2) \\ \\ L(r_1 \cdot r_2) \\ \\ L(r_1 *) \\ \\ L((r_1)) \end{array}\right\}$$

Are regular Languages

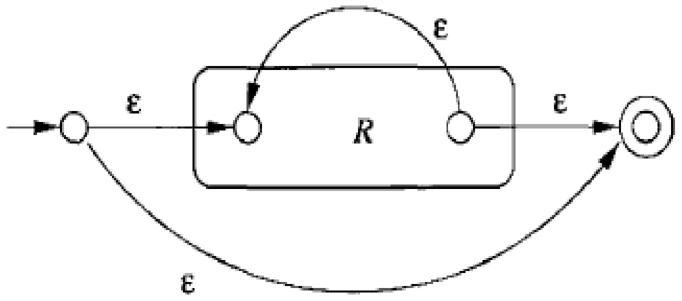
Inductive Step



(a)



(b)



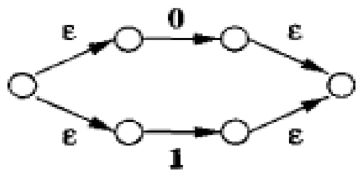
(c)

End of Proof-Part 1

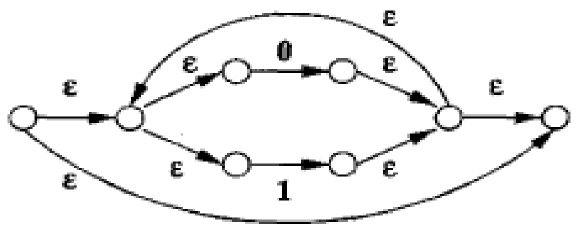
Example

From RegExps to clean NFAs: example

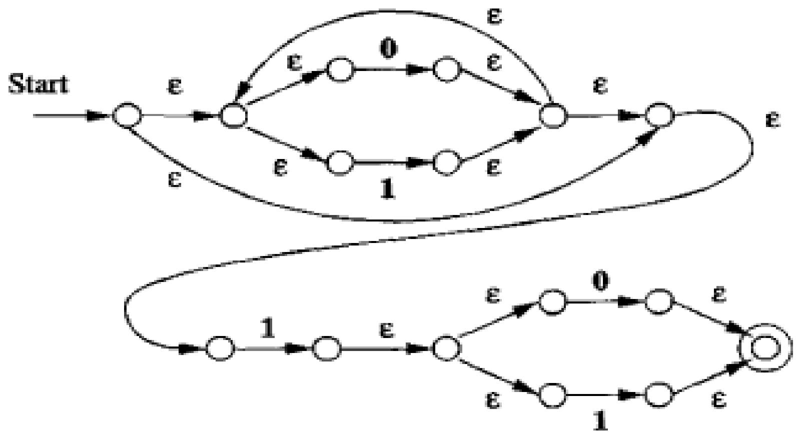
Conversion of a regexp $(0 \cup 1)^*1(0 \cup 1)$ into an NFA:



(a)



(b)



Interesting Properties of Regular languages

Regular languages are closed under:

Union	$L(r_1) \cup L(r_2)$
Concatenation	$L(r_1)\,L(r_2)$
Star	$(L(r_1))^*$

We can prove the closure property

$$\left.\begin{aligned} L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 \cdot r_2) &= L(r_1)\, L(r_2) \\ L(r_1 *) &= (L(r_1))* \end{aligned}\right\} \text{Are regular languages}$$

$L((r_1)) = L(r_1)$ is trivially a regular language

Proof - Part 2

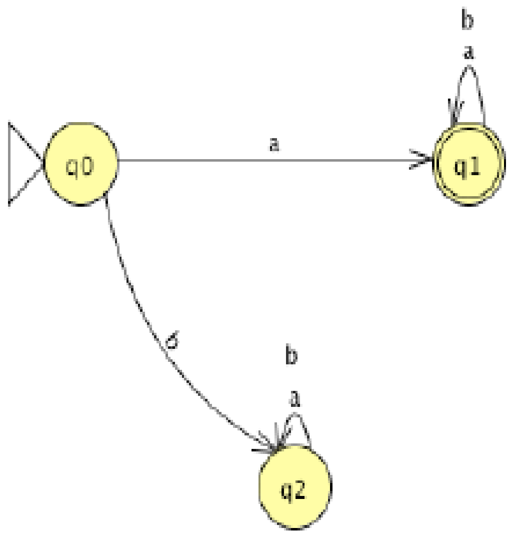
$$\left\{\begin{array}{l}\text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions}\end{array}\right\} \supseteq \left\{\begin{array}{l}\text{Regular} \\ \text{Languages}\end{array}\right\}$$

For any regular language L there is a regular expression r with $L(r) = L$

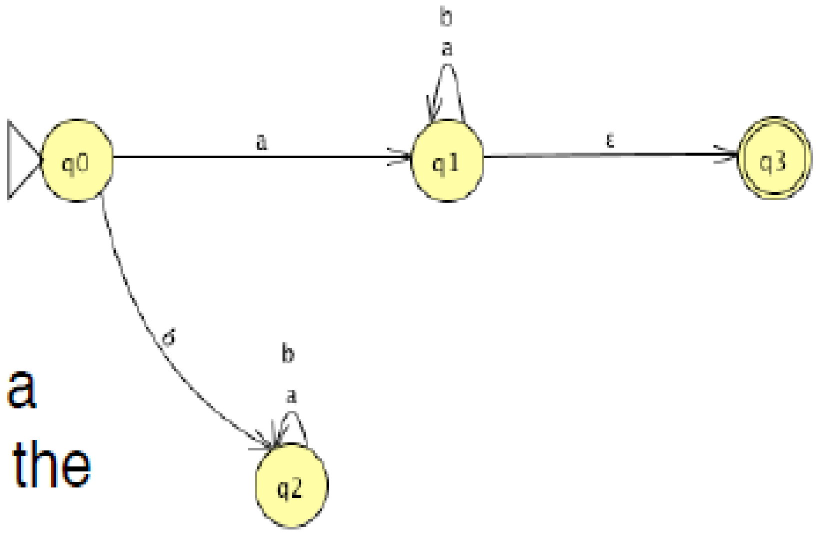
We will convert an NFA that accepts L to a regular expression

NFA to RE

0. If there is no arc from state *i* to state *j*, imagine one with label ∅.

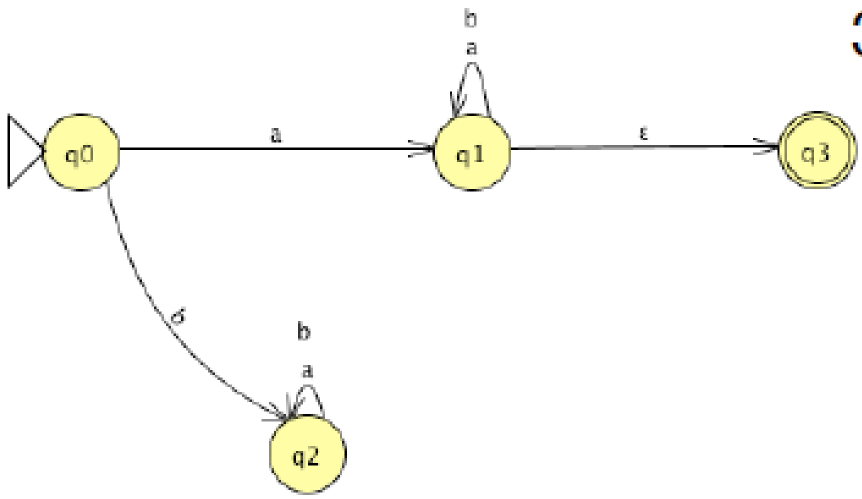


1. If the initial state has a self-transition, create a new initial state with a single ε-transition to the old initial state.

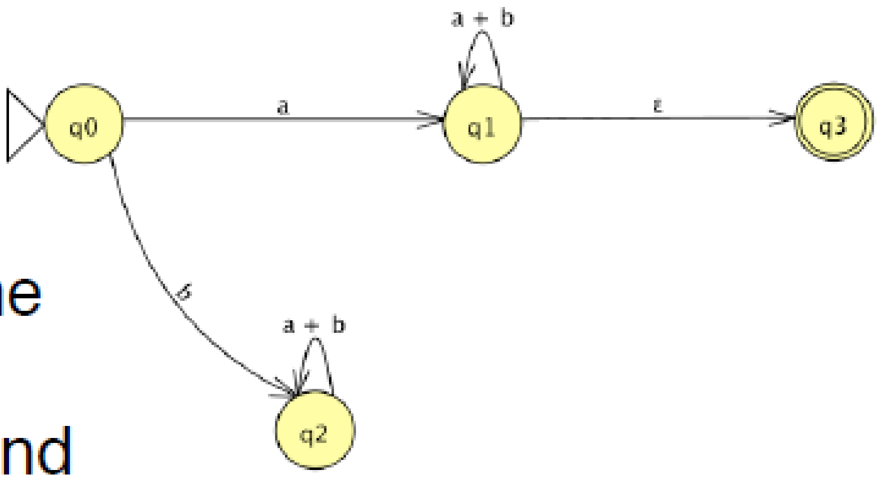


2. Create a new final state with a ε-transition to it from each of the old final states.

NFA to RE

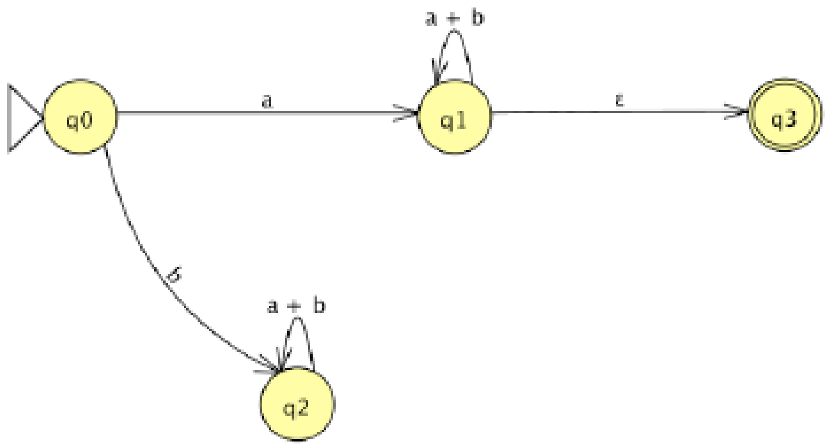


3. For each pair of states *i*, *j* with more than one transition from *i* to *j*, replace them all by a single transition labeled with the r.e. that is the sum of the old labels.



4. Eliminate one state at a time until the only states that remain are the start state and the final state:

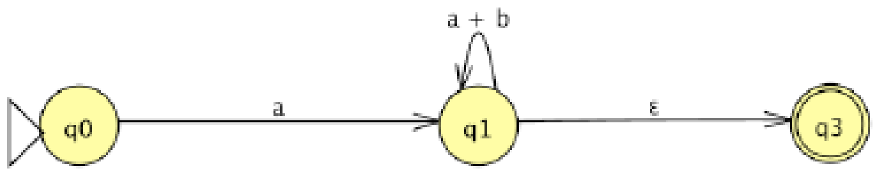
NFA to RE

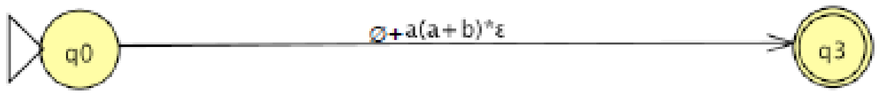


- For each pair of nodes *i*, *j* (*i* ≠ *k*, *j* ≠ *k*), label the transition from *i* to *j* with:

  $$(i, j) + (i, k)(k, k)^*(k, j)$$

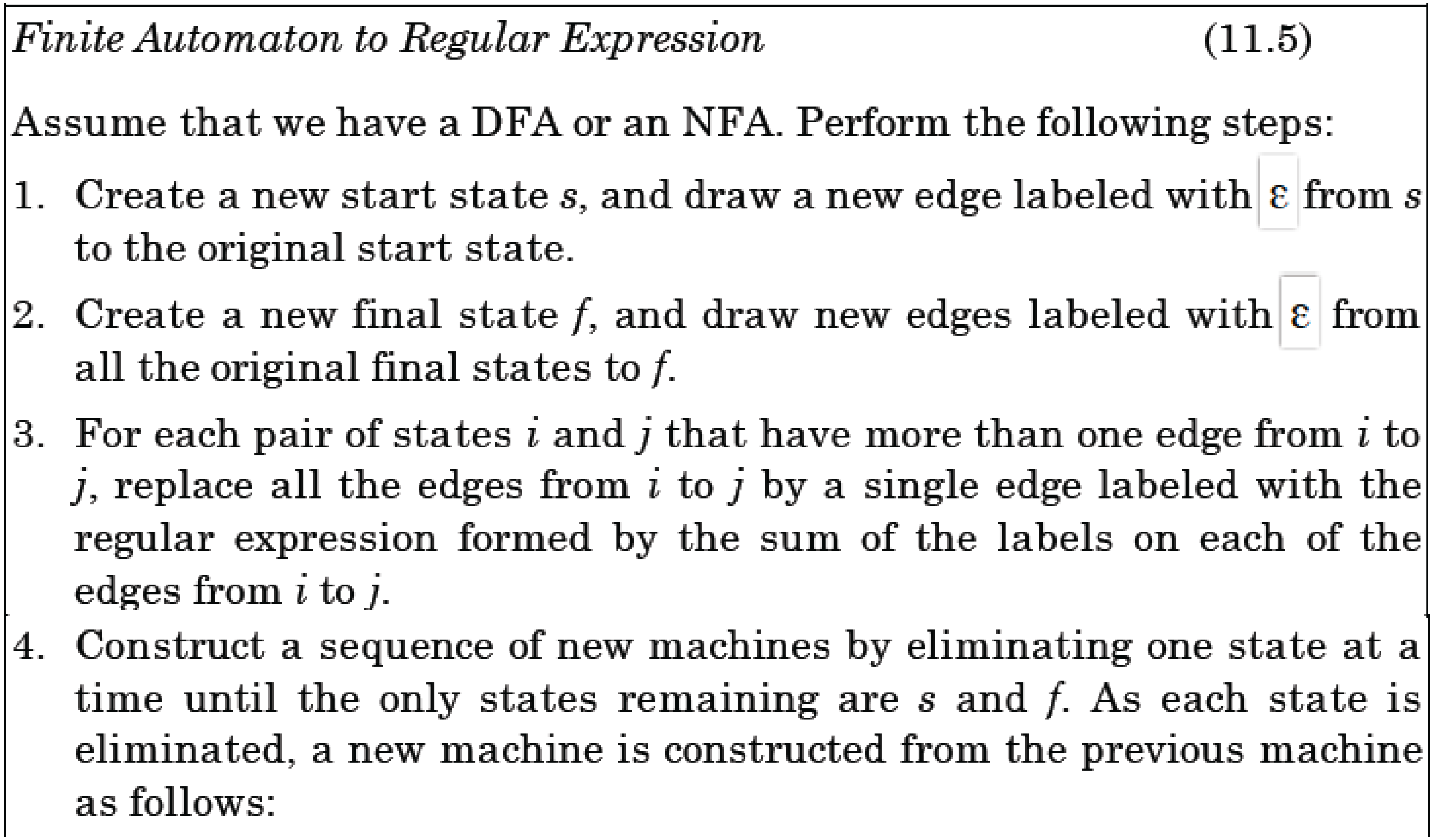
- Remove state *k* and all its transitions.





NFA to RE

The Algorithm from Hein:



Finite Automaton to Regular Expression (11.5)

Assume that we have a DFA or an NFA. Perform the following steps:

1. Create a new start state *s*, and draw a new edge labeled with $\boxed{\varepsilon}$ from *s* to the original start state.
2. Create a new final state *f*, and draw new edges labeled with $\boxed{\varepsilon}$ from all the original final states to *f*.
3. For each pair of states *i* and *j* that have more than one edge from *i* to *j*, replace all the edges from *i* to *j* by a single edge labeled with the regular expression formed by the sum of the labels on each of the edges from *i* to *j*.
4. Construct a sequence of new machines by eliminating one state at a time until the only states remaining are *s* and *f*. As each state is eliminated, a new machine is constructed from the previous machine as follows:

NFA to RE

Eliminate State k

For convenience we'll let old(i, j) denote the label on edge (i, j) of the current machine. If there is no edge (i, j), then set old(i, j) = $\varnothing$. Now for each pair of edges (i, k) and (k, j), where $i \neq k$ and $j \neq k$, calculate a new edge label, new(i, j), as follows:

$$\text{new}(i, j) = \text{old}(i, j) + \text{old}(i, k)\ \text{old}(k, k)^*\ \text{old}(k, j).$$

For all other edges (i, j) where $i \neq k$ and $j \neq k$, set

$$\text{new}(i, j) = \text{old}(i, j).$$

The states of the new machine are those of the current machine with state k eliminated. The edges of the new machine are the edges (i, j) for which label new(i, j) has been calculated.

Now s and f are the two remaining states. If there is an edge (s, f), then the regular expression new(s, f) represents the language of the original automaton. If there is no edge (s, f), then the language of the original automaton is empty, which is signified by the regular expression $\varnothing$.

NFA to RE

From NFAs to RegExps

Theorem

Every regular language is described by some regular expression.

The proof is done by construction of a regular expression for a language A, given an NFA recognizing A. The construction uses a generalized form of NFA (GNFA).

So, transforming an NFA to an equivalent regexp will be done as follows:

$$\text{NFA} \longrightarrow \text{GNFA} \longrightarrow \text{RegExp}$$

NFA to RE

Generalized NFA

Definition

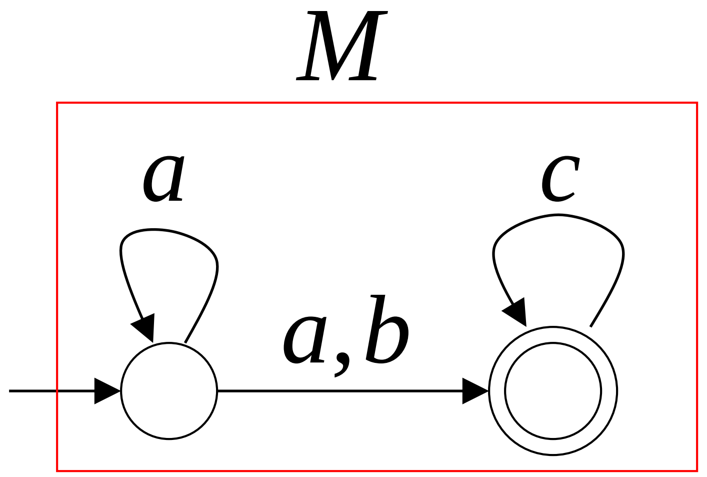
Let REGEXP be the set of all regular expressions. A *generalized nondeterministic finite automaton* (GNFA) is a tuple $G = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set (the set of states),
- Σ is a finite alphabet,
- $q_0 \in Q$ (the start state),
- $F \subseteq Q$ (the set of final states), and
- $\delta : Q \times Q \to \mathrm{REGEXP}$ is the transition function.

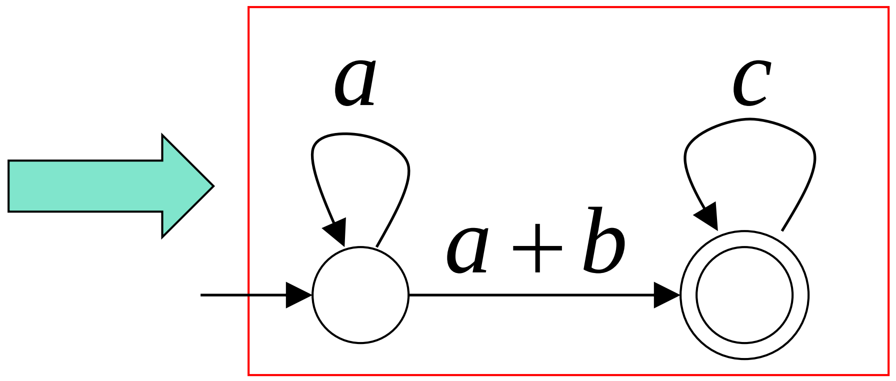
The transition diagram for a GNFA is the same for an NFA, except that there is exactly one edge from each state q to each state r, and the edges are labeled with regular expressions, the edge (q, r) being labeled with $\delta(q, r)$. The edges labeled with the regexp $\emptyset$ can be omitted from the diagram as they can never be followed.

From M construct the equivalent
Generalized Transition Graph
in which transition labels are regular expressions

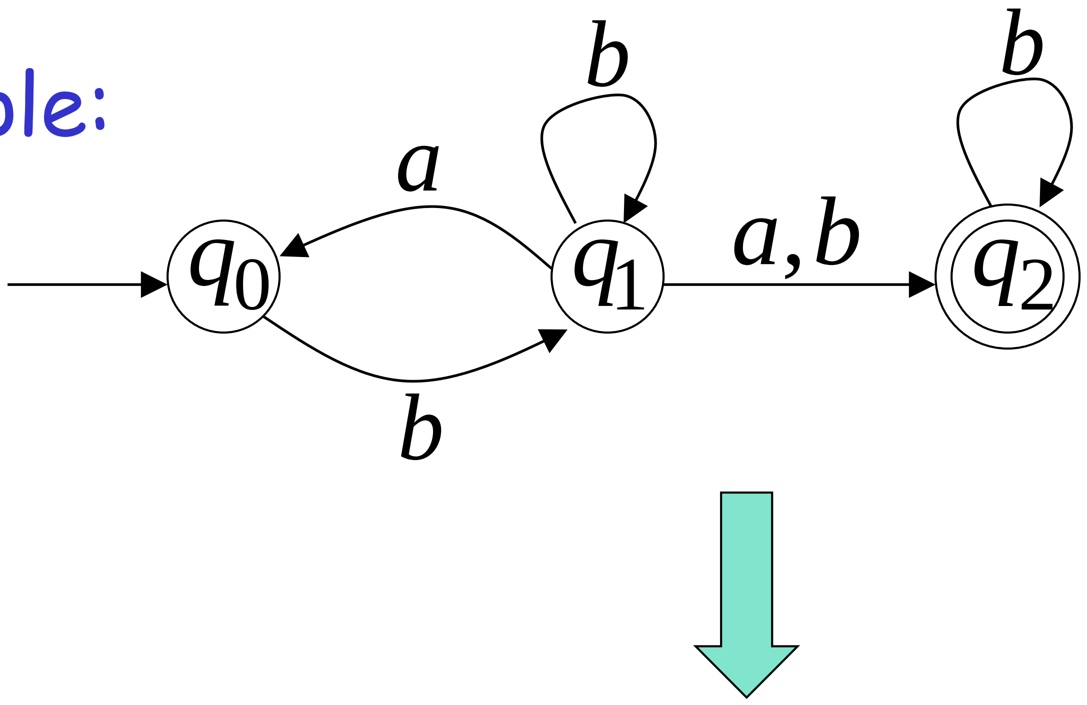
Example:



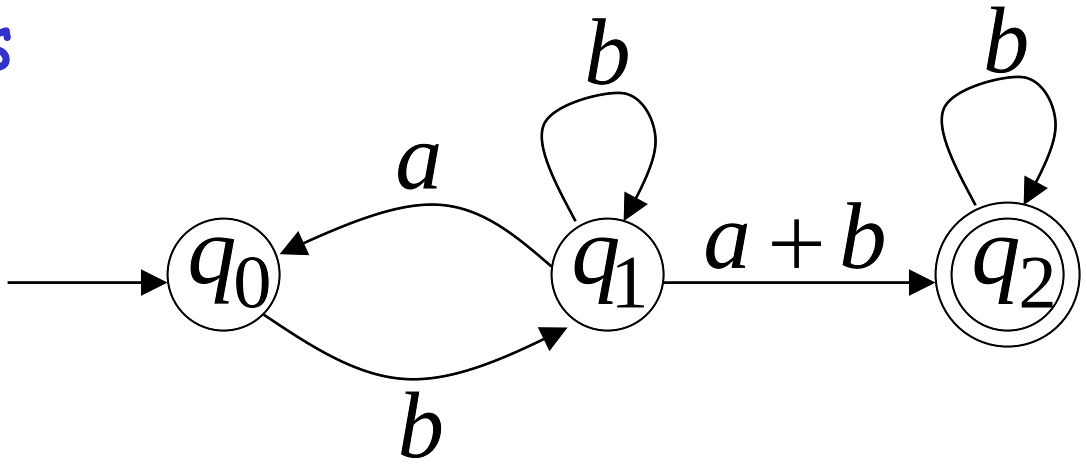
Corresponding
Generalized transition graph



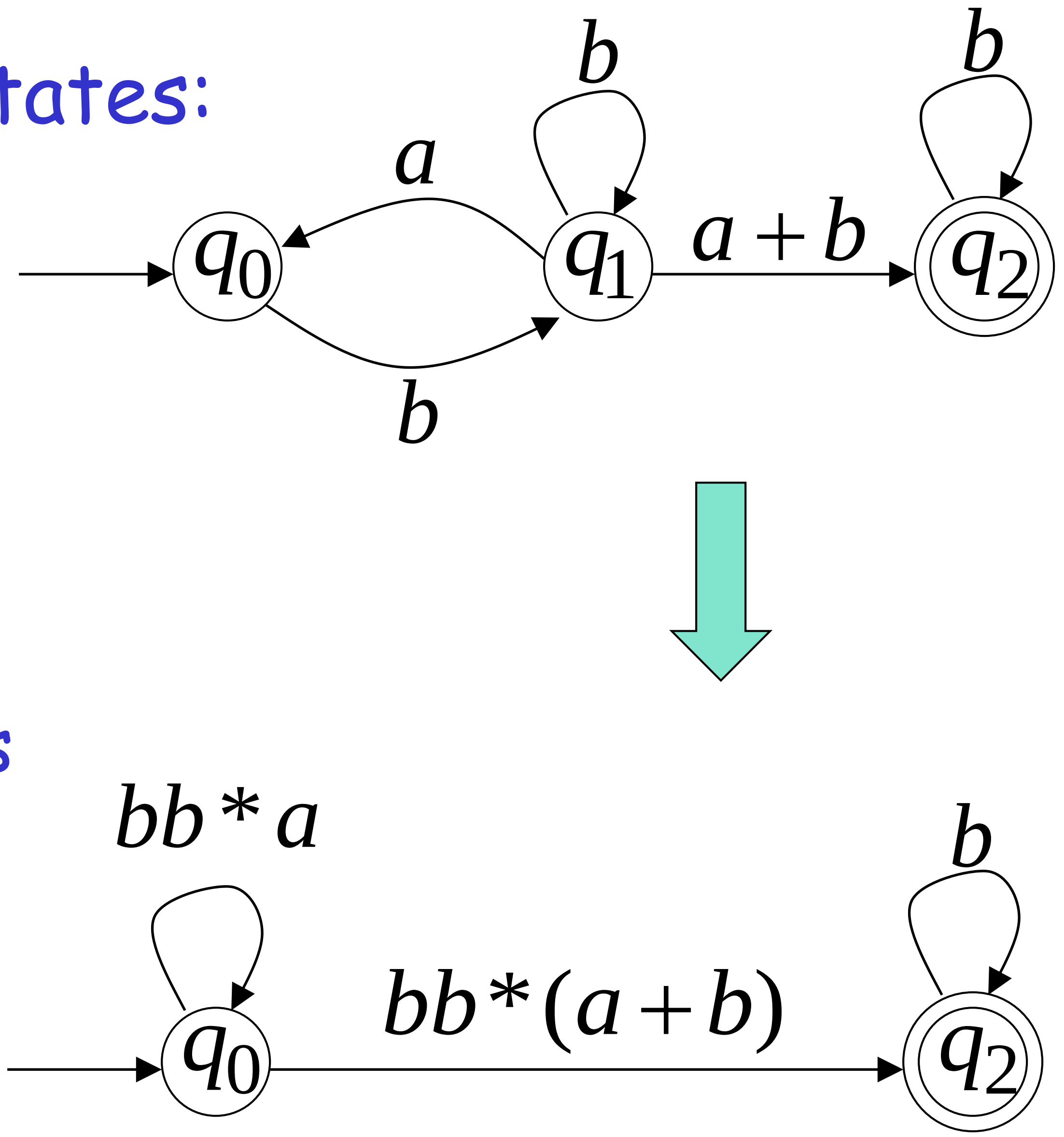
Another Example:



Transition labels are regular expressions

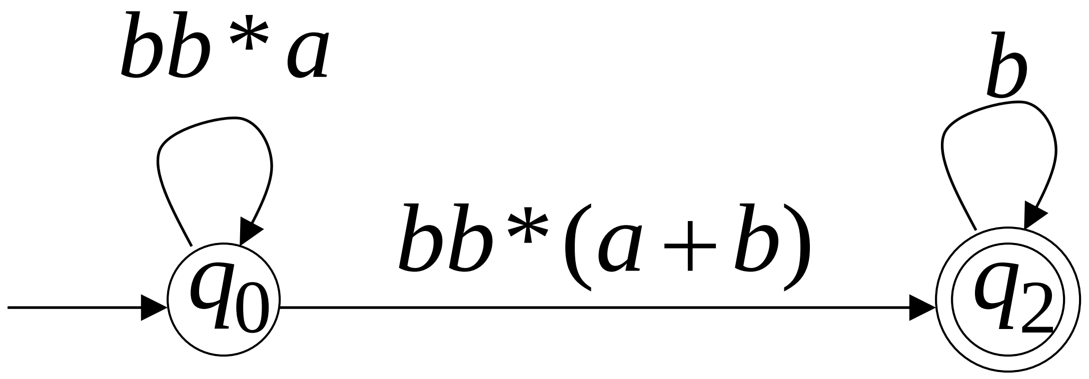


Reducing the states:



Transition labels are regular expressions

Resulting Regular Expression:

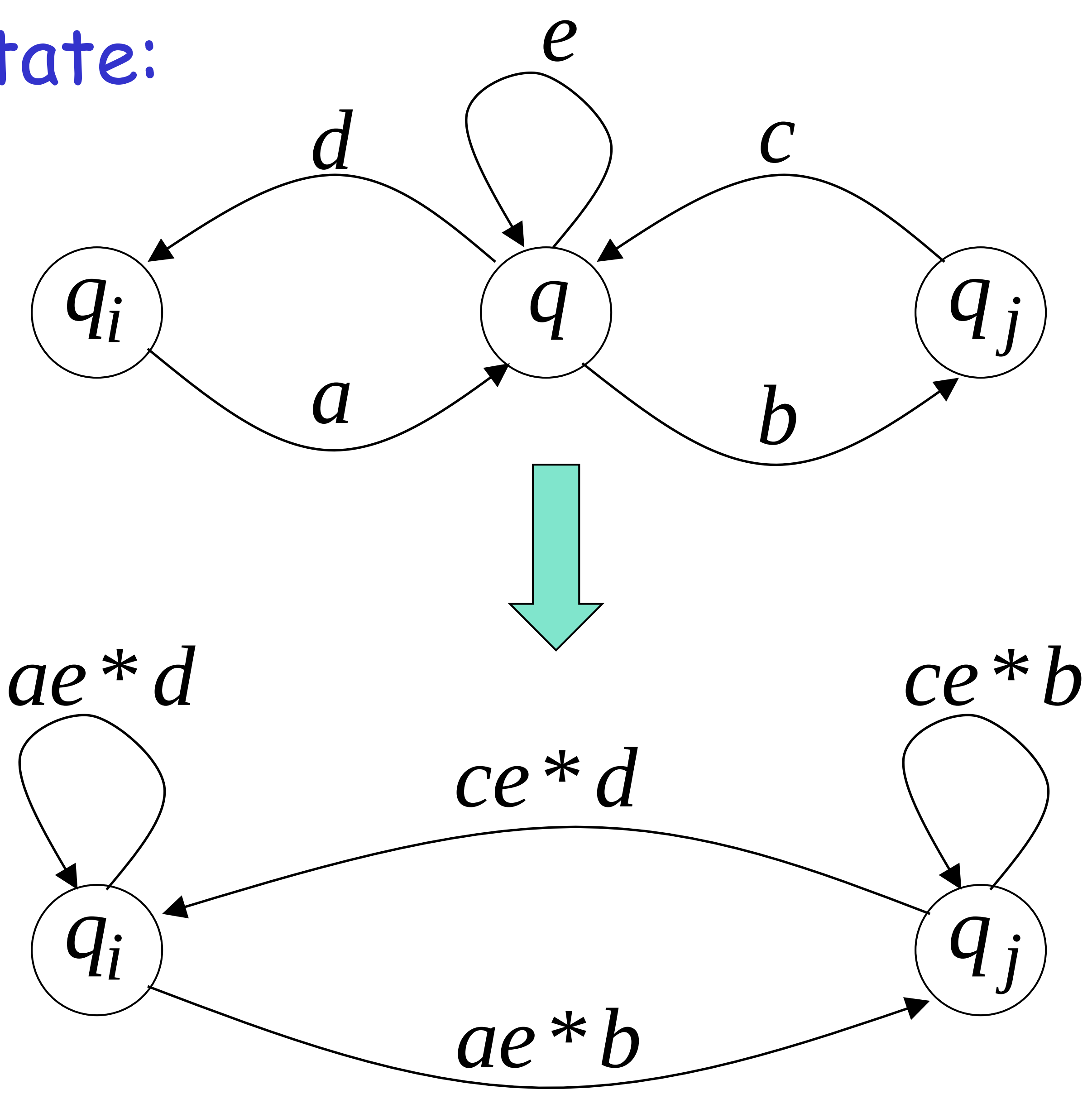


$$r = (bb^*a)^*bb^*(a+b)b^*$$

$$L(r) = L(M) = L$$

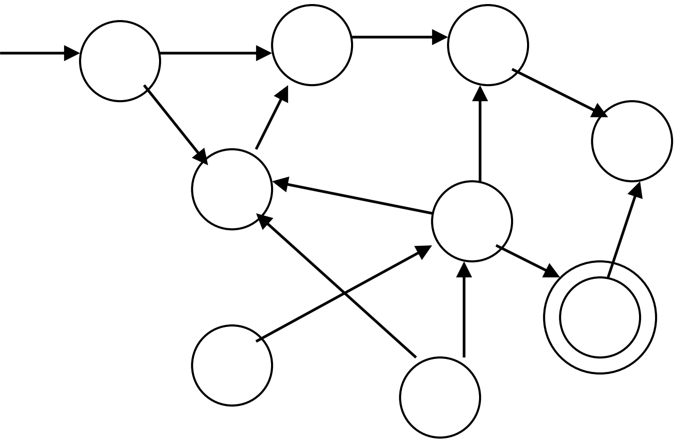
In General

Removing a state:

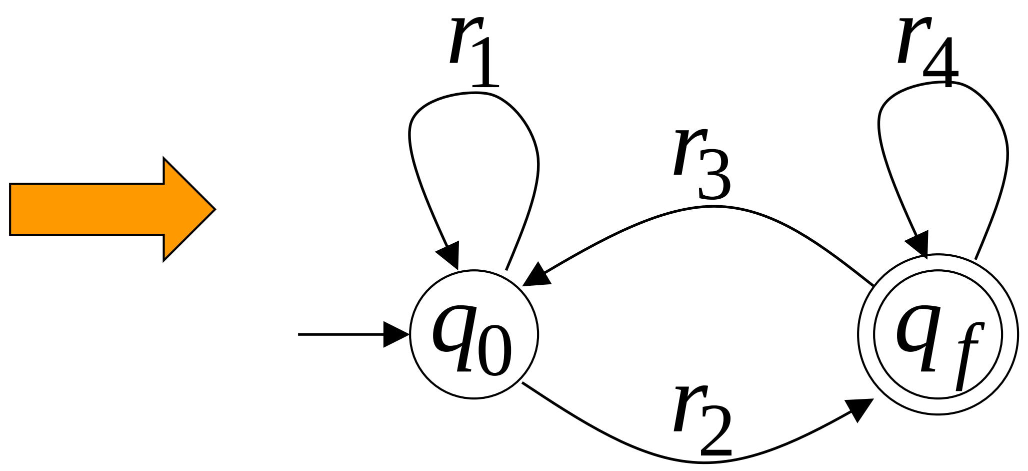


By repeating the process until two states are left, the resulting graph is

Initial graph



Resulting graph



The resulting regular expression:

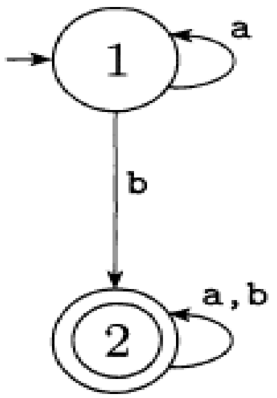
$$r = r_1{}^* r_2 (r_4 + r_3 r_1{}^* r_2)^*$$

$$L(r) = L(M) = L$$

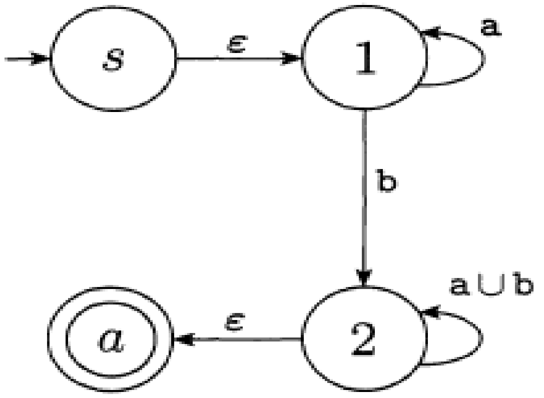
End of Proof-Part 2

NFA to RE

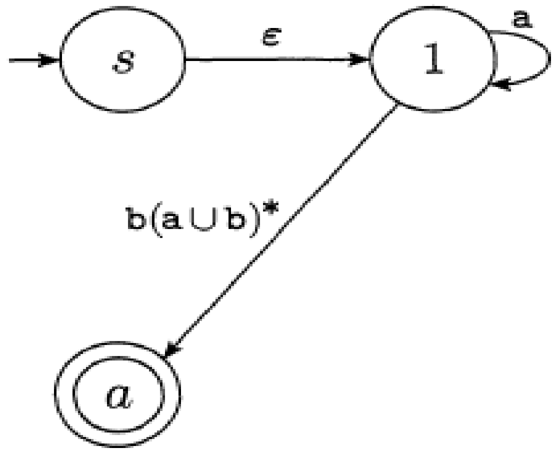
Example: Converting a 2-state DFA into RegExp



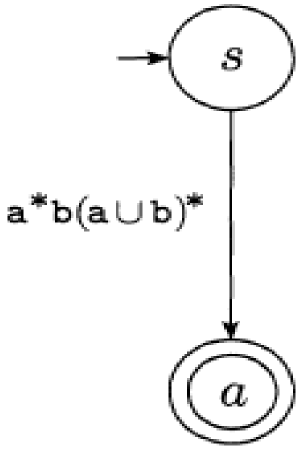
(a)



(b)



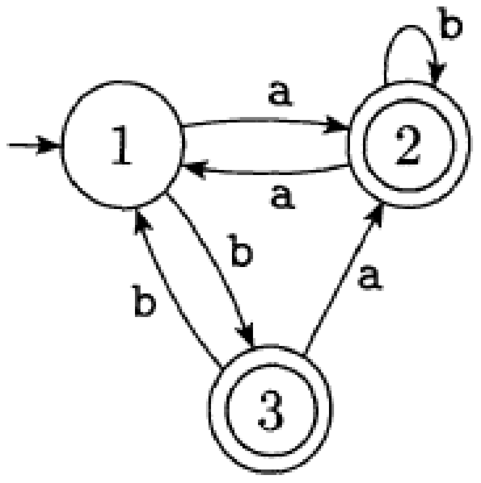
(c)



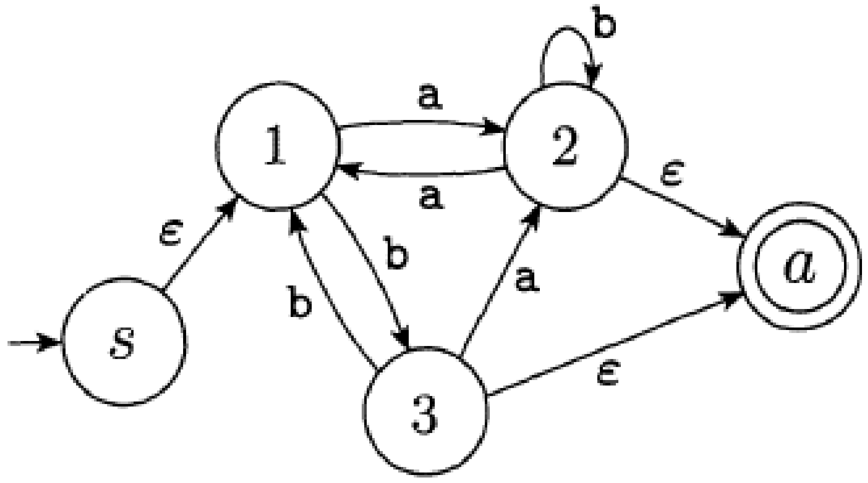
(d)

NFA to RE

Example: Converting a 3-state DFA into RegExp



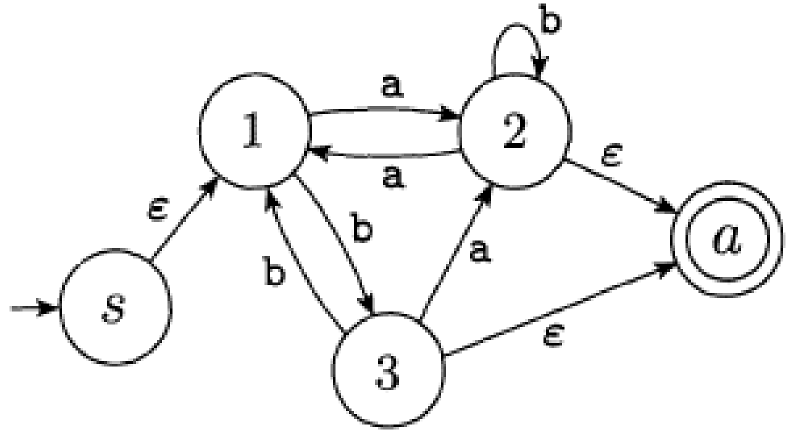
(a)



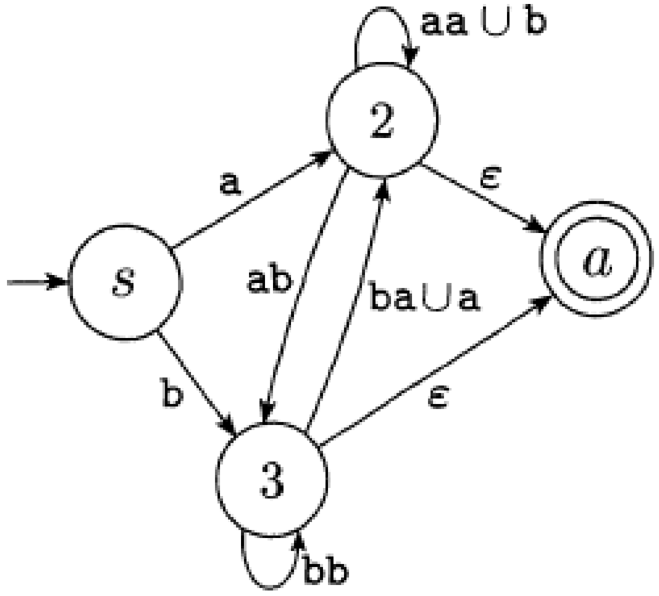
(b)

NFA to RE

Example: Converting a 3-state DFA into RegExp



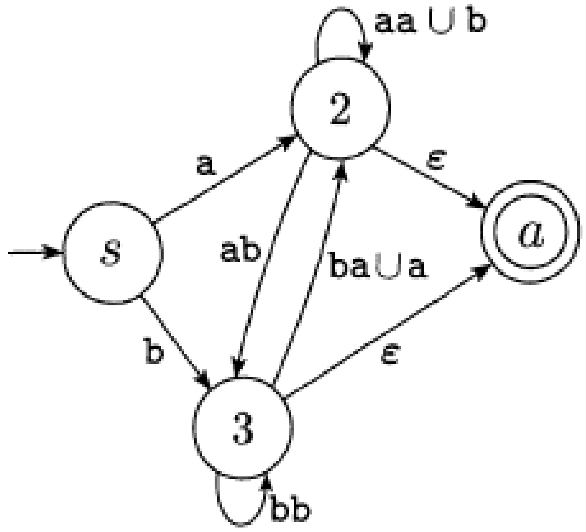
(b)



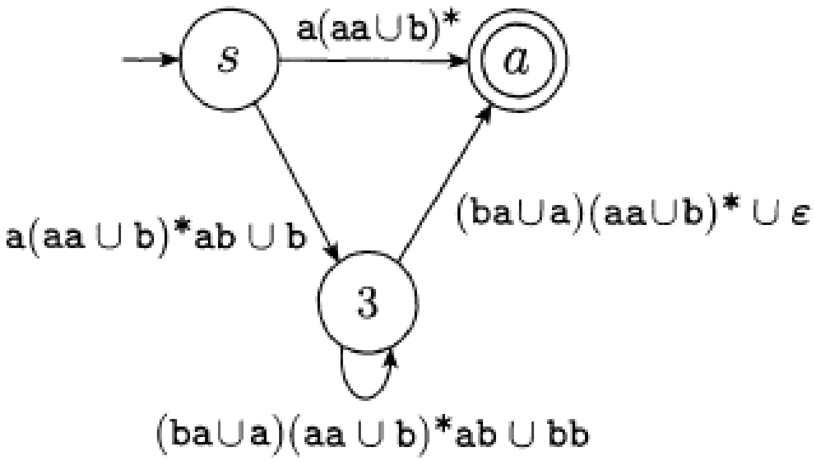
(c)

NFA to RE

Example: Converting a 3-state DFA into RegExp



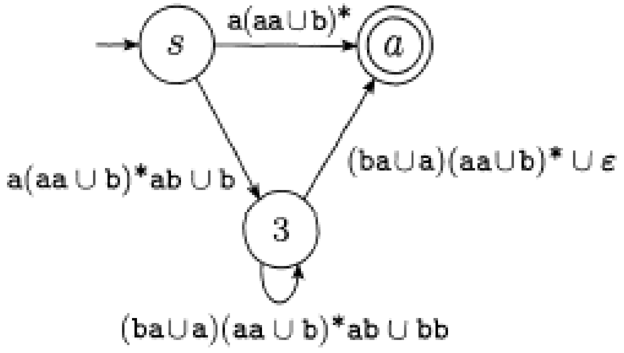
(c)



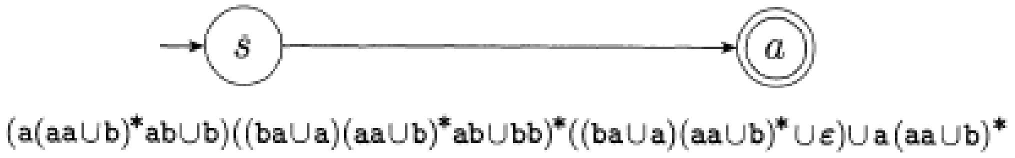
(d)

NFA to RE

Example: Converting a 3-state DFA into RegExp



(d)



s
a
(a(aa∪b)*ab∪b)((ba∪a)(aa∪b)*ab∪bb)*((ba∪a)(aa∪b)*∪ε)∪a(aa∪b)*

(e)

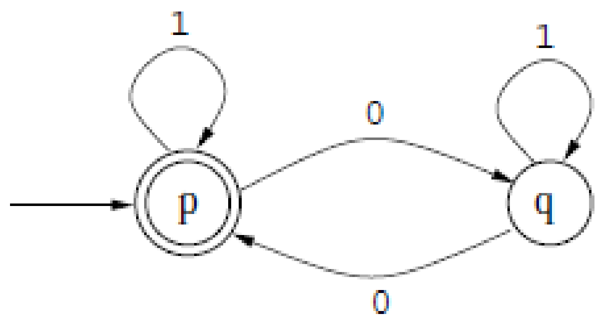
Alternative: NFA to RE

Arden's rule: Given an equation of the form $X = \alpha X + \beta$, its smallest solution is $X = \alpha^* \beta$.

What's more, if $\epsilon \notin \mathcal{L}(\alpha)$, this is the *only* solution.

Intriguing fact: The rules on this slide and the last form a complete set of reasoning principles, in the sense that if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, then '$\alpha \equiv \beta$' is provable using these rules. (Beyond scope of Inf2A.)

Alternative: NFA to RE



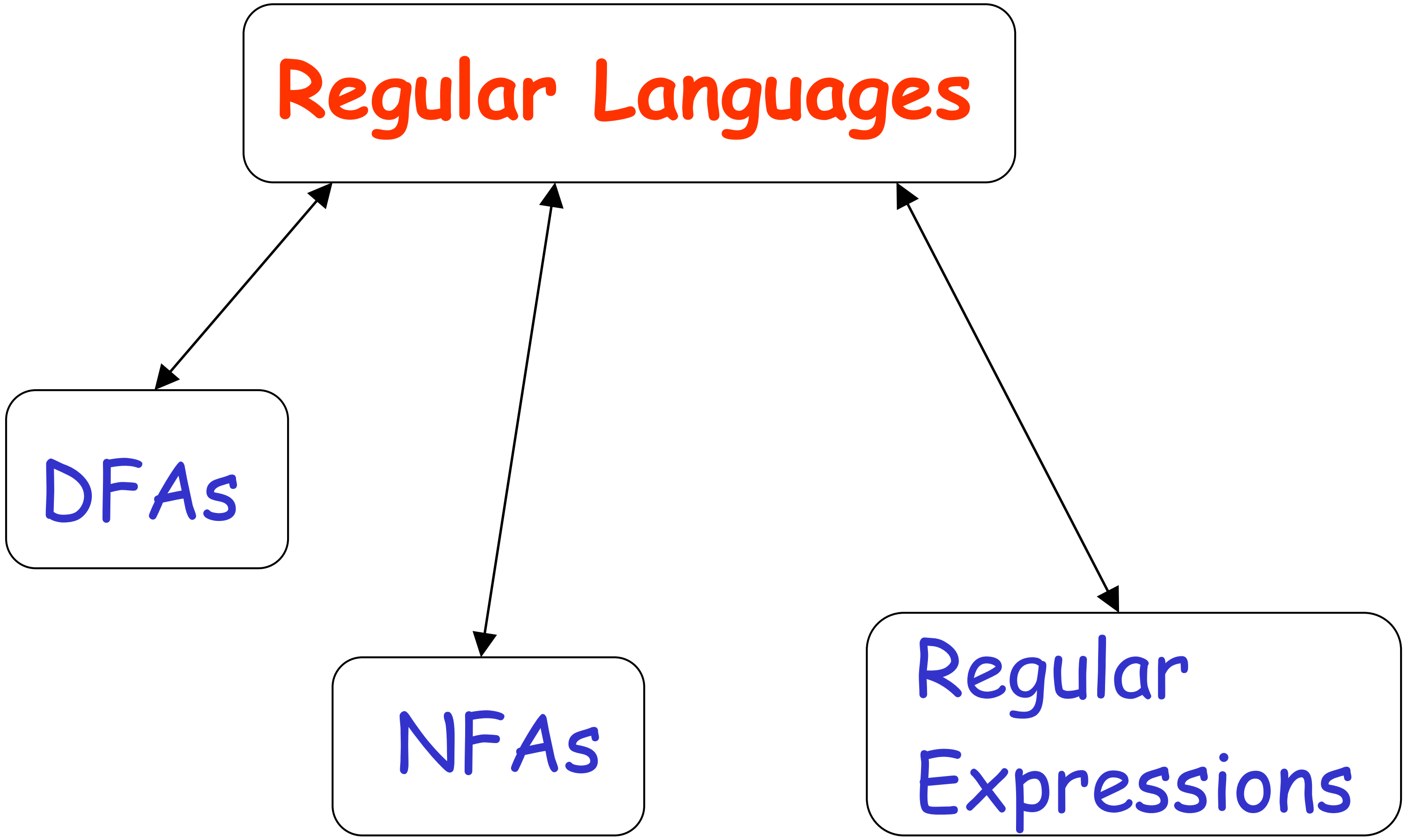
For each state a, let X_a stand for the set of strings that take us from a to an accepting state. Then we can write some equations:

$$
\begin{aligned}
X_p &= 1.X_p + 0.X_q + \epsilon \quad \text{(Final State)} \\
X_q &= 1.X_q + 0.X_p
\end{aligned}
$$

Solve by eliminating one variable at a time:

$$
\begin{aligned}
X_q &= 1^*0.X_p \quad \text{by Arden's rule} \\
\text{So} \quad X_p &= 1.X_p + 01^*0X_p + \epsilon \\
&= (1 + 01^*0)X_p + \epsilon \\
\text{So} \quad X_p &= (1 + 01^*0)^* \quad \text{by Arden's rule}
\end{aligned}
$$

Standard Representations of Regular Languages



When we say: We are given a Regular Language L

We mean: Language L is in a standard representation

(DFA, NFA, or Regular Expression)