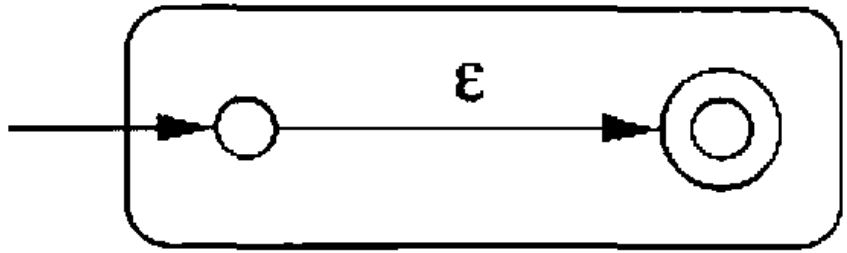
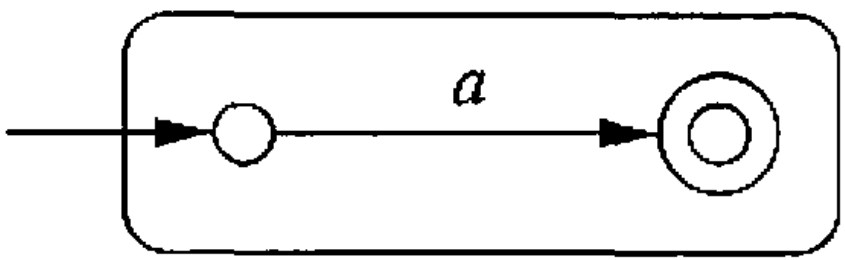
From RegExps to clean NFAs: induction basis



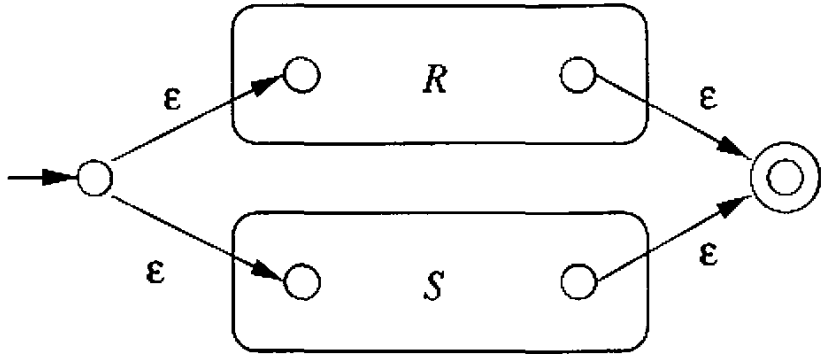
(a)

(b)

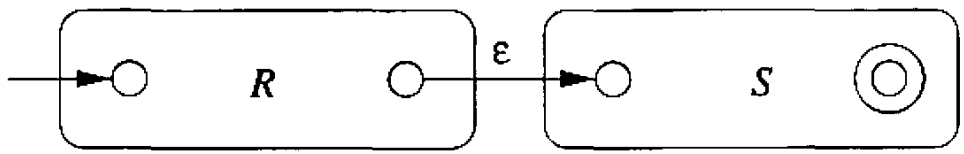


(c)

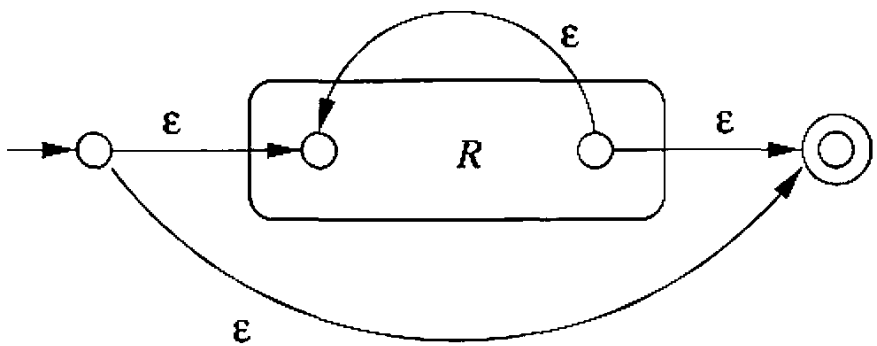
From RegExps to clean NFAs: inductive step



(a)



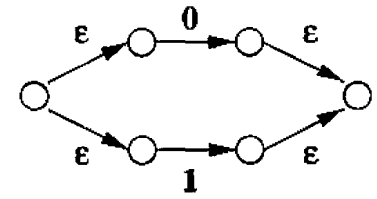
(b)



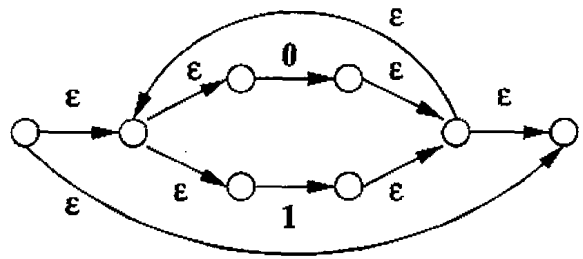
(c)

From RegExps to clean NFAs: example

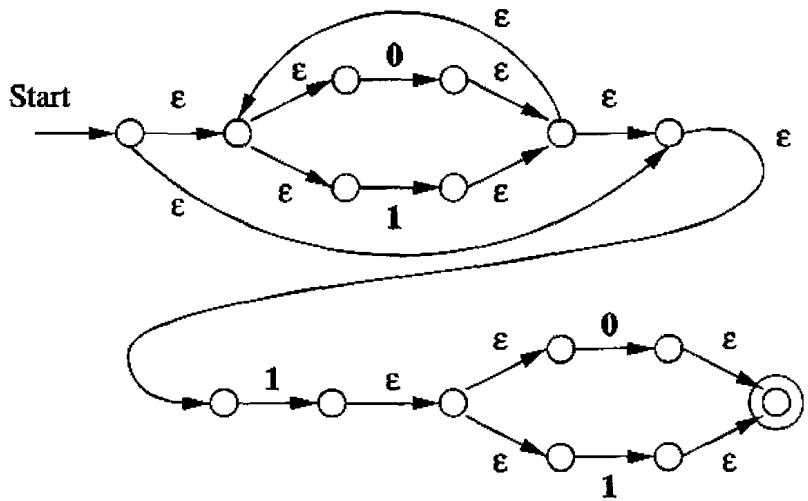
Conversion of a regexp $(0 \cup 1)^*1(0 \cup 1)$ into an NFA:



(a)



(b)



(c)

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:

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

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 NFA to GNFA

We will find it convenient to work with a *clean GNFA* that has a start state with no incoming transitions and a single final state that has no outgoing transitions. So, our goal is to contruct an equivalent clean GNFA from a given NFA.

1. Given an NFA, we first transform it to a clean NFA M.
2. The states GNFA G that we are constructing are the same as in M, with the same starting and final states.
3. For any two states p and q, if there is a transition $p \to q$ labelled $x_1, \ldots, x_k$ in M, we define a transition $p \to q$ labelled by a regexp $x_1 \cup \cdots \cup x_k$ in G; if there is no transition $p \to q$ in M, we define a transition $p \to q$ labelled by a regexp $\emptyset$ in G.

Q: Are we done?

From NFA to GNFA

We will find it convenient to work with a *clean GNFA* that has a start state with no incoming transitions and a single final state that has no outgoing transitions. So, our goal is to contruct an equivalent clean GNFA from a given NFA.

1. Given an NFA, we first transform it to a clean NFA M.
2. The states GNFA G that we are constructing are the same as in M, with the same starting and final states.
3. For any two states p and q, if there is a transition $p \to q$ labelled $x_1, \ldots, x_k$ in M, we define a transition $p \to q$ labelled by a regexp $x_1 \cup \cdots \cup x_k$ in G; if there is no transition $p \to q$ in M, we define a transition $p \to q$ labelled by a regexp $\emptyset$ in G.

Q: Prove that M and G are equivalent. How?

From NFA to GNFA

We will find it convenient to work with a *clean GNFA* that has a start state with no incoming transitions and a single final state that has no outgoing transitions. So, our goal is to contruct an equivalent clean GNFA from a given NFA.

1. Given an NFA, we first transform it to a clean NFA M.
2. The states GNFA G that we are constructing are the same as in M, with the same starting and final states.
3. For any two states p and q, if there is a transition $p \to q$ labelled $x_1, \ldots, x_k$ in M, we define a transition $p \to q$ labelled by a regexp $x_1 \cup \cdots \cup x_k$ in G; if there is no transition $p \to q$ in M, we define a transition $p \to q$ labelled by a regexp $\emptyset$ in G.

Q: Prove that for every computation in M, there exists an equivalent computation in G; and vice versa.

From GNFA to RegExp

If a constructed GNFA G has just two states, we are done (Q: why?).

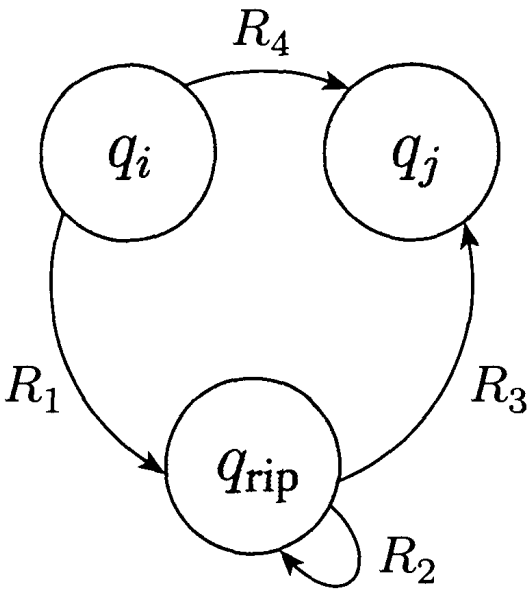
From GNFA to RegExp

If a constructed GNFA G has just two states, we are done (Q: why?).

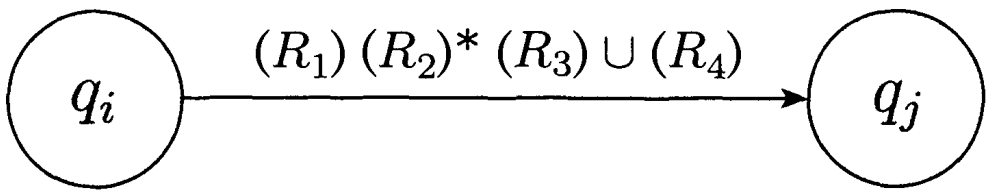
For a constructed GNFA G with $k > 2$ states, we will eliminate states (other than the start and final ones) one by one and modify the transitions such that a new GNFA G' with $k - 1$ states is equivalent to the original one.
We will continue in similar way until we get an equivalent GNFA with just two states. From this 2-state GNFA we will derive a regexp as above.

From GNFA to RegExp: Eliminating States

- Select any state, which we denote q_{rip}, other than the start and final states.
- GNFA G' has the same states as G, except q_{rip}.
- For any two other states q_i and q_j, let $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$. In G' transition from q_i to q_j is labeled $R_1 R_2^* R_3 \cup R_4$.

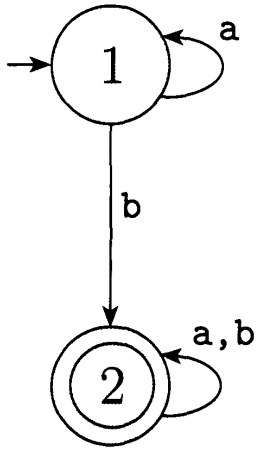


before

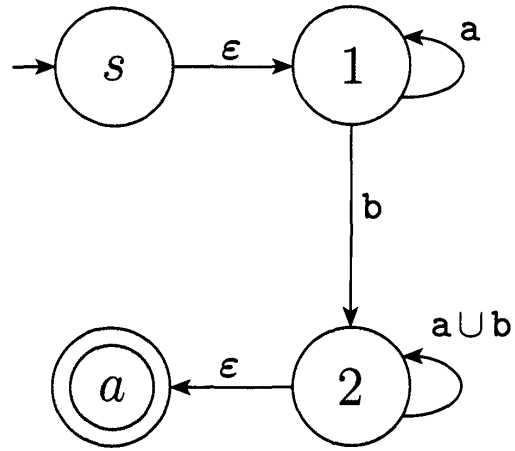


after

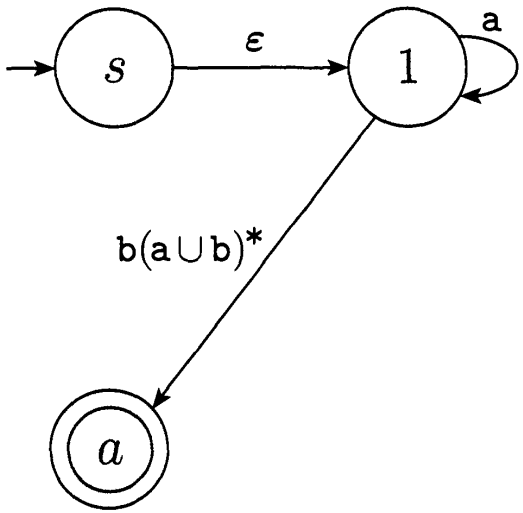
Example: Converting a 2-state DFA into RegExp



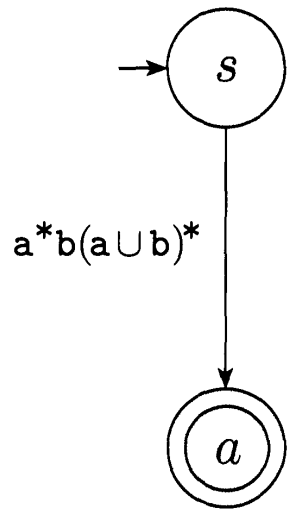
(a)



(b)

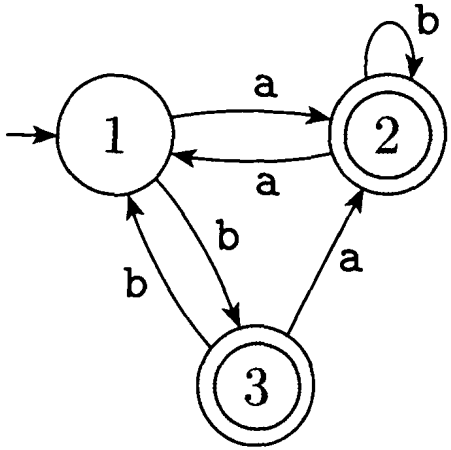


(c)

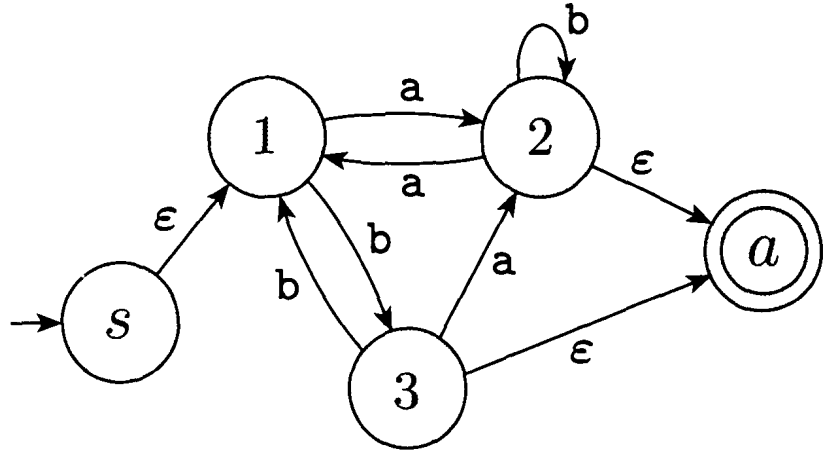


(d)

Example: Converting a 3-state DFA into RegExp

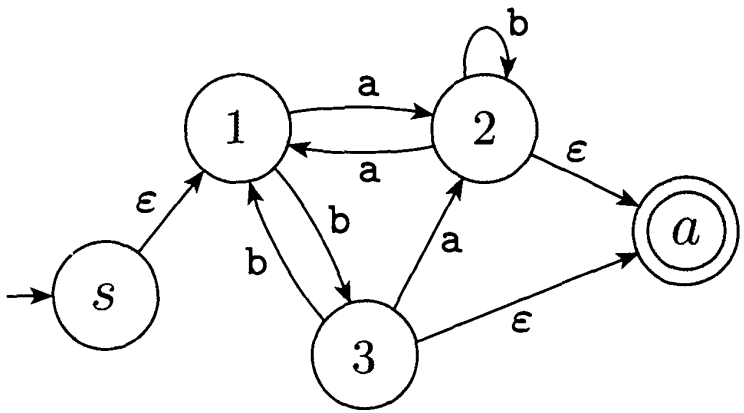


(a)

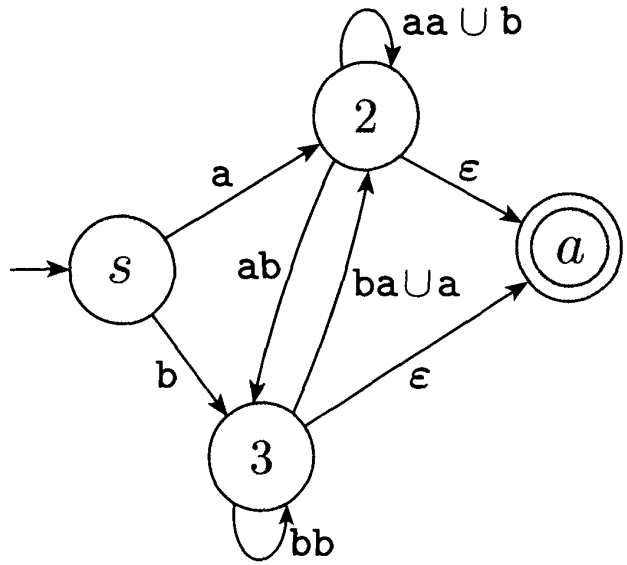


(b)

Example: Converting a 3-state DFA into RegExp

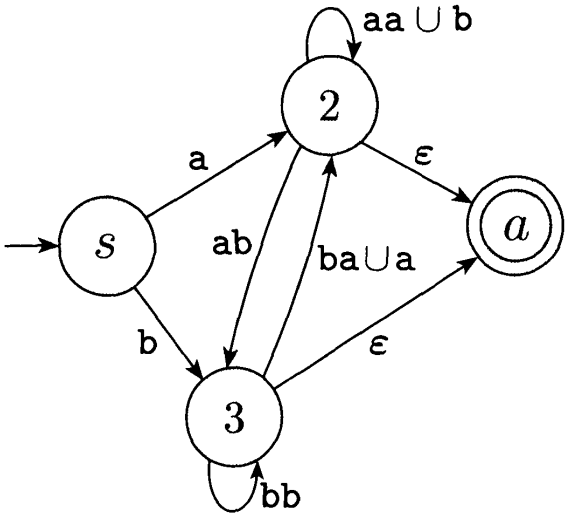


(b)

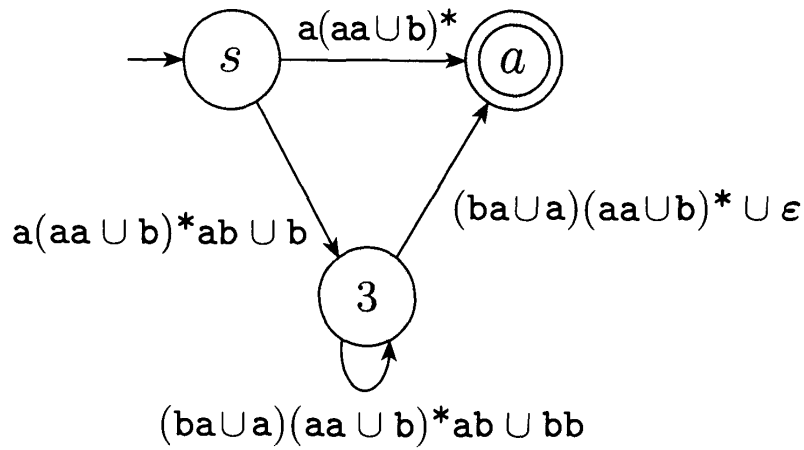


(c)

Example: Converting a 3-state DFA into RegExp

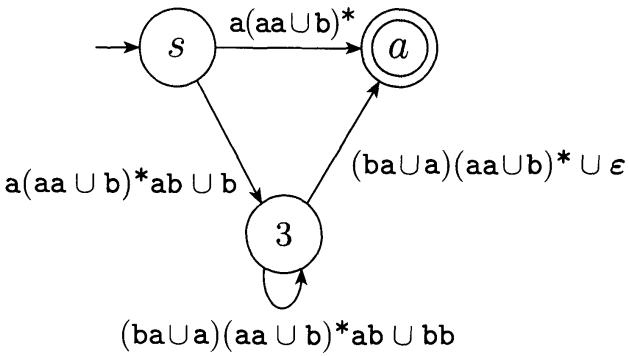


(c)



(d)

Example: Converting a 3-state DFA into RegExp



(d)

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

(e)

Algebraic Regular Expressions

Algebraic regular expressions may contain variables that may stand for any language (over any alphabet).

Associativity and Commutativity

- $L + M = M + L$, the *commutativity law for union*
- $(L + M) + N = L + (M + N)$, the *associativity law for union*
- $(LM)N = L(MN)$, the *associativity law for concatenation*

Identities and Annihilators

- $\emptyset + L = L + \emptyset = L$, i.e., the empty regexp is the identity for union.
- $\epsilon L = L\epsilon = L$, i.e., the empty string regexp is the identity for concatenation.
- $\emptyset L = L\emptyset = \emptyset$, i.e., the empty regexp is the annihilator for concatenation.

Distributive Laws

- $L(M+N) = LM + LN$, the *left distributive law of concatenation over union*.
- $(M+N)L = ML + NL$, the *right distributive law of concatenation over union*.

For example, we can factor 0 out of the regexp $0 + 01^*$ as follows:

$$0 + 01^* = 0\epsilon + 01^* = 0(\epsilon + 1^*).$$

The first equality is due the fact that ϵ is the identity for concatenation, while the second equality is due the left distributive law.
We can further recognize that $\epsilon \in 1^*$ and simplify the regexp to 01^*.

Proof for Left Distributive Law

Theorem

If L, M, and N are any languages, then

$$L(M \cup N) = LM \cup LN.$$

Proof.

We need to show that $w \in L(M \cup N)$ if and only if $w \in LM \cup LN$.

- If $w \in L(M \cup N)$, then $w = xy$ for some $x \in L$ and y is in either M or N. If $y \in M$, then $xy \in LM$ and therefore $xy \in LM \cup LN$. Likewise, if $y \in N$, then $xy \in LN$ and therefore $xy \in LM \cup LN$.
- If $w \in LM \cup LN$, then w is in either LM or LN. If $w \in LM$, then $w = xy$ for some $x \in L$ and $y \in M$. Since y is in M, it is also in $M \cup N$ and therefore $xy \in L(M \cup N)$. The case of $w \in LN$ is proved similarly.

Idempotence Law

- $L + L = L$, the *idempotence law for union*.

Laws Involving Closures

- $(L^*)^* = L^*$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $L^+ = LL^* = L^*L$
- $L^* = L^+ + \epsilon$
- $L? = \epsilon + L$ (the definition of the ? operator)

RegExps in UNIX

UNIX regexp operators:

- The operator | is used for union.
- The operator ? means "zero or one of". Thus, `R?` is the same as $\epsilon + R$.
- The operator + means "one or more of". Thus, `R+` is the same as R^+.
- The operator $\{n\}$ means "n copies of". Thus, `R{5}` is the same as R^5.