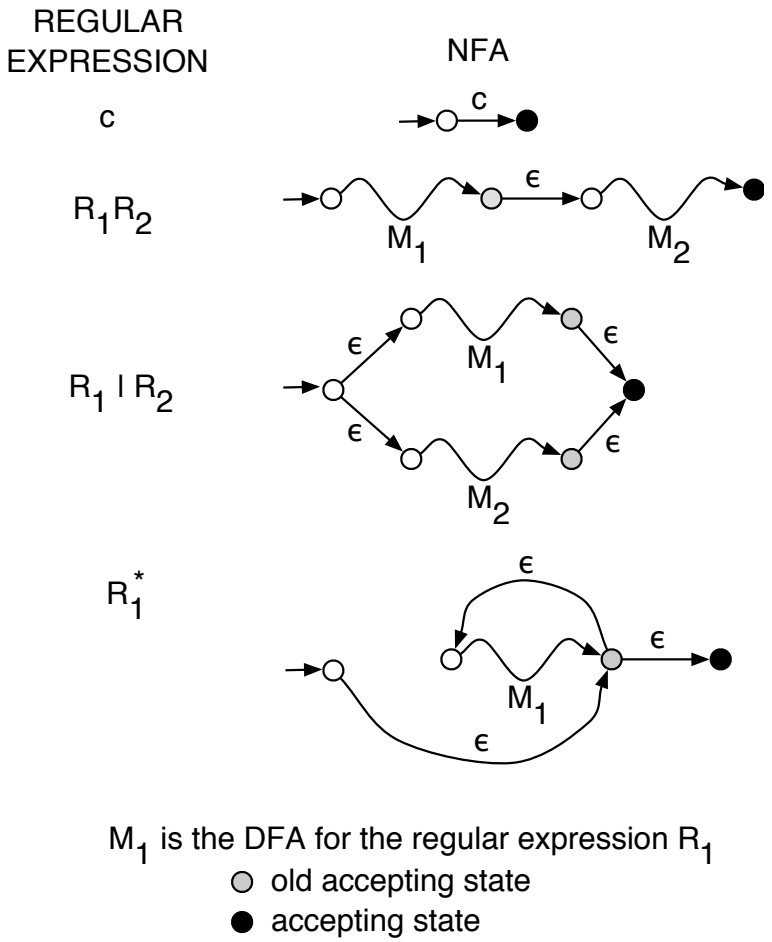
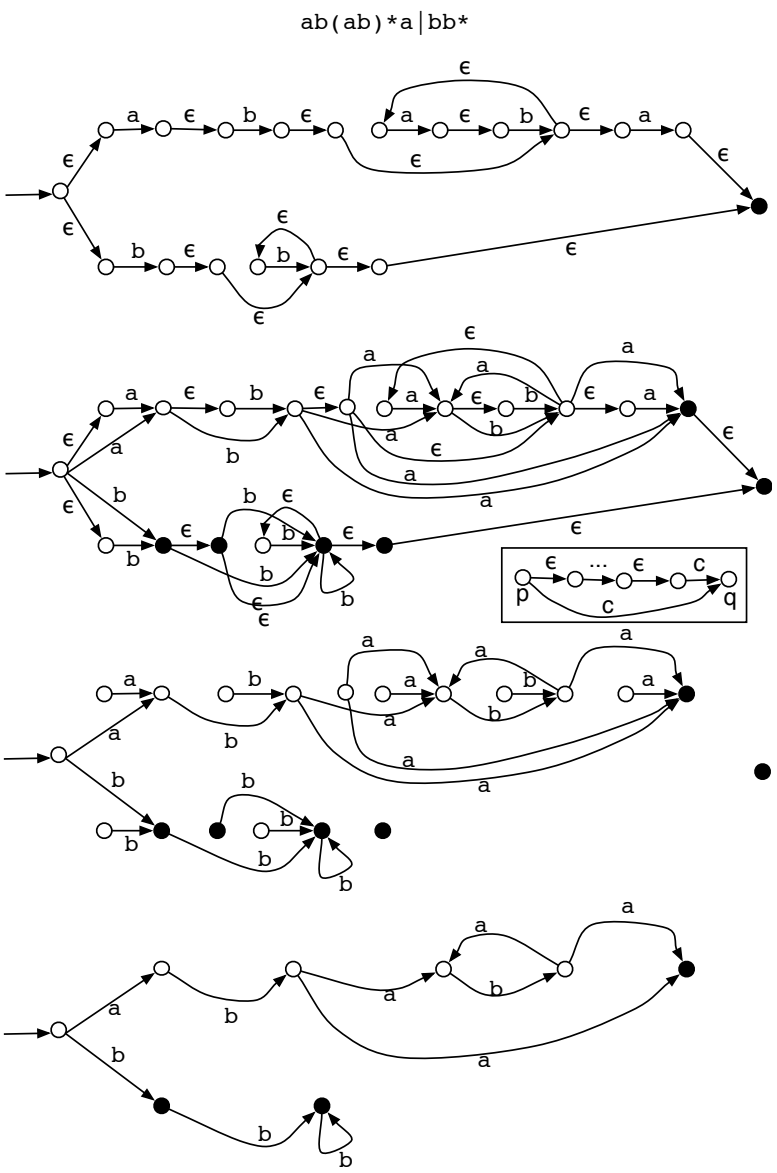
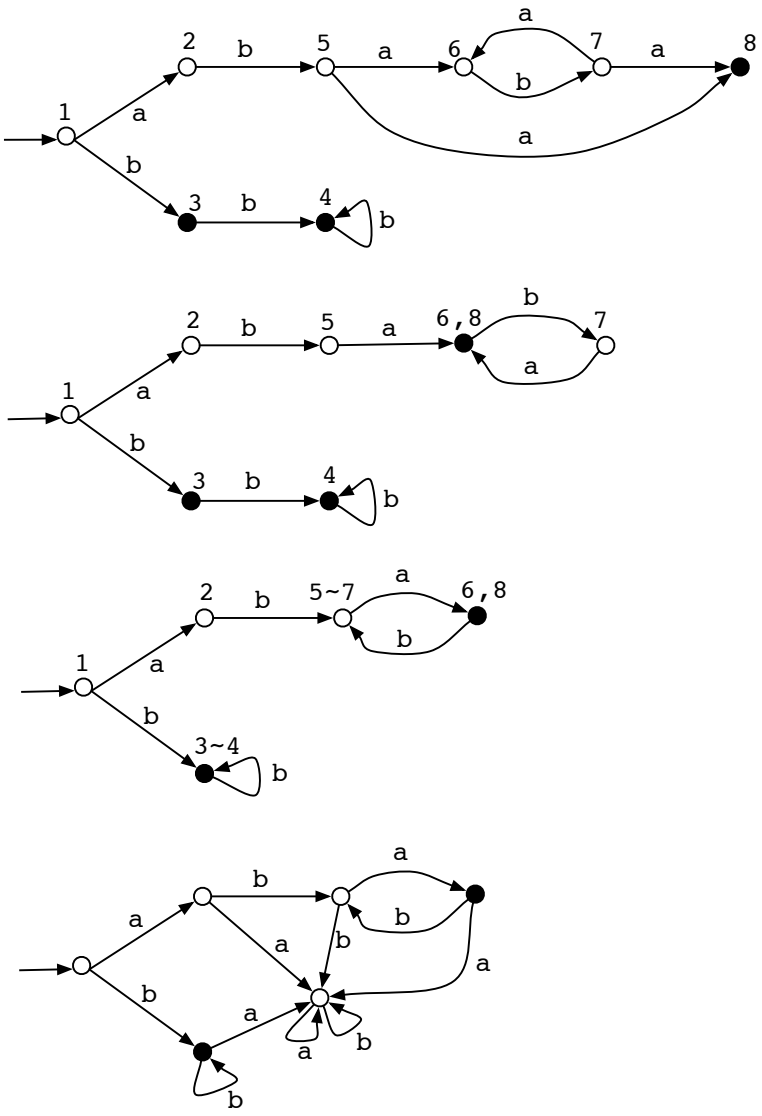


Four cases used to build NFA's from regular expressions consisting of a *single character* c, *concatenation*, *union*, and *Kleene closure*. In each case there is only one start state (with no edges leading to it) and only one accepting state (with no edges emanating from it).



An NFA constructed from the regular expression `ab(ab)*a|bb*`. We then switch some states to accepting states and add edges to bypass ϵ-transitions. The ϵ-transitions are then removed. Unreachable states and their associated edges are then deleted.



We label each edge and use the *subset construction* to convert the NFA into a DFA. In some cases this may be impractical since the resulting DFA may require $2^{|Q|}$ states! We then minimize the number of states using the *quotient construction.* The final DFA is non-blocking.

Algorithm for simulating a non-blocking DFA $M = (Q, \Sigma, \delta, s, F)$:

$q = s$; /* q = start state */
while (input not empty) {
  scan(c);
  $q = \delta(q, c)$;
}
accept iff ($q \in F$); /* in accepting state? */

Algorithm for simulating a blocking DFA:

$q = s$;
while (input not empty) {
  scan(c);
  if ($\delta(q, c)$ defined)
    $q = \delta(q, c)$;
  else
    reject;
}
accept iff ($q \in F$);

Algorithm for simulating NFA $M = (Q, \Sigma, \Delta, S, F)$ without ϵ-transitions:

$P = S$; /* P = set containing all start states */
while (input not empty) {
  scan(c);
  $T = \emptyset$;
  for (each $\Delta(q, c)$ where $q \in P$)
    $T = T \cup \Delta(q, c)$;
  if ($T \subseteq \emptyset$) /* blocked */
    reject;
  $P = T$;
}
accept iff ($P \cap F \neq \emptyset$);

Algorithm for simulating NFA with ϵ-transitions:

$P = S$;
while (true) {
  do {
    $T = P$;
    for (each $\Delta(q, \epsilon)$ where $q \in P$)
      $P = P \cup \Delta(q, \epsilon)$;
  } **while** ($T \neq P$); /* loop again if P modified */
  if (input empty)
    break;
  scan(c);
  $T = \emptyset$;
  for (each $\Delta(q, c)$ where $q \in P$)
    $T = T \cup \Delta(q, c)$;
  if ($T \subseteq \emptyset$)
    reject;
  $P = T$;
}
accept iff ($P \cap F \neq \emptyset$);

Data structure for NFA

```
interface Label {
    boolean match(char c);
    boolean epsilon(); // true iff e-transition
}

class Edge {
    Integer src, dst;  // source, destination
    Label label;
}

class NFA {
    Set<Integer> S, F; // start, final states
    Set<Edge> edges;   // digraph
    NFA(String regex) {..}
    boolean accept(String input) {..}
}
```

```
boolean accept(String input) {
  Set<Integer> P = new HashSet<Integer>(S);
  int i = 0;

  while (true) {
    boolean changed;
    do {
      changed = false;
      for (Edge e : edges)
        if (e.label.epsilon() && P.contains(e.src))
          changed = changed || P.add(e.dst);
    } while (changed);

    if (i >= input.length()) break;
    char c = input.charAt(i++);

    Set<Integer> T = new HashSet<Integer>();
    for (Edge e : edges)
      if ((e.label.match(c) && P.contains(e.src))
        T.add(e.dst);

    if (T.isEmpty()) return false;
    P = T;
  }

  for (Integer q : P)
    if (F.contains(q)) return true;
  return false;
}
```