Recall: An NFA is a 5-tuple \((S, \Sigma, \text{move}, s_0, F)\) where

- \(S\) is a finite set (of states)
- \(\Sigma\) is an alphabet (the input alphabet)
- \(\text{move}\) is a function from \(S \times (\Sigma \cup \{\epsilon\})\) to \(2^S\) (the powerset of \(S\), that is, the set of all subsets of \(S\))
- \(s_0 \in S\) (the start state)
- \(F \subseteq S\) (the set of final, or accepting, states)

An NFA \(M\) accepts a string \(x\) if there is a path in the transition graph of \(M\) from the start state to an accepting state, such that the labels along this path “spell out” \(x\).

An NFA \(M\) accepts or recognizes, a language \(L\) if it accepts all, and only, strings from \(L\).

A deterministic finite automaton is an NFA in which

- no state has an \(\epsilon\)-transition (that is, in the transition graph, no node has an outgoing edge labeled \(\epsilon\))
- for each state \(s\) and input symbol \(a\) there is at most one outgoing edge labeled \(a\) (in the transition graph).

**From regular expression to NFA**

Algorithm is syntax-directed: follows recursive definition of regular expressions.

Recall: Simultaneous recursive definition of regular expressions and regular languages over alphabet \(\Sigma\):

1. \(\epsilon\) is a regular expression denoting the regular language \(\{\epsilon\}\).
2. For every \(a \in \Sigma\), \(a\) is a regular expression denoting the regular language \(\{a\}\).
3. For any two regular expressions \(r\) and \(s\) denoting regular languages \(L_1\) and \(L_2\):
   - \((r)\cup(s)\) is a regular expression denoting the regular language \(L_1 \cup L_2\).
   - \((r)s\) is a regular expression denoting the regular language \(L_1L_2\).
   - \((r)^*\) is a regular expression denoting the regular language \(L_1^*\).
   - \((r)\) is a regular expression denoting the regular language \(L_1\).
1. \( \epsilon \) is a regular expression denoting the regular language \( \{ \epsilon \} \).

2. For every \( a \in \Sigma \), \( a \) is a regular expression denoting the regular language \( \{ a \} \).

3. For any two regular expressions \( r \) and \( s \) denoting regular languages \( L_1 \) and \( L_2 \):
   - \( (r)(s) \) is a regular expression denoting the regular language \( L_1 \cup L_2 \).
   - \( (r)s \) is a regular expression denoting the regular language \( L_1L_2 \).
   - \( r^* \) is a regular expression denoting the regular language \( L_1^* \).
   - \( r \) is a regular expression denoting the regular language \( L_1 \).

   Recall: We can drop many parentheses: the operators \(|\), concatenation and Kleene star appear in order of increasing precedence, and all are left-associative. So, for example, \( (a)(((b)^*)(c)) \) can be written \( a|b^*c \).

We specify NFA’s for \( \epsilon \), and for \( a \) (for all \( a \in \Sigma \)).

Given NFA’s for \( r \) and \( s \), we show how to construct an NFA for
   - \( (r)(s) \)
   - \( (r)s \)
   - \( r^* \)
   - \( r \)

Each NFA we construct will have the following properties:
   - Exactly one final state.
   - Start state and final state are distinct.
   - No edges enter the start state.
   - No edges leave the final state.

These properties simplify the construction.

For \( \epsilon \), construct the NFA

For \( a \in \Sigma \), construct the NFA
Assume that $N(s)$ and $N(t)$ are NFA’s for regular expressions $s$ and $t$:

For $s|t$, construct the composite NFA

Of course $i$ and $f$ are new states — they are not states in either $N(s)$ or $N(t)$. (Similarly, the states of $N(s)$ and $N(t)$ are disjoint.)

The start and accepting states of $N(s)$ and $N(t)$ are not start and accepting states of the new NFA.

Notice that any path from start state $i$ to final state $f$ must pass through either $N(s)$ or $N(t)$ exclusively, and will “spell out” a string accepted by the machine it passes through.

For $st$, construct the composite NFA

Here, the start state $i$ of $N(s)$ becomes the start state of the composite NFA.

Similarly, the final state $f$ of $N(t)$ becomes the final state of the composite NFA.

(Again, of course, the states of $N(s)$ and $N(t)$ are disjoint.)

The final state of $N(s)$ and the initial state of $N(t)$ are merged in the composite NFA, and the resulting state is neither final nor initial. (It was not final in $N(t)$, and not initial in $N(s)$!)

Any path from $i$ to $f$ in the new NFA must pass through $N(s)$ and then $N(t)$, and so will “spell out” a string in $st$. 
For $s^*$, construct the composite NFA

Finally, for $(s)$ we can simply use the NFA $N(s)$.

Example $(a|b)^*abb$

Here $i$ and $f$ are new states. (The start and accepting states of $N(s)$ are not start and accepting states of the new NFA.)

In the composite NFA, there is an $\epsilon$-transition from $i$ to $f$, reflecting the fact that the empty string belongs to $s^*$.

Other paths from $i$ to $f$ pass through $N(s)$ one or more times, and so “spell out” the concatenation of one or more strings from $s$. 


Simulating an NFA

Much as before, given a set $T$ of states, and an input symbol $a$,

$$Dmove(T, a) = \bigcup_{s \in T} move(s, a)$$

$$T := \epsilon\text{-closure}\{s_0\};$$
$$c := \text{nextchar}();$$
while $c \neq \text{eof}$ do begin

$$T := \epsilon\text{-closure}(Dmove(T, c));$$
$$c := \text{nextchar}();$$
end;
if $T \cap F \neq \emptyset$ then

return “yes”
else return “no”

So, essentially, this algorithm performs the relevant portion of the subset construction at runtime.

The text claims that this can be done in time proportional to the number of states in the NFA times the length of the input string, in the special case of the NFA’s we generate from regular expressions.

Time-Space Tradeoffs

To this point, to decide whether a string $x$ belongs to the language of regular expression $r$, we may

- reduce $r$ to an NFA, and simulate the NFA on $x$
  $$O(|r|) \text{ time and } O(|r|) \text{ space for reduction to NFA}$$
  $$O(|r| \cdot |x|) \text{ time for simulation of NFA (depends on special characteristics of NFA obtained by reduction)}$$
- reduce $r$ to an NFA, reduce the NFA to a DFA, and simulate the DFA on $x$
  $$O(|r|) \text{ time and } O(|r|) \text{ space for reduction to NFA}$$
  $$O(2^{|r|}) \text{ time and space for reduction from NFA to DFA}$$
  $$O(|x|) \text{ time for simulation of DFA}$$
For next time

Continue reading: 3.8. Also, subsections of 3.9 on
Minimizing the number of states of a DFA and State
minimization in lexical analyzers