

Finite Automata

Compiler Construction, Lecture 5

K.R. Chowdhary, Professor of CS

July 19, 2019 | Fall 2019

- A finite automaton is a **recognizer**: it only answers “yes” or “no” for a string
- Both kinds recognize exactly the **regular languages**
- Represented by a transition graph

Formal definition

$$M = (Q, \Sigma, \delta, s, F)$$

Q : states, Σ : alphabet, s : start state,
 $F \subseteq Q$: final states, δ : transition function

DFA

Exactly one transition per state and symbol.

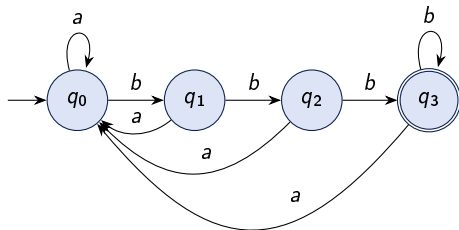
$$\delta : Q \times \Sigma \rightarrow Q$$

NFA

No restriction on edge labels: one symbol may label several edges from a state, and ε is allowed.

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$$

Example: DFA for $(a|b)^*bbb$



Accepts every string of a 's and b 's that ends in bbb .

$M = (\{q_0, \dots, q_3\}, \{a, b\}, \delta, q_0, \{q_3\})$

Algorithm 1: simulate a DFA

```
 $s \leftarrow s_0; c \leftarrow nextChar()$   
while  $c \neq eof$  do  
     $s \leftarrow move(s, c)$   
     $c \leftarrow nextChar()$   
if  $s \in F$  then return "yes"  
else return "no"
```

From Regular Expressions to Automata

- Regular expressions describe lexers, but implementation must simulate an automaton
- An NFA is harder to simulate: ϵ -moves and several moves on one symbol
- So an NFA is converted to an equivalent DFA by the **subset construction**
- NFA states reached by the same symbol are grouped; ϵ -reachable states are merged

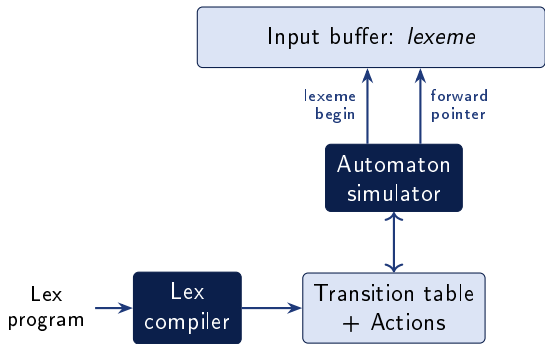
Example: if $s_0 \xrightarrow{\epsilon} s_1$, then
 $\epsilon\text{-closure}(s_0) = \{s_0, s_1\} = S_0$.

Algorithm 2: simulate an NFA

```
 $S \leftarrow \epsilon\text{-closure}(\{s_0\})$   
 $c \leftarrow \text{nextChar}()$   
while  $c \neq \text{eof}$  do  
     $S \leftarrow \epsilon\text{-closure}(\text{move}(S, c))$   
     $c \leftarrow \text{nextChar}()$   
if  $S \cap F \neq \emptyset$  then return "yes"  
else return "no"
```

The state set S is uppercase because every NFA move leads to a *set* of states.

Design of a Lexical Analyser

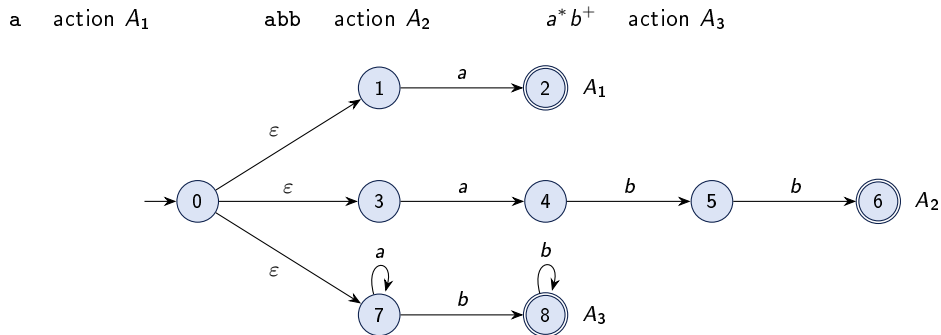


The Lex compiler turns the Lex program into:

1. A transition table for the automaton
2. Functions passed through to the output
3. Actions: code fragments the simulator invokes when a pattern matches

Pipeline: regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ minimized DFA, for efficient code.

Example: NFA for a Set of Lex Patterns



The NFAs $N(p_1), N(p_2), N(p_3)$ are joined under a new start state with ε -edges, then converted to a DFA by subset construction.

Key takeaways

- DFA and NFA recognize the same regular languages
- DFAs are simple to simulate; NFAs are converted by subset construction
- Lex builds one NFA from all patterns, converts it to a DFA, and attaches actions to its final states

Exercises

1. Construct NFAs for:
 - ▶ $(a|b)^*b(a|b)$
 - ▶ $(a|b)^*(a|b)a(a|b)$
 - ▶ $(a|b)(a|b)a(a|b)^*$
2. Convert the NFAs into DFAs
3. Minimize those DFAs