Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

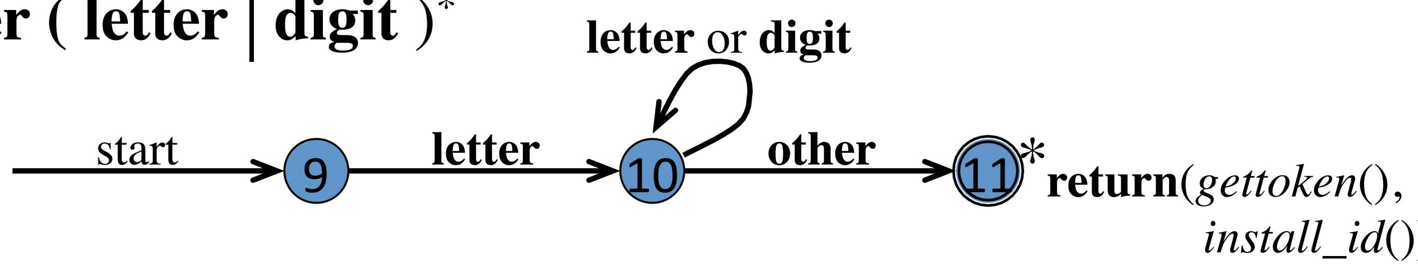
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 6

- Towards Generation of Lexical Analyzers
  - Finite state automata (FSA)
  - From Regular Expressions to FSA
  - The Lex-Flex lexical analyzer generator

- We have seen that:
  - Tokens are defined with regular expressions
  - RE → Transition diagrams → code, by hand!!!
- Example:

$$\textbf{id} \rightarrow \textbf{letter} \; (\; \textbf{letter} \; | \; \textbf{digit} \;)^*$$

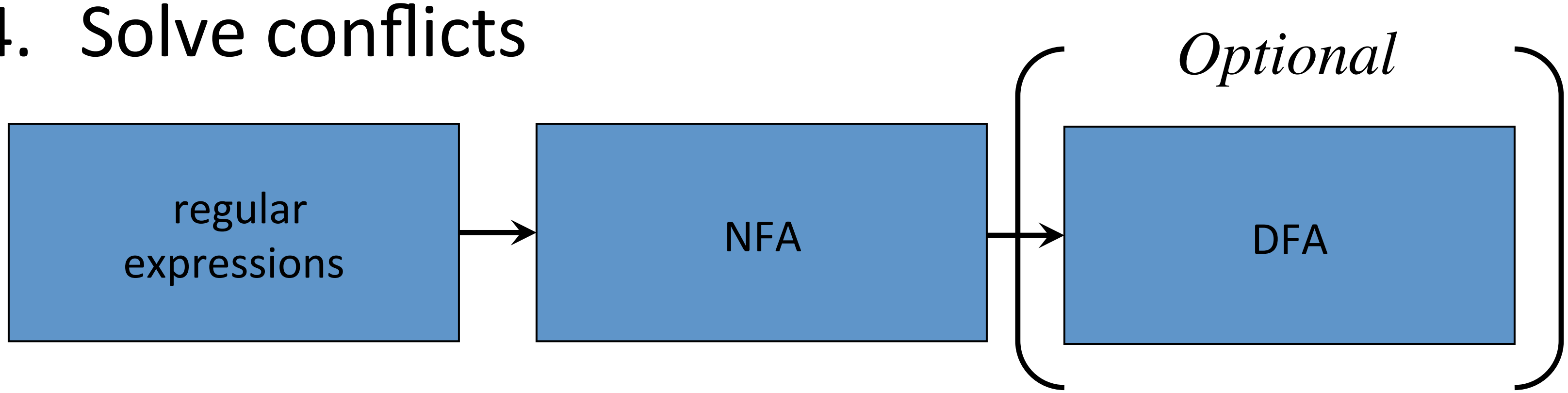



```
…
  case 9: c = nextchar();
    if (isletter(c)) state = 10;
    else state = fail();
    break;
  case 10: c = nextchar();
    if (isletter(c)) state = 10;
    else if (isdigit(c)) state = 10;
    else state = 11;
    break;
  …
```

We present a more systematic and formalized approach

Design of a Lexical Analyzer Generator

1. From the RE of each token build an NFA (non-deterministic finite automaton) that accepts the same regular language
2. Combine the NFAs into a single one
3. Either
   1. Simulate directly the NFA, or
   2. Determinize the NFA and simulate the resulting DFA (deterministic FA)
4. Solve conflicts



Non-deterministic Finite Automata

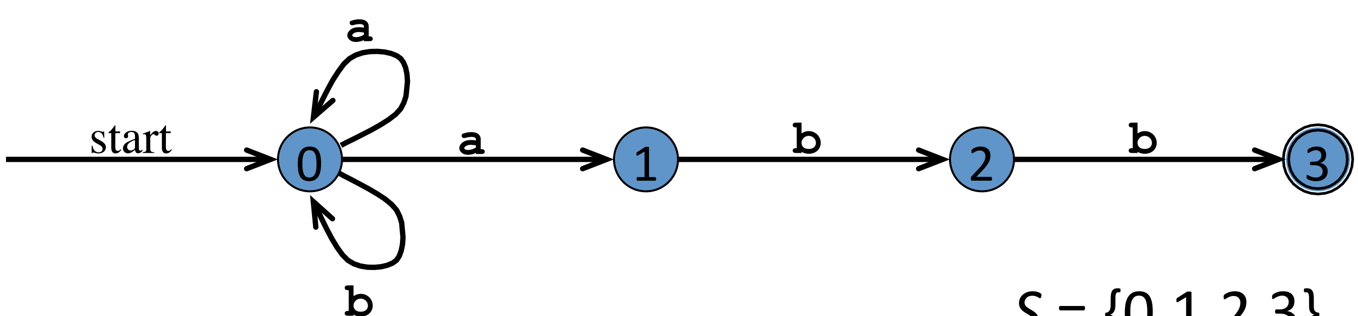
- An NFA is a 5-tuple $(S, \Sigma, \delta, s_0, F)$ where
  - S is a finite set of *states*
  - Σ is a finite set of symbols, the *alphabet*
  - δ is a *mapping* from $S \times (\Sigma \cup \{\varepsilon\})$ to a set of states

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

  - $s_0 \in S$ is the *start state*
  - $F \subseteq S$ is the set of *accepting* (or *final) states*

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$S = \{0,1,2,3\}$
$\Sigma = \{\mathbf{a},\mathbf{b}\}$
$s_0 = 0$
$F = \{3\}$

Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0,\mathtt{a}) = \{0,1\}$
$\delta(0,\mathtt{b}) = \{0\}$
$\delta(1,\mathtt{b}) = \{2\}$
$\delta(2,\mathtt{b}) = \{3\}$

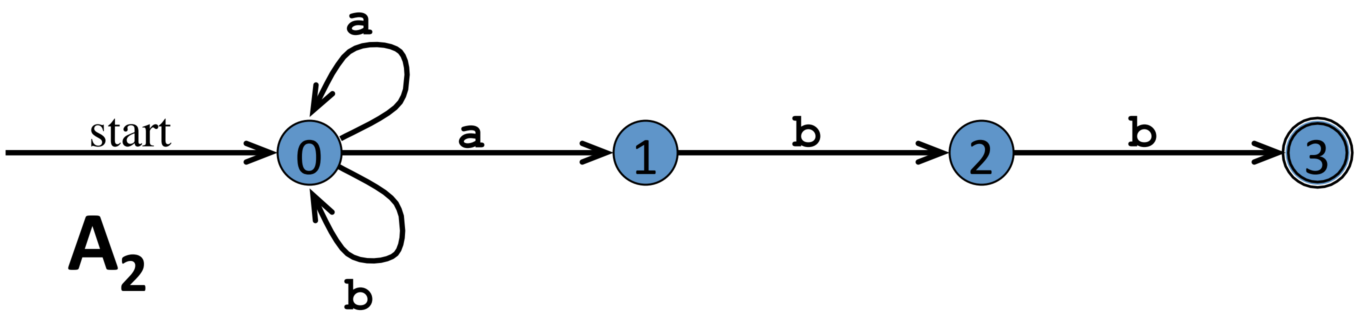
→

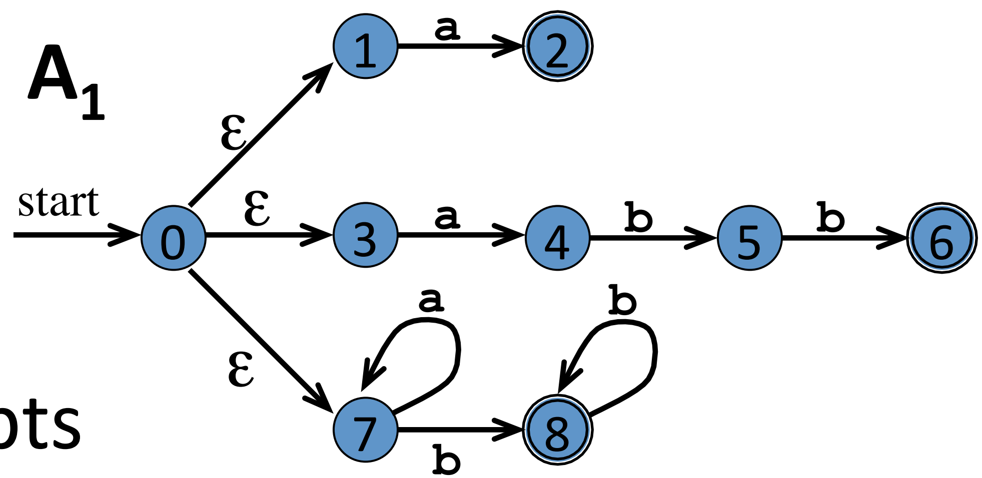
State	*Input* **a**	*Input* **b**
0	{0, 1}	{0}
1		{2}
2		{3}

The Language Defined by an NFA

- An NFA *accepts* an input string *w* (over Σ) if and only if there is at least one path with edges labeled with symbols from *w* in sequence from the start state to some accepting state in the transition graph
- Note that ε-transitions do not contribute with symbols
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA ***A*** is the set of input strings it accepts, denoted *L(**A**)*

Examples





- Which NFA, if any, accepts
  - **aaabb** ?
  - **ababb** ?
  - **abb** ?
  - **abab** ?
- Which are the languages accepted by $\mathbf{A_1}$ and $\mathbf{A_2}$?

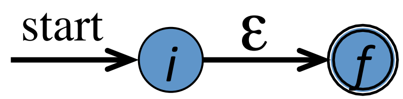
From Regular Expression to NFA: Thompson's Construction

- Given a RE, it builds by *structural induction* a NFA that:
  - **Accepts exactly the language of the RE**
  - Has a single accepting state
  - Has no transitions to the initial state
  - Has no transitions from the final state

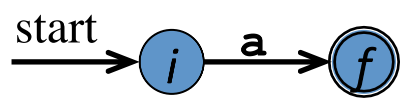
Thompson's Construction

r : RE → $N(r)$: NFA

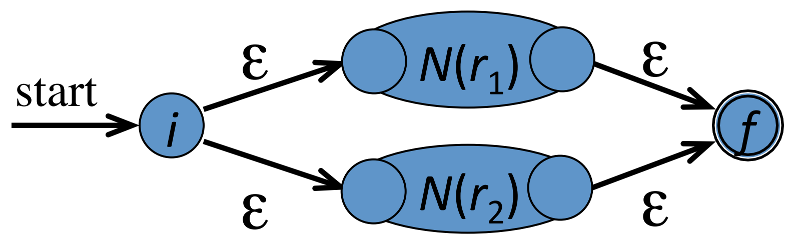
ε



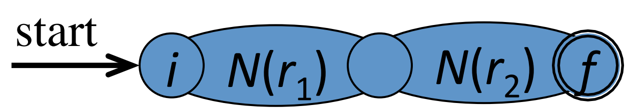
a



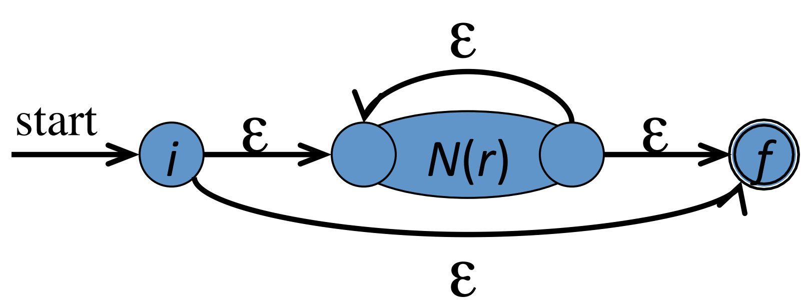
$r_1 \mid r_2$



$r_1 r_2$



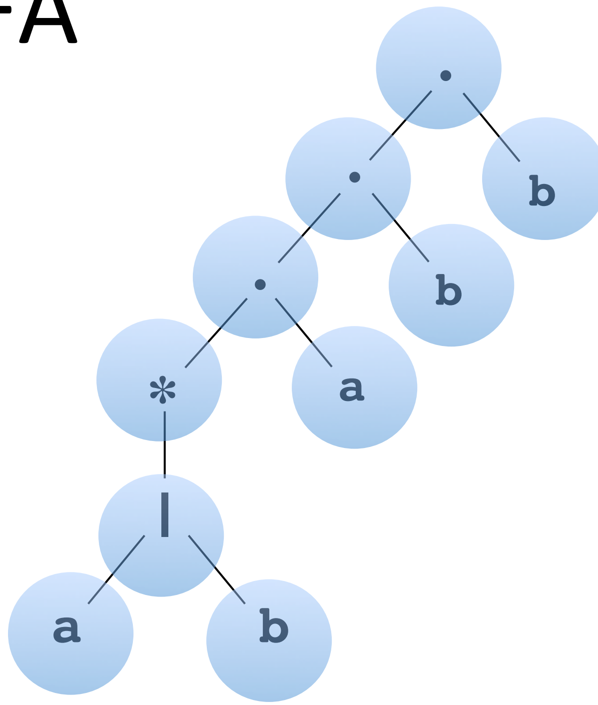
r^*

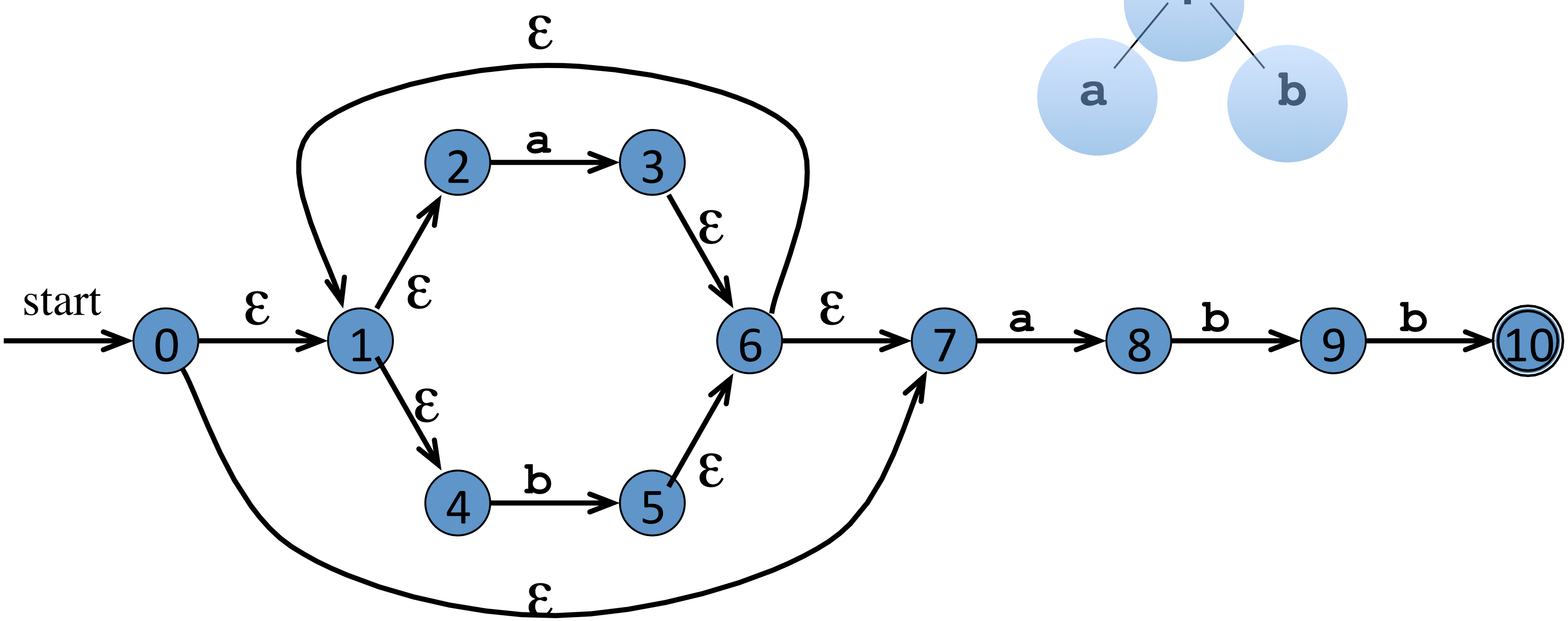


Complexity: linear in the size of the RE

An example:
RE → Syntax Tree → NFA

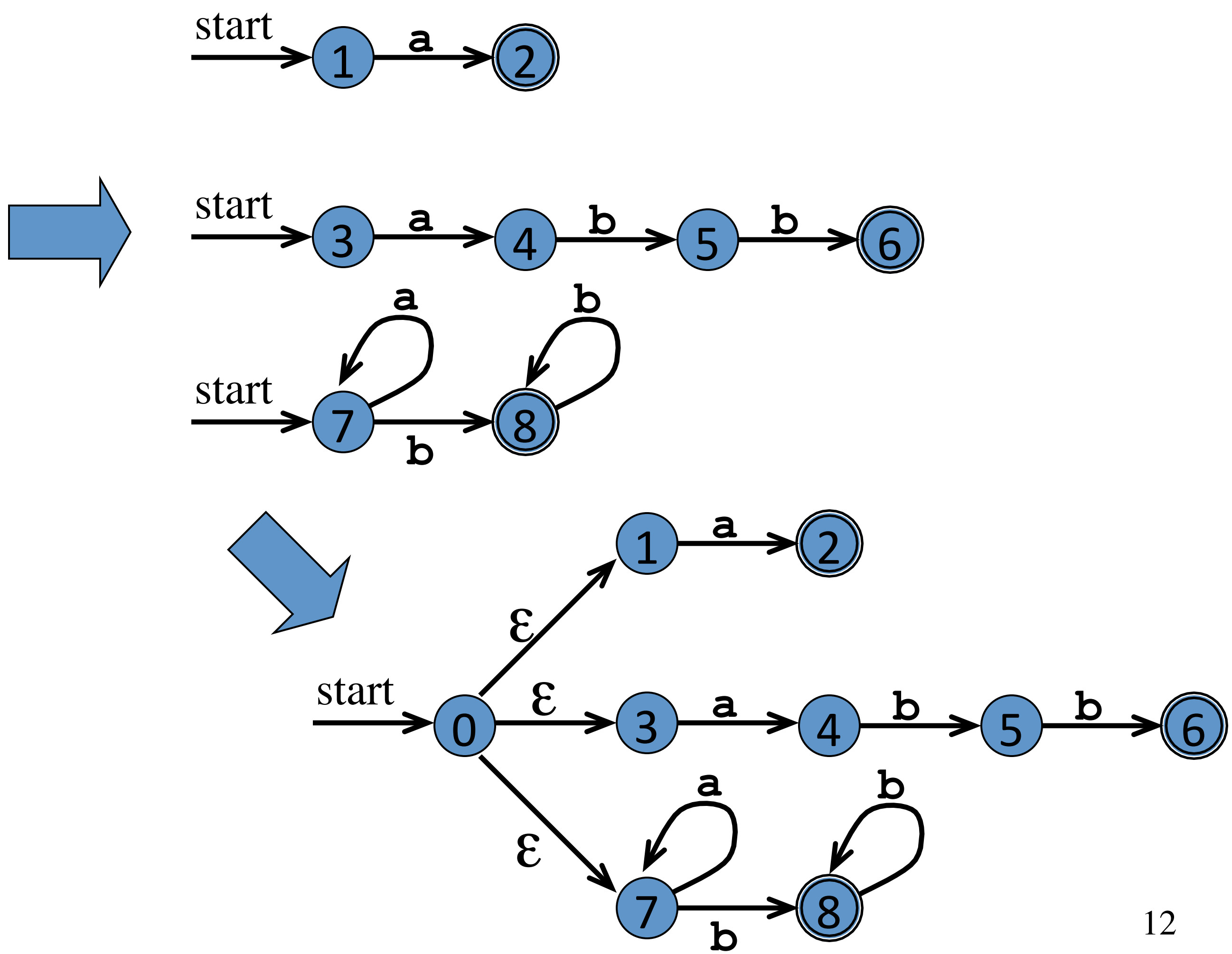
$(\mathbf{a} \mid \mathbf{b})^*\mathbf{abb}$





Combining the NFAs of a Set of Regular Expressions

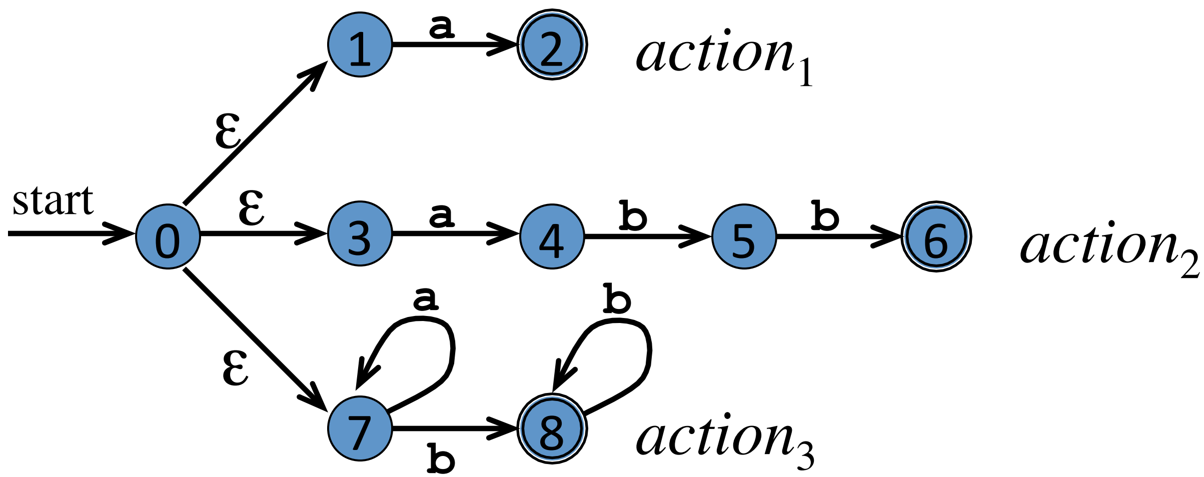
a { *action*$_1$ }
abb { *action*$_2$ }
a*b+ { *action*$_3$ }

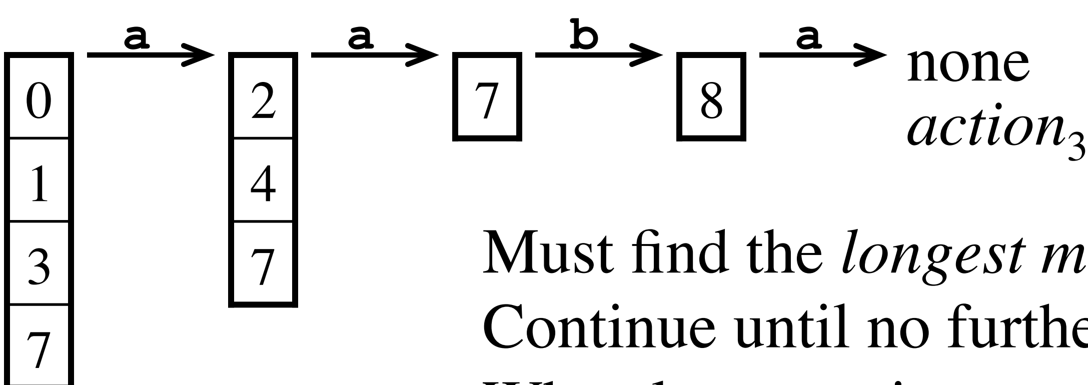


Simulating the Combined NFA

- Given an input string **w**, we look for a prefix accepted by the NFA, i.e. that is the lexeme of a token
  - We start with the set of states reachable by *start* with ε-transitions
  - For each symbol we collect all states to which we can move from the current states
- Complexity: linear in
  (length of **w**) * (number of states),
  using efficient representation of set of states
- Conflicts: several prefixes of **w** can be legal lexemes

Simulating the Combined NFA Example 1

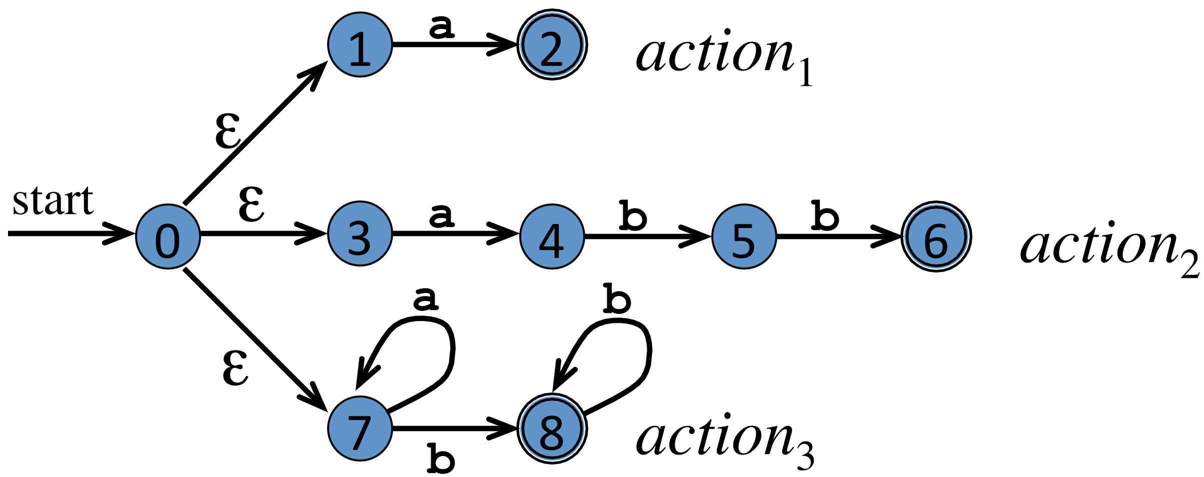


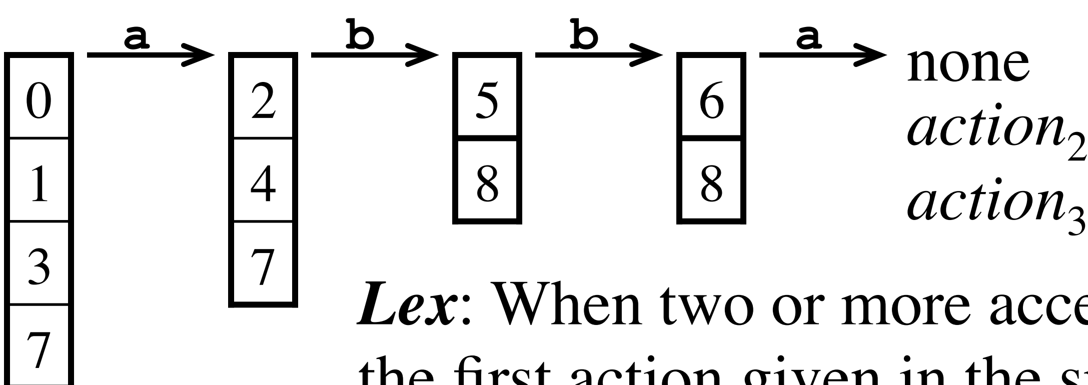


Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

Conflict resolution I

Simulating the Combined NFA Example 2





Conflict resolution II

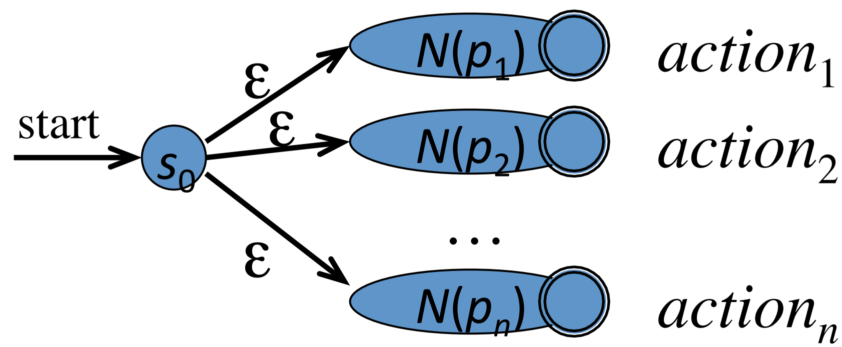
Lex: When two or more accepting states are reached, the first action given in the specification is executed

Design of a Lexical Analyzer Generator: RE to NFA to DFA

Specification with regular expressions

$p_1 \quad \{\ action_1\ \}$
$p_2 \quad \{\ action_2\ \}$
$\ldots$
$p_n \quad \{\ action_n\ \}$

NFA



Subset construction

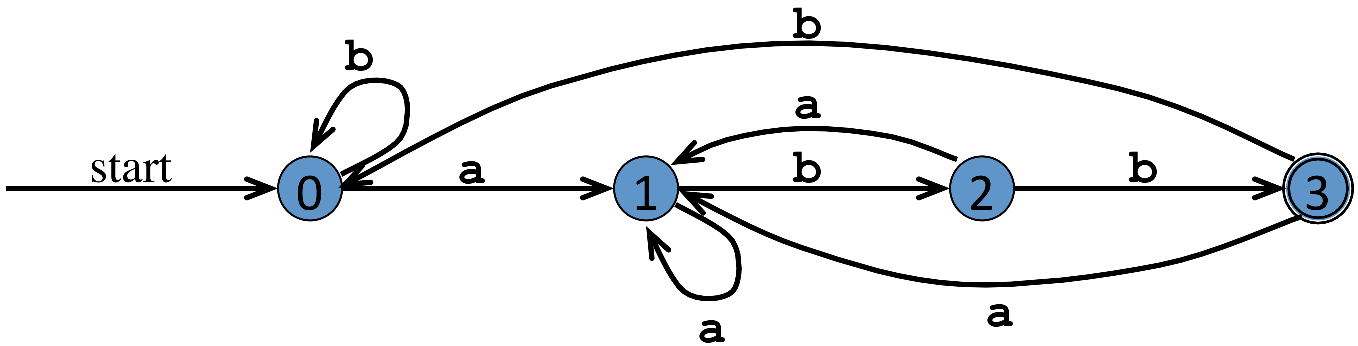
DFA

- Simulating the DFA is more efficient, but
- The size of the DFA could be exponential w.r.t. the NFA

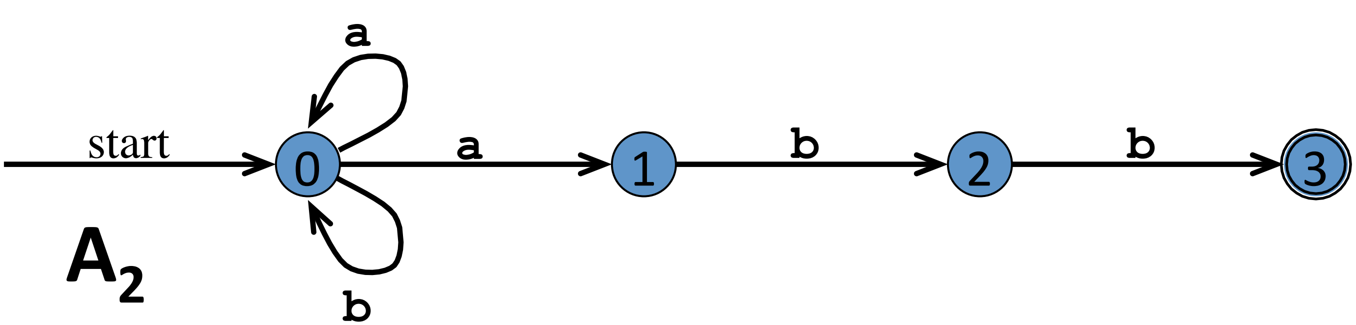
Deterministic Finite Automata

- A deterministic finite automaton is a special case of an NFA
  - No state has an ε-transition
  - For each state *s* and input symbol a there is **at most one** edge labeled a leaving *s*
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple
- Alternative definition:
  - For each state *s* and input symbol a there is **exactly one** edge labeled a leaving *s*
  - Easily shown to be equivalent (sink state…)

Example DFA



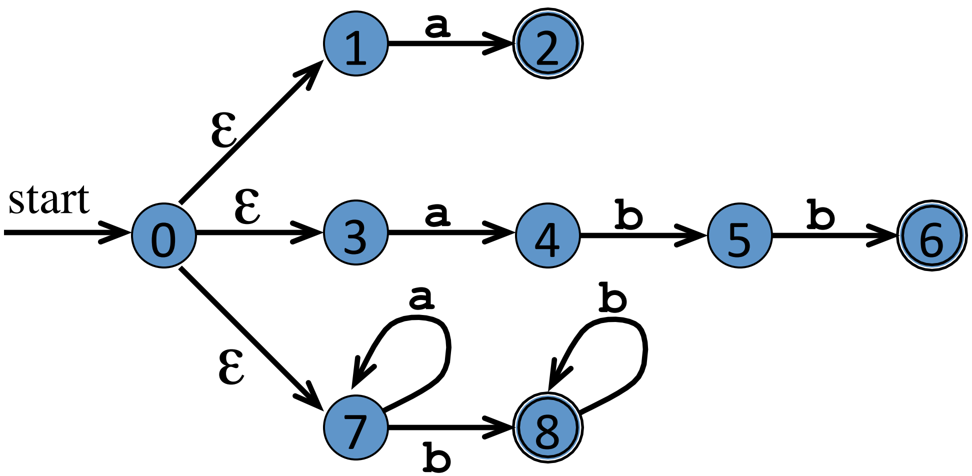
A DFA that accepts the same language of A_2, $(\mathbf{a}\,|\,\mathbf{b})^*\mathbf{abb}$



Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:
  - $\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \rightarrow_\varepsilon \ldots \rightarrow_\varepsilon t\}$
  - $\varepsilon\text{-}closure(T) = \cup_{s \in T}\, \varepsilon\text{-}closure(s)$
  - $move(T, a) = \{t \mid s \rightarrow_a t \text{ and } s \in T\}$
- The algorithm produces:
  - *Dstates* is the set of states of the new DFA consisting of sets of states of the NFA
  - *Dtran* is the transition table of the new DFA

ε-*closure* and *move* Examples



ε-*closure*({0}) = {0,1,3,7}

move({0,1,3,7},**a**) = {2,4,7}

ε-*closure*({2,4,7}) = {2,4,7}

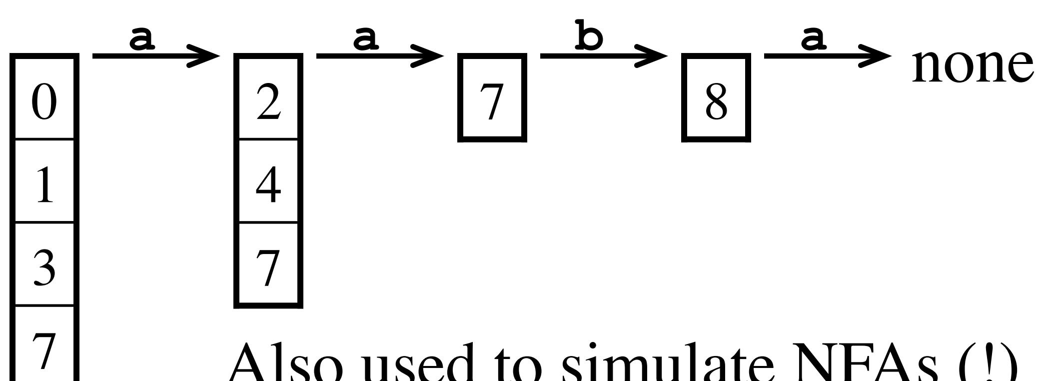
move({2,4,7},**a**) = {7}

ε-*closure*({7}) = {7}

move({7},**b**) = {8}

ε-*closure*({8}) = {8}

move({8},**a**) = ∅



Also used to simulate NFAs (!)

Simulating an NFA using ε-*closure* and *move*

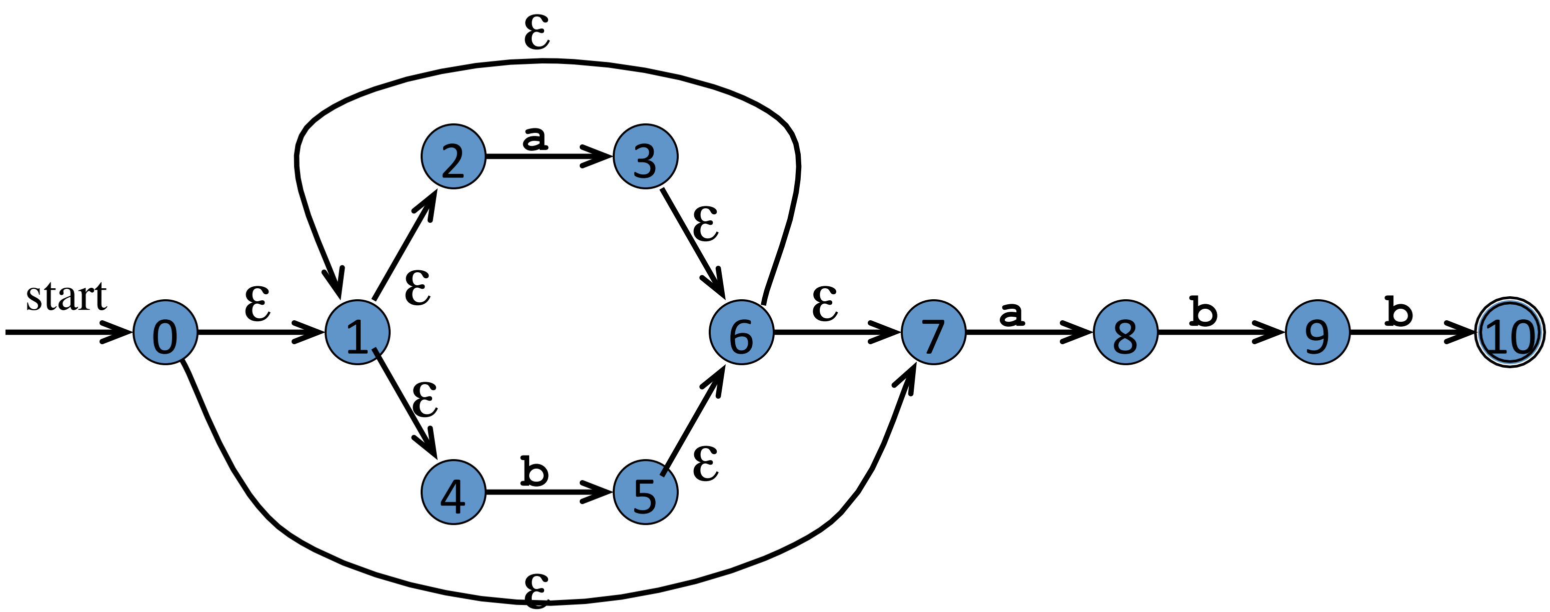
$S := \varepsilon\text{-}closure(\{s_0\})$
$S_{prev} := \emptyset$
$a := nextchar()$
while $S \neq \emptyset$ **do**
    $S_{prev} := S$
    $S := \varepsilon\text{-}closure(move(S,a))$
    $a := nextchar()$
end do
if $S_{prev} \cap F \neq \emptyset$ **then**
    execute *action in* S_{prev}
    return "yes"
else **return** "no"

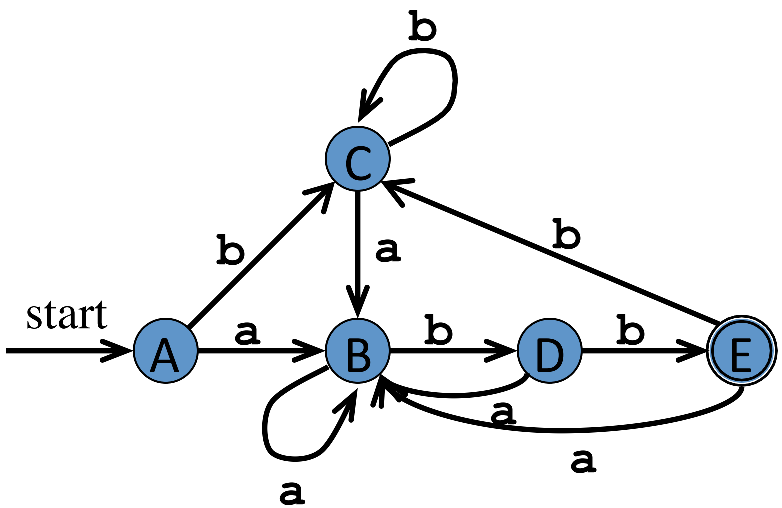
The Subset Construction Algorithm: from a NFA to an equivalent DFA

- Initially, ε-*closure*(s_0) is the only state in *Dstates* and it is unmarked

while there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    for each input symbol $a \in \Sigma$ **do**
        U := ε-*closure*(*move*(*T*,*a*))
        if *U* is not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        end if
        Dtran[*T*, *a*] := *U*
    end do
end do

Subset Construction Example 1





Dstates

A = {0,1,2,4,7}

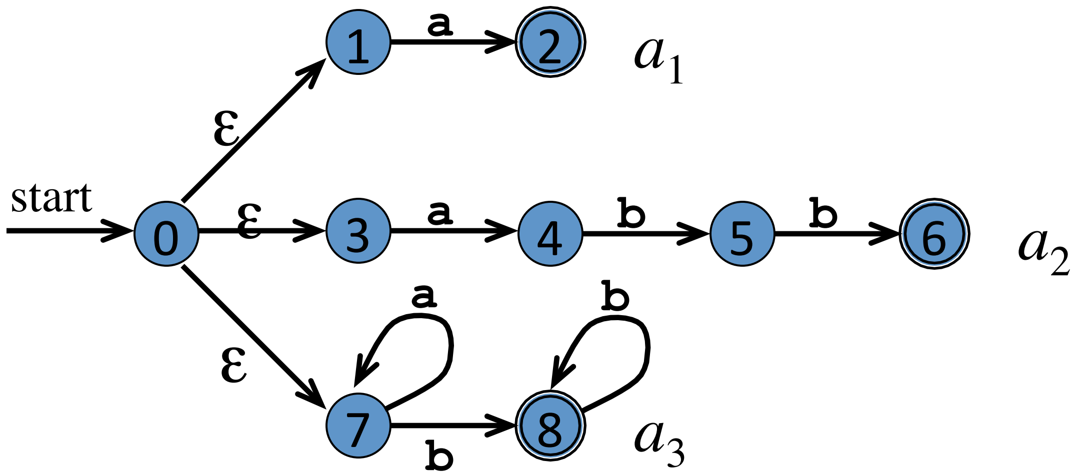
B = {1,2,3,4,6,7,8}

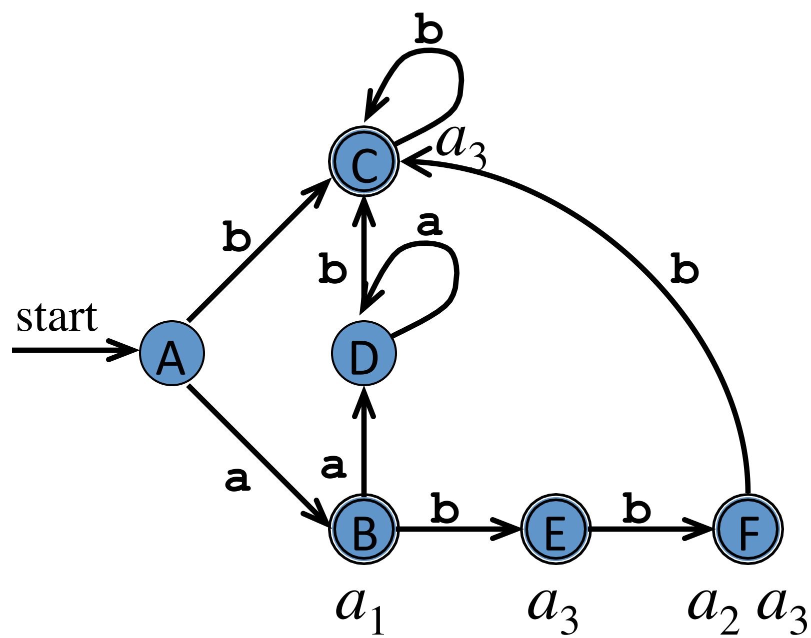
C = {1,2,4,5,6,7}

D = {1,2,4,5,6,7,9}

E = {1,2,4,5,6,7,10}

Subset Construction Example 2





Dstates

A = {0,1,3,7}

B = {2,4,7}

C = {8}

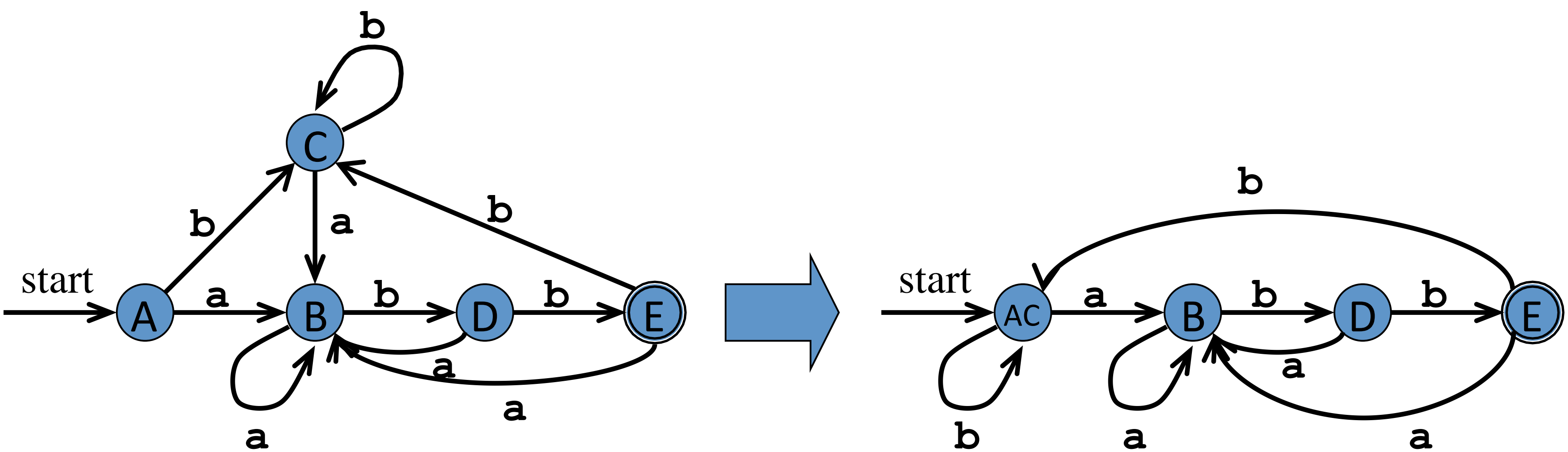
D = {7}

E = {5,8}

F = {6,8}

Minimizing the Number of States of a DFA

- Given a DFA, let us show how to get a DFA which accepts the same regular language with a minimal number of states



On the Minimization Algorithm

- Two states q and q' in a DFA $M = (Q, \Sigma, \delta, q_0, F)$ are ***equivalent*** (or ***indistinguishable***) if for all strings $\boldsymbol{w} \in \Sigma^*$, the states on which $\boldsymbol{w}$ ends on when read from q and q' are both *accept*, or both *non-accept*.
- An automaton is ***irreducible*** if
  - it contains no useless (unreachable) states, and
  - no two distinct states are equivalent
- The **Minimization Algorithm** creates an irreducible automaton accepting the same language
- Partition-refinement: starts with partition of states {Accepting, Non-accepting} and refines it till done

Minimization Algorithm (Partition Refinement) Code

DFA **minimize**(DFA (Q, Σ, d, q_0, F))

  remove any state q unreachable from q_0

  Partition P = $\{F, Q - F\}$

  boolean Consistent = false

  while (Consistent == false) Consistent = true

    for(every Set $S \in P$, char $a \in \Sigma$, Set $T \in P$)

        // collect states of T that reach S using a

      Set temp = $\{q \in T \mid d(q,a) \in S\}$

      if (temp != Ø && temp != T)

        Consistent = false

        $P = (P \setminus \{T\}) \cup \{\text{temp}, T\text{-temp}\}$

  return **defineMinimizor**($(Q, \Sigma, d, q_0, F), P$)

Minimization Algorithm. (Partition Refinement) Code

DFA **defineMinimizor** (DFA $(Q, \Sigma, \delta, q_0, F)$, Partition P)

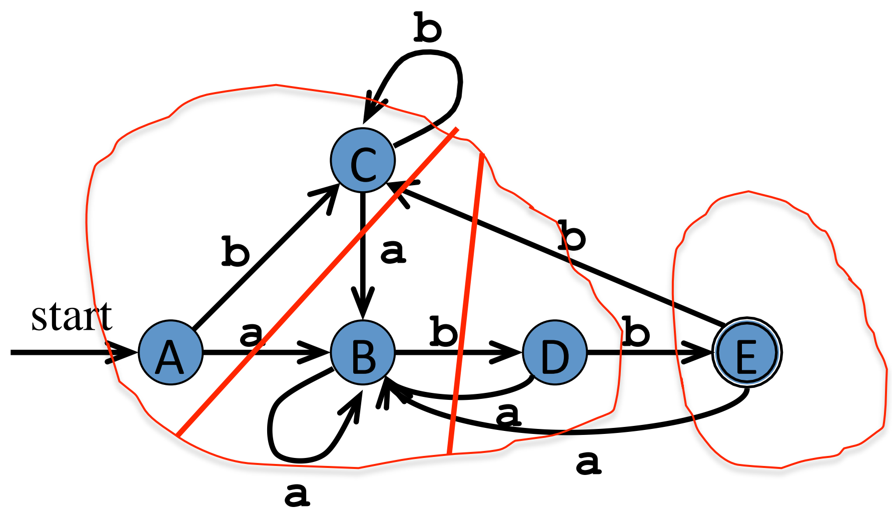
- Set $Q' = P$
- State q'_0 = the set in P which contains q_0
- $F' = \{ S \in P \mid S \subseteq F \}$
- for (each $S \in P$, $a \in \Sigma$)

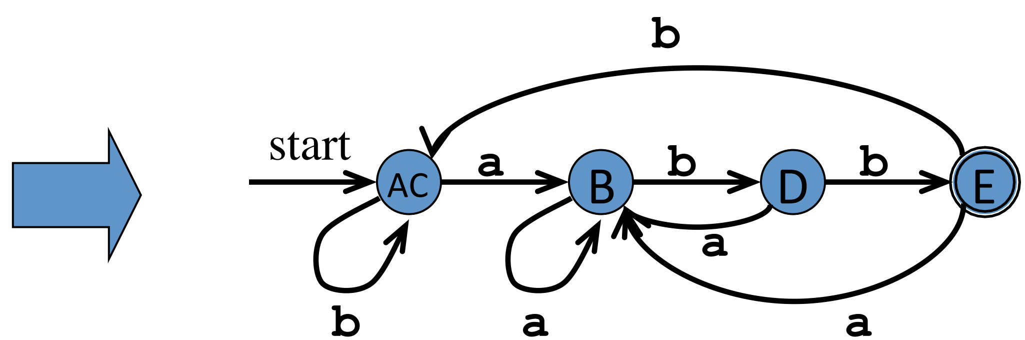
  define $\delta'(S,a)$ = the set $T \in P$ which contains the states $\delta(s,a)$ for each $s \in S$

- return $(Q', \Sigma, \delta', q'_0, F')$

Minimization Algorithm: Example

- P_1 = {{A, B, C, D}, {E}}
  - ({A,B,C,D}, b) not consistent
- P_2 = {{A, B, C}, {D}, {E}}
  - ({A,B,C}, b) not consistent
- P_3 = {{A, C}, {B}, {D}, {E}}
  - Consistent!





Is the constructed automaton minimal?

- The previous algorithm guaranteed to produce an ***irreducible*** DFA. Why should that FA be the *smallest possible* FA for its accepted language?
- THM (Myhill-Nerode): *The minimization algorithm produces the smallest possible automaton for its accepted language.*

Proof of Myhill-Nerode theorem

Proof. Show that any irreducible automaton is the smallest for its accepted language L:

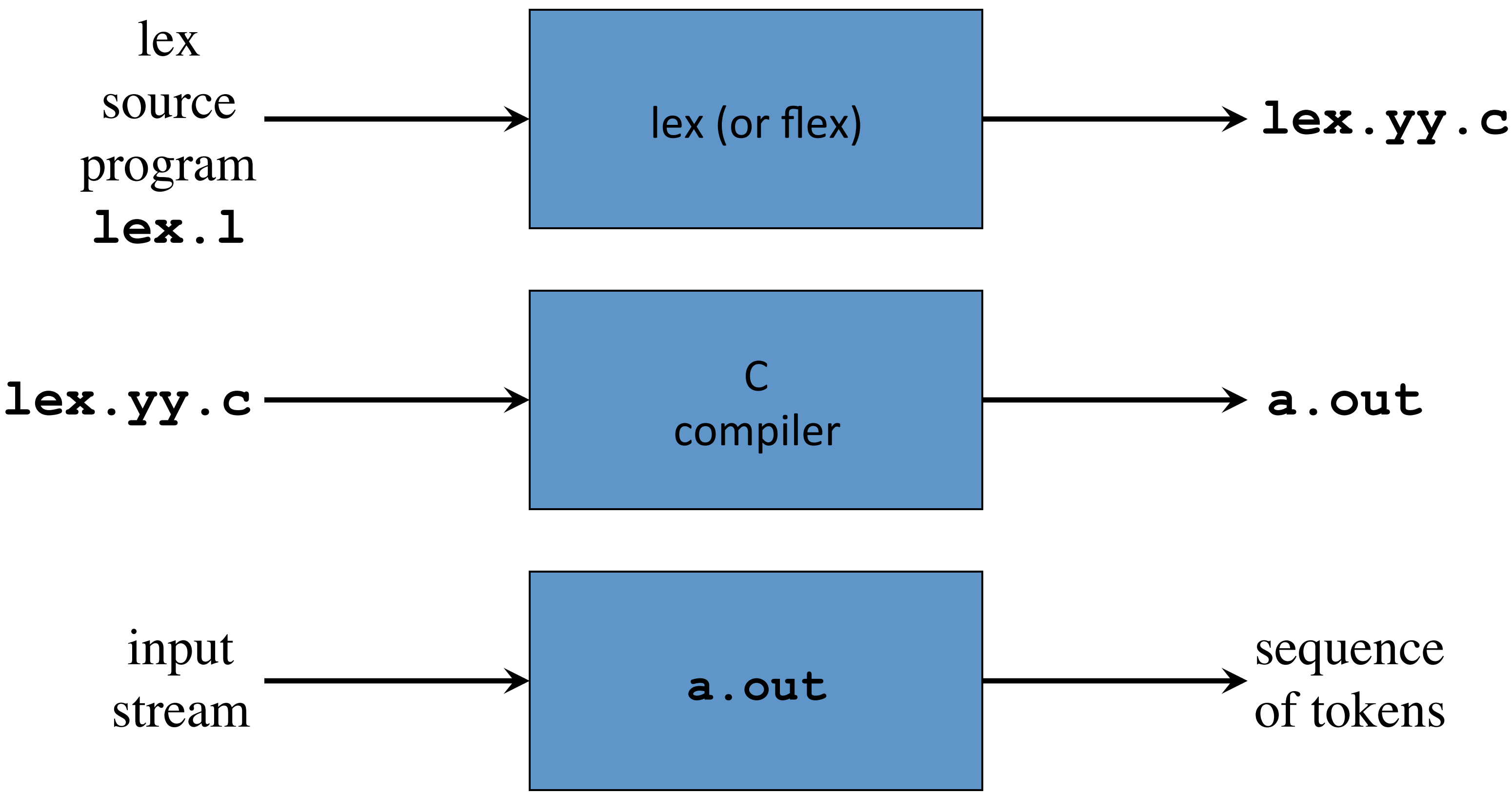
- Two strings $u,v \in \Sigma^*$ are ***indistinguishable*** if for all strings w, $uw \in L \Leftrightarrow vw \in L$.
- Thus if u and v *are* **distinguishable**, their paths from the start state must have different endpoints.
- Therefore the number of states in any DFA for L must be larger than or equal to the number of mutually distinguishable strings for L.
- But in an *irreducible* DFA every state gives rise to another mutually distinguishable string!
- Therefore, any other DFA for the same language must have at least as many states as the irreducible DFA

The Lex and Flex Scanner Generators

- *Lex / flex* are *scanner generators*
- Scanner generators translate regular definitions into C source code for efficient scanning
- Typically used with *parser generators* like Yacc / Bison
- Generated code is easy to integrate in C applications
- Several others: JavaCC, JFlex in Java, etc.
- See

`https://en.wikipedia.org/wiki/Comparison_of_parser_generators`

Creating a Lexical Analyzer with Lex and Flex



Lex Specification

- A *lex specification* consists of three parts:
  regular definitions, C declarations in `%{ %}`
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures
- The *translation rules* are of the form:
  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

Regular Expressions in Lex

x	match the character **x**
\\.	match the character **.**
"*string*"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
r*****	closure (match zero or more occurrences)
r**+**	positive closure (match one or more occurrences)
r**?**	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
r_1**|**r_2	match r_1 or r_2 (union)
(r **)**	grouping
r_1****r_2	match r_1 when followed by r_2
{d**}**	match the regular expression defined by d

Example Lex Specification 1

Translation rules

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    { }
%%
int main()
{ yylex();
}
```

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

Example Lex Specification 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim       [ \t]+
%%
\n          { ch++; wd++; nl++; }
^{delim}    { ch+=yyleng; }
{delim}     { ch+=yyleng; wd++; }
.           { ch++; }
%%
Int main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Example Lex Specification 3

Regular definitions

Translation rules

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
Int main()
{ yylex();
}
```

Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
"<>"       {yylval = NE; return RELOP;}
">"        {yylval = GT; return RELOP;}
">="       {yylval = GE; return RELOP;}
%%
int install_id() { … }
```

Return token to parser

Token attribute

Install **`yytext`** (of length **`yyleng`**) as identifier in symbol table