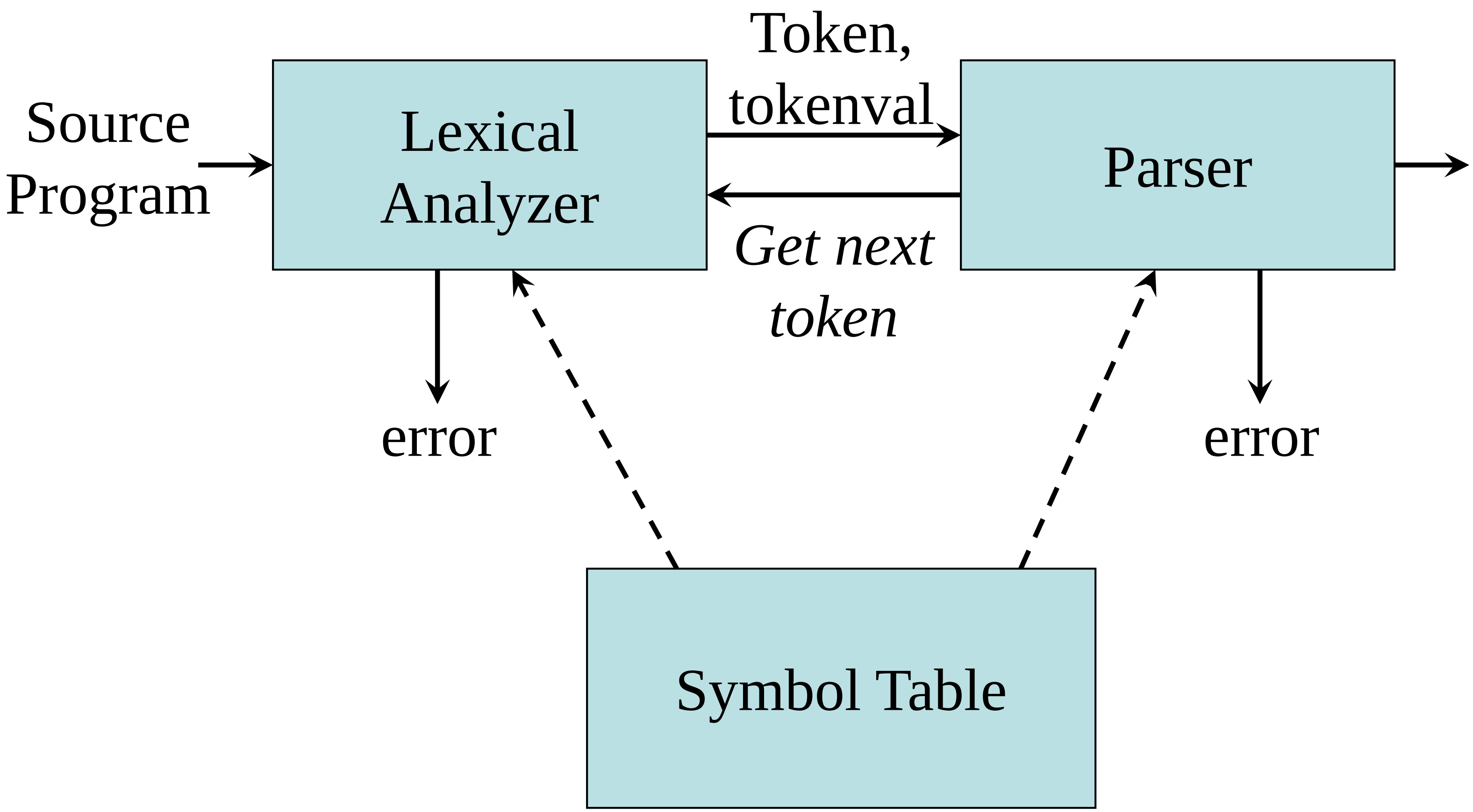
Lexical Analyzer

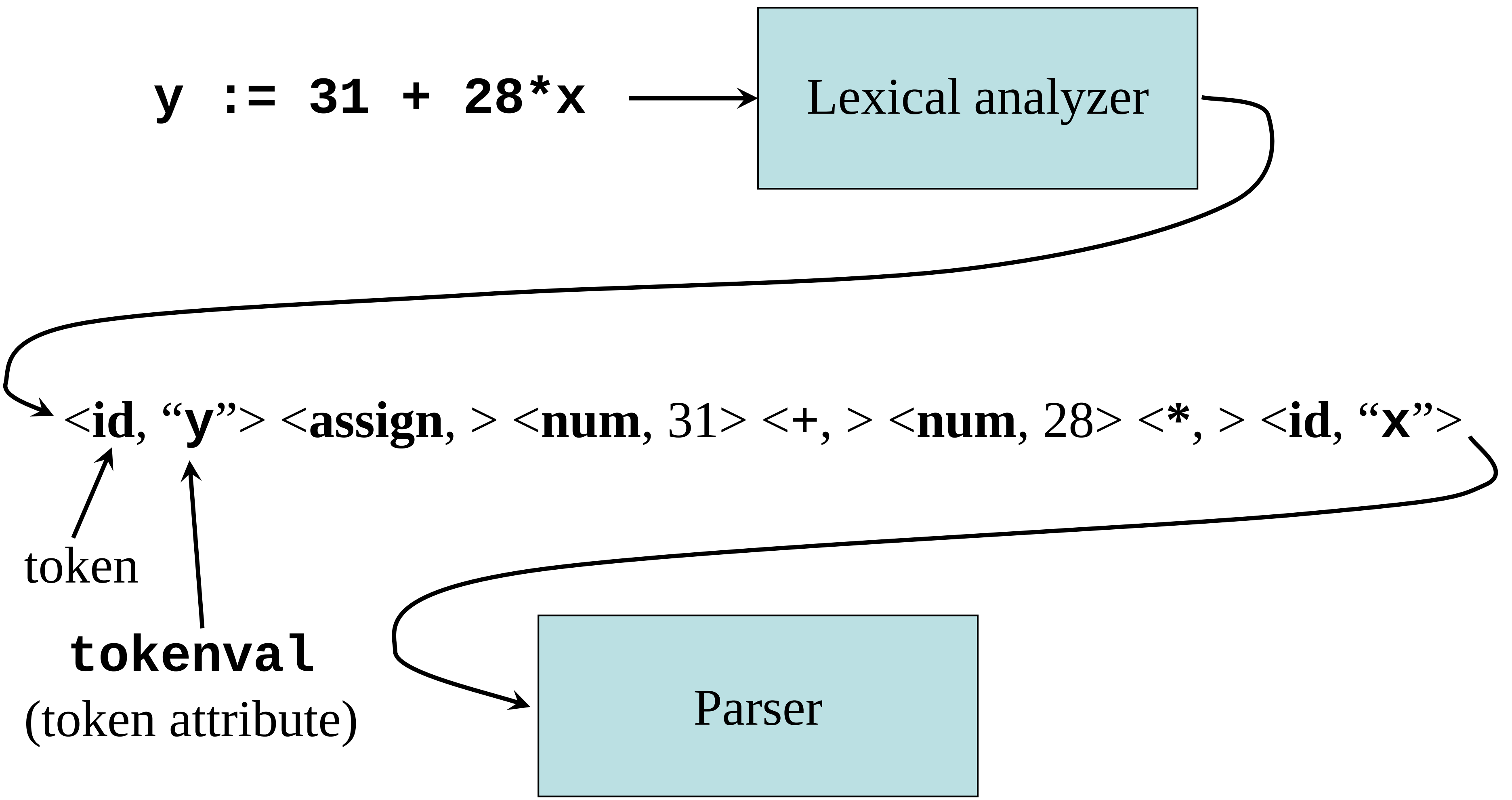
The Reason Why Lexical Analysis is a Separate Phase

- Simplifies the design of the compiler
  - LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be normalized (e.g. trigraphs)

Interaction of the Lexical Analyzer with the Parser



Attributes of Tokens



Tokens, Patterns, and Lexemes

- A *token* is a classification of lexical units
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **`abc`** and **`123`**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"

Specification of Patterns for Tokens: *Definitions*

- An *alphabet* Σ is a finite set of symbols (characters)
- A *string s* is a finite sequence of symbols from Σ
  - $|s|$ denotes the length of string *s*
  - ε denotes the empty string, thus $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet Σ

Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings x and y is denoted by xy
- The *exponentiation* of a string s is defined by

$$s^0 = \varepsilon$$
$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that $s\varepsilon = \varepsilon s = s$

Specification of Patterns for Tokens: *Language Operations*

- *Union*

  $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
- *Concatenation*

  $LM = \{xy \mid x \in L \text{ and } y \in M\}$
- *Exponentiation*

  $L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L$
- *Kleene closure*

  $L^* = \cup_{i=0,\ldots,\infty} L^i$
- *Positive closure*

  $L^+ = \cup_{i=1,\ldots,\infty} L^i$

Specification of Patterns for Tokens: *Regular Expressions*

- Basis symbols:
  - ε is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$
- If r and s are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  - $r \mid s$ is a regular expression denoting $L(r) \cup M(s)$
  - rs is a regular expression denoting $L(r)M(s)$
  - r^* is a regular expression denoting $L(r)^*$
  - (r) is a regular expression denoting $L(r)$
- A language defined by a regular expression is called a *regular set*

Specification of Patterns for Tokens: *Regular Definitions*

- Regular definitions introduce a naming convention:

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  …
  $d_n \rightarrow r_n$

  where each r_i is a regular expression over

  $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

- Any d_j in r_i can be textually substituted in r_i to obtain an equivalent set of definitions

Specification of Patterns for Tokens: *Regular Definitions*

- Example:

$$\begin{aligned} \textbf{letter} &\to \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\ \textbf{digit} &\to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\ \textbf{id} &\to \textbf{letter} \; (\, \textbf{letter} \mid \textbf{digit} \,)^* \end{aligned}$$

- Regular definitions are not recursive:

$\textbf{digits} \to \textbf{digit digits} \mid \textbf{digit}$ *wrong!*

Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

$$r^+ = rr^*$$
$$r? = r \mid \varepsilon$$
$$[\mathbf{a\text{-}z}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \ldots \mid \mathbf{z}$$

- Examples:
  digit $\rightarrow$ [**0-9**]
  num $\rightarrow$ **digit**$^+$ (**.** **digit**$^+$)? (**E** (**+** | **-**)? **digit**$^+$)?

Regular Definitions and Grammars

Grammar

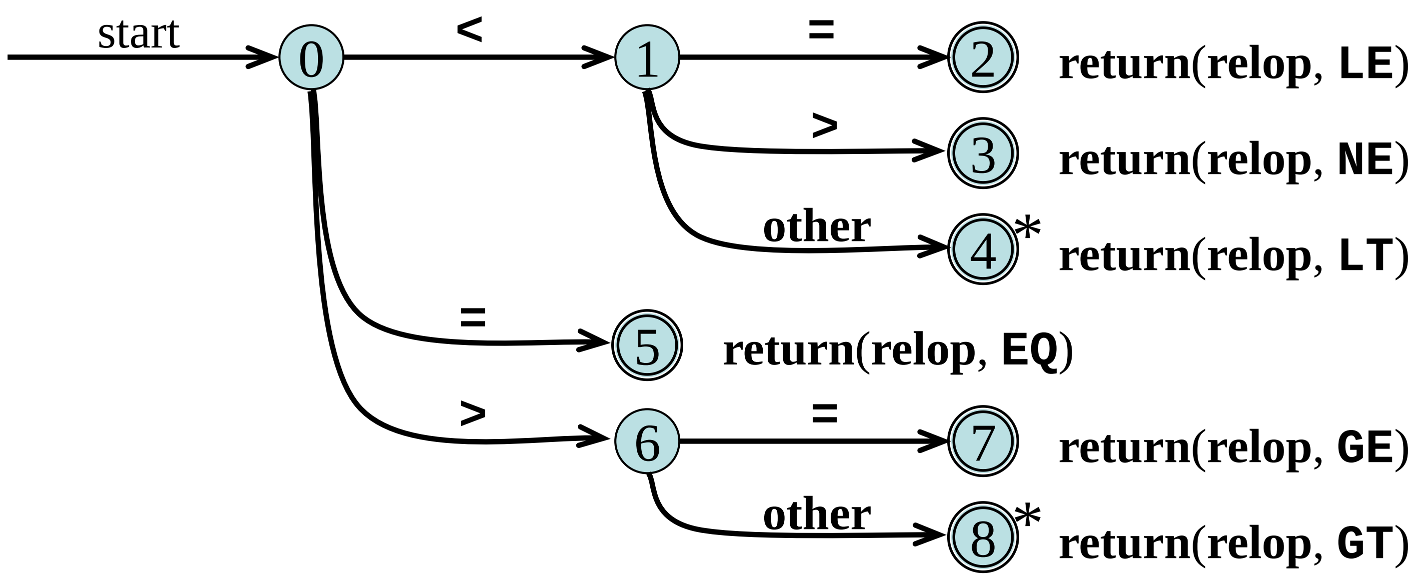
$$
\begin{aligned}
\mathit{stmt} &\rightarrow \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
&\quad\mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
&\quad\mid \varepsilon \\
\mathit{expr} &\rightarrow \mathit{term}\ \textbf{relop}\ \mathit{term} \\
&\quad\mid \mathit{term} \\
\mathit{term} &\rightarrow \textbf{id} \\
&\quad\mid \textbf{num}
\end{aligned}
$$

Regular definitions

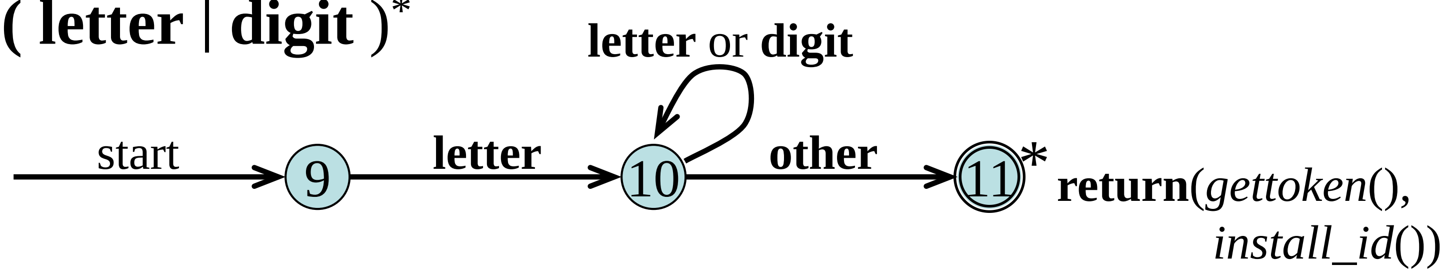
$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \\
\textbf{id} &\rightarrow \textbf{letter}\ (\ \textbf{letter} \mid \textbf{digit}\ )^* \\
\textbf{num} &\rightarrow \textbf{digit}^+\ (.\ \textbf{digit}^+)?\ (\ \textbf{E}\ (+ \mid -)?\ \textbf{digit}^+\ )?
\end{aligned}
$$

Coding Regular Definitions in *Transition Diagrams*

relop $\rightarrow$ **`<` | `<=` | `<>` | `>` | `>=` | `=`**



id $\rightarrow$ **letter (letter | digit)***



Coding Regular Definitions in Transition Diagrams: Code

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
       if (c==blank || c==tab || c==newline) {
         state = 0;
         lexeme_beginning++;
       }
       else if (c=='<') state = 1;
       else if (c=='=') state = 5;
       else if (c=='>') state = 6;
       else state = fail();
       break;
     case 1:
       …
     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;
     …
```

Decides the next start state to check

```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

Limits of Regular Languages

Not all languages are regular

RL's ⊂ CFL's ⊂ CSL's

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ (parenthesis languages)
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language (nor an RE)

But, this is a little subtle. You can construct DFA's for

- Strings with alternating 0's and 1's

  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$

- Strings with and even number of 0's and 1's

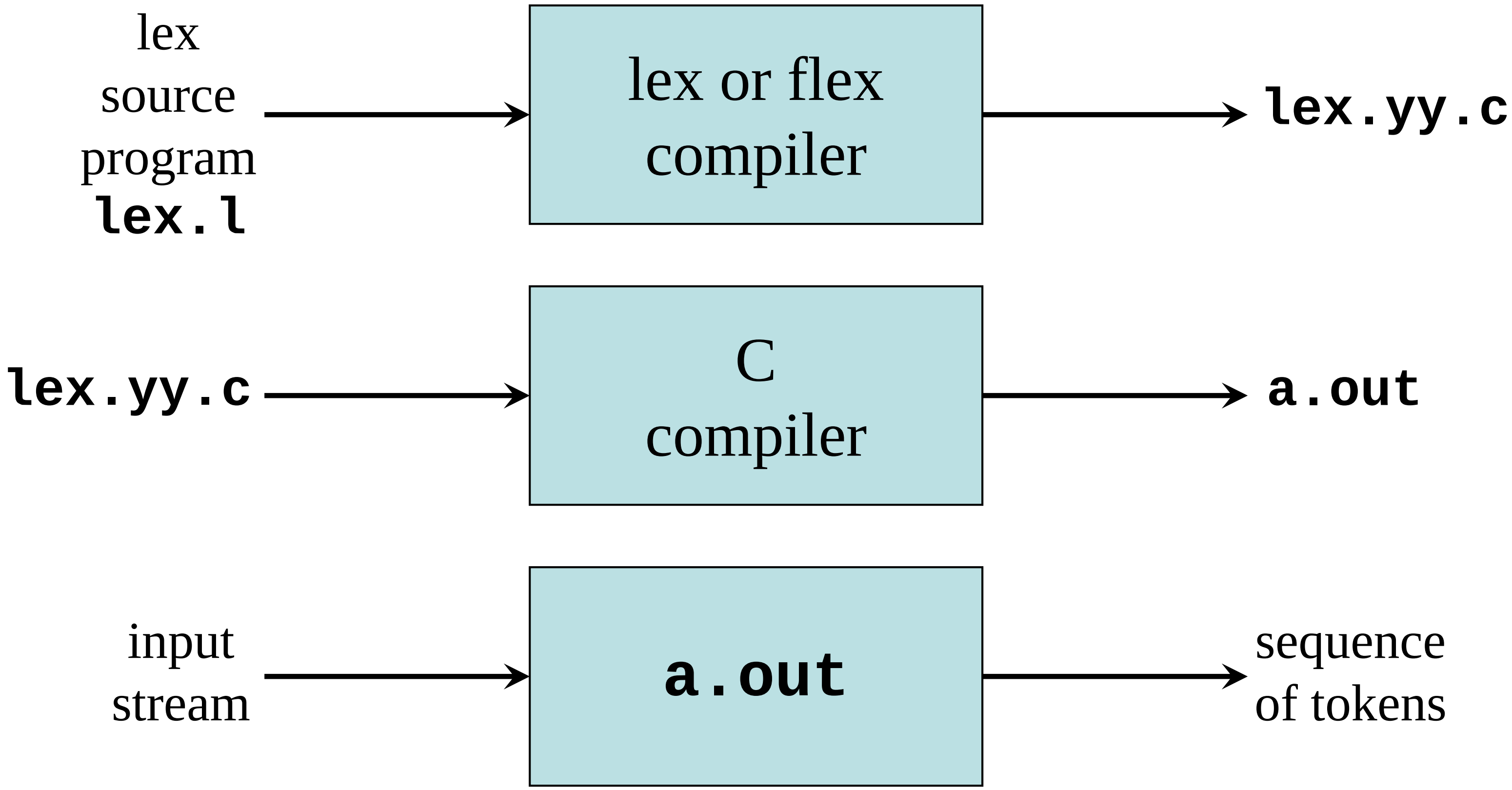
  See Homework 1!

RE's can count bounded sets and bounded differences

The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

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)
$r_1 \vert r_2$	match r_1 or r_2 (union)
`(` r `)`	grouping
$r_1 \backslash r_2$	match r_1 when followed by r_2
`{`d`}`	match the regular expression defined by d

Example Lex Specification 1

Translation rules

Contains the matching lexeme

Invokes the lexical analyzer

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

```
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++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#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); }
%%
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()
…
```

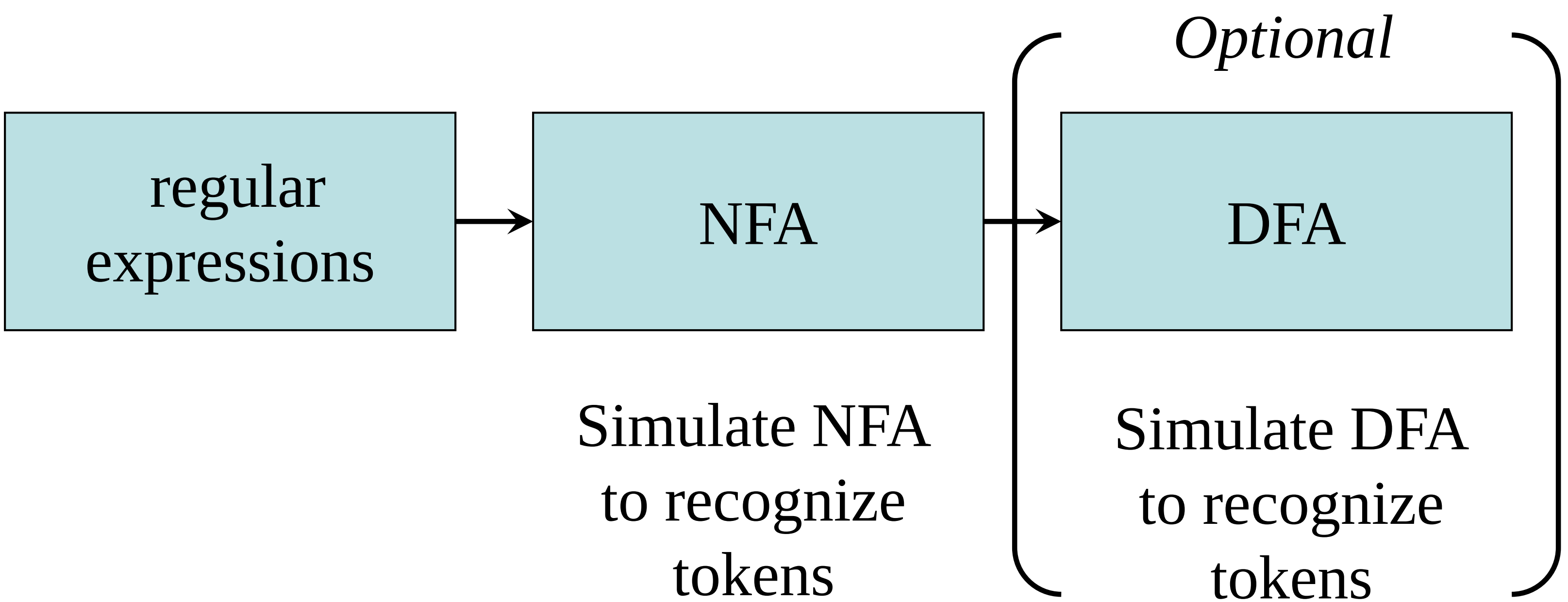
Return token to parser

Token attribute

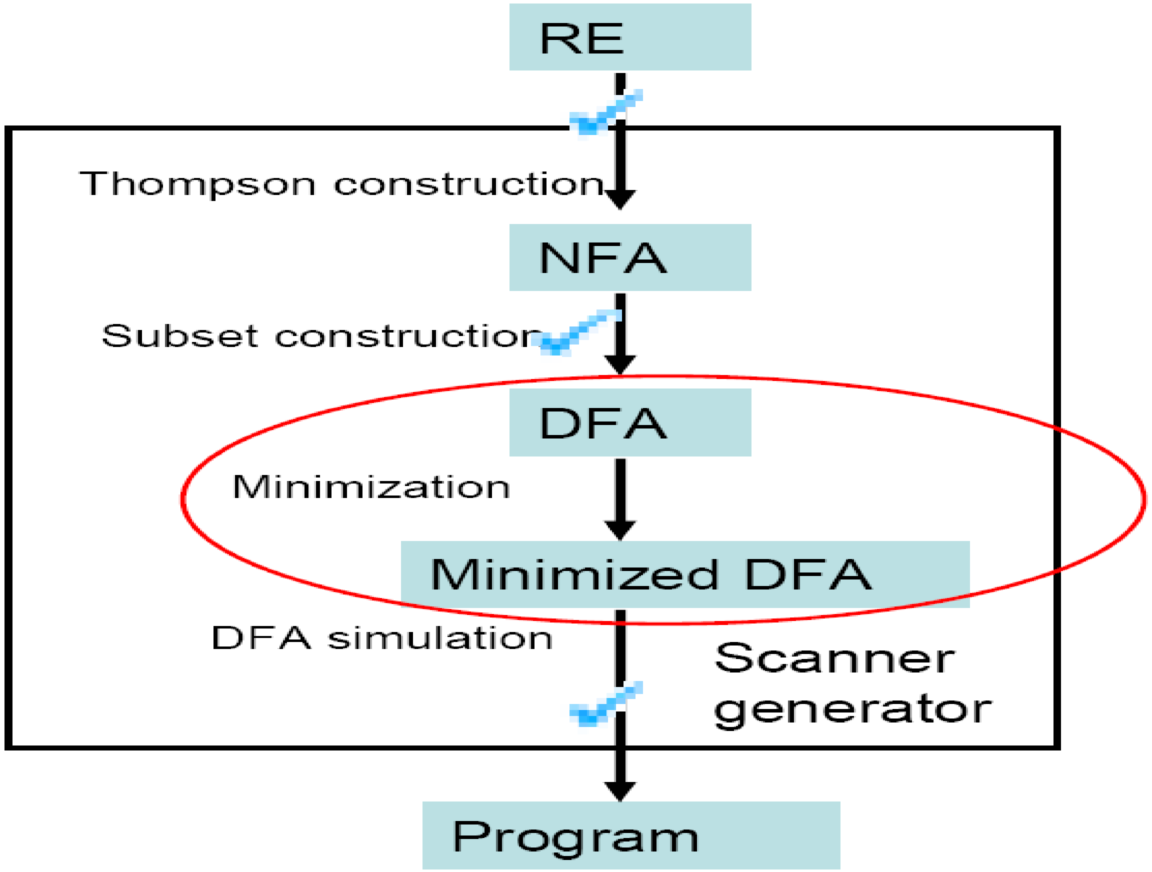
Install **`yytext`** as identifier in symbol table

Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



Lexical Analyzer Generator – Design



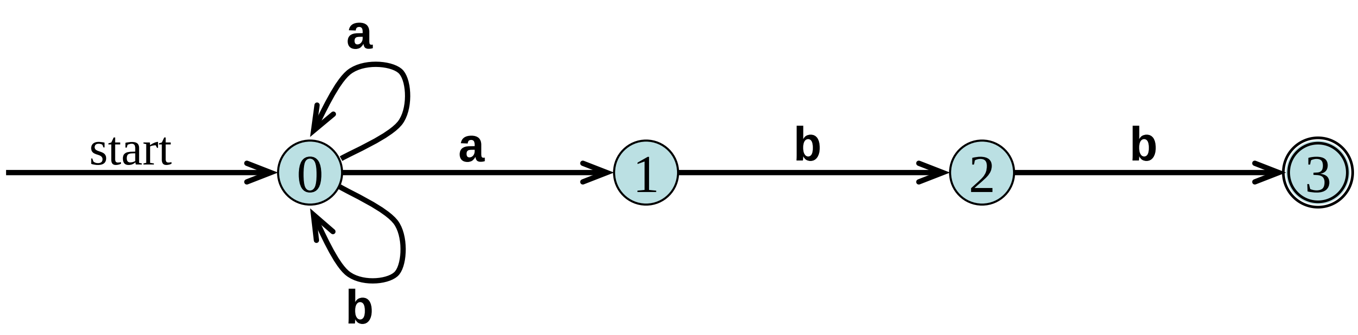
Nondeterministic Finite Automata

- An NFA is a 5-tuple (S, Σ, δ, 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 $S \times \Sigma$ to a set of states
  $s_0 \in S$ 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*

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

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

The Language Defined by an NFA

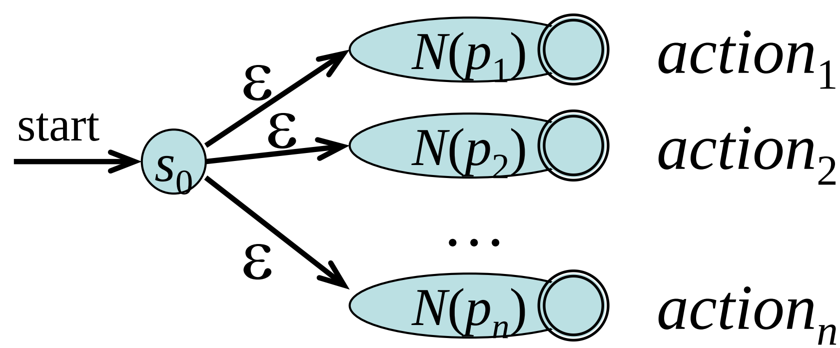
- An NFA *accepts* an input string *x* if and only if there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as (**a**|**b**)***abb** for the example NFA

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

Lex specification with regular expressions

p_1	{ $action_1$ }
p_2	{ $action_2$ }
…	
p_n	{ $action_n$ }

NFA

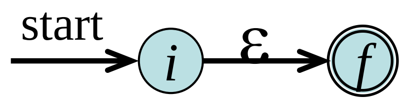


Subset construction

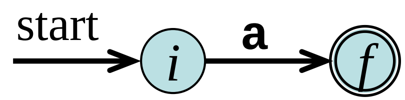
DFA

From Regular Expression to NFA (Thompson's Construction)

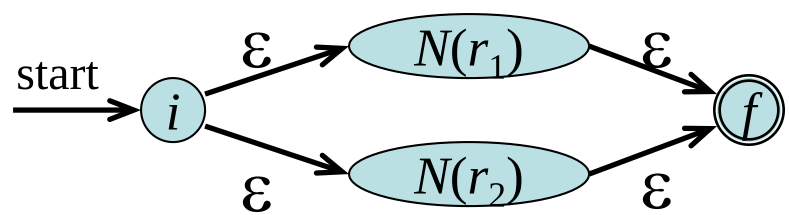
ε



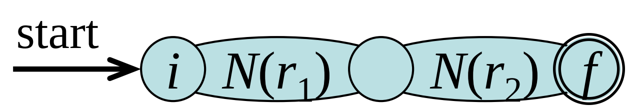
a



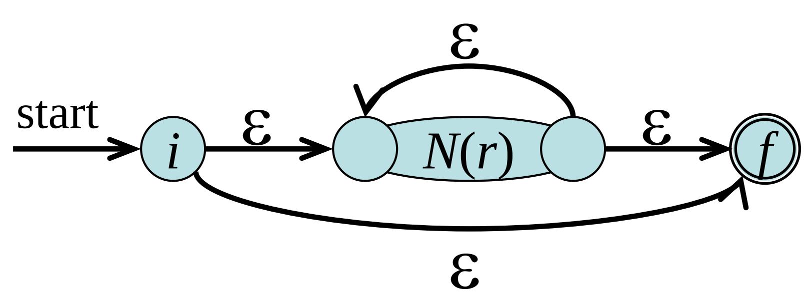
$r_1 \mid r_2$



$r_1 r_2$

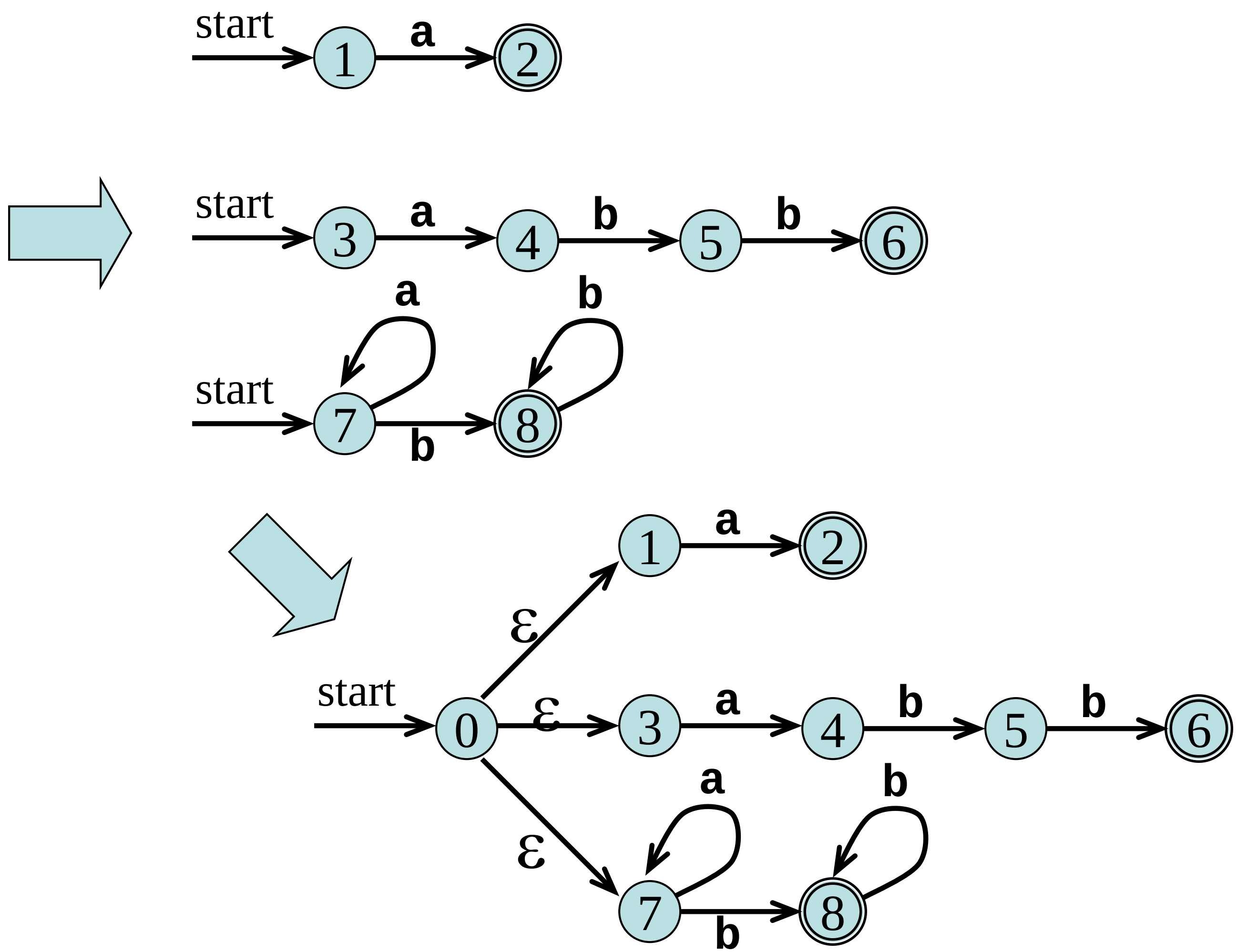


r^*

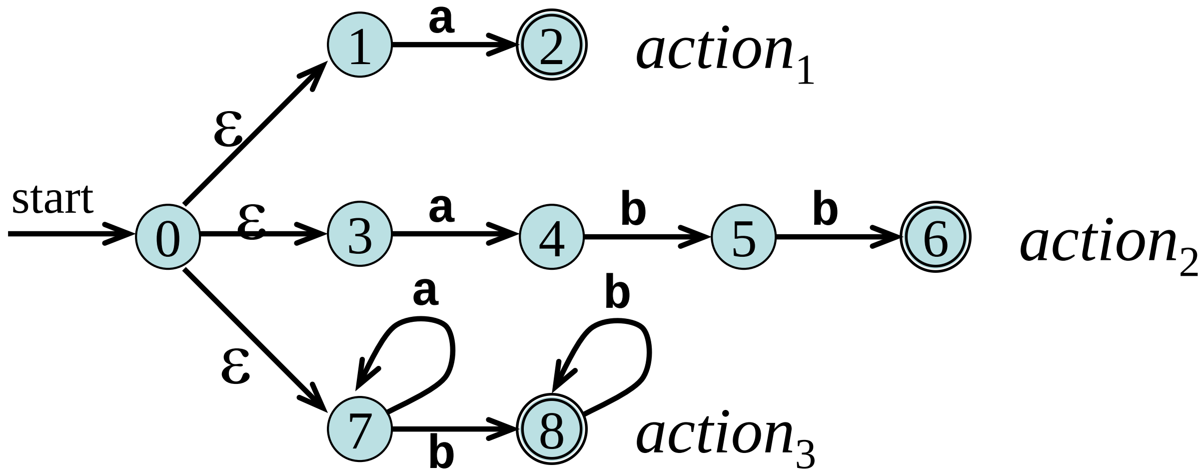


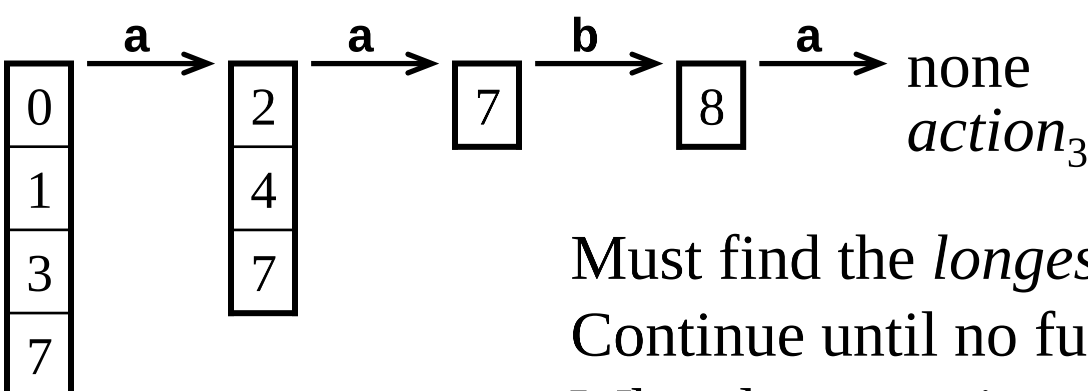
Combining the NFAs of a Set of Regular Expressions

a { $action_1$ }
abb { $action_2$ }
a***b**+ { $action_3$ }



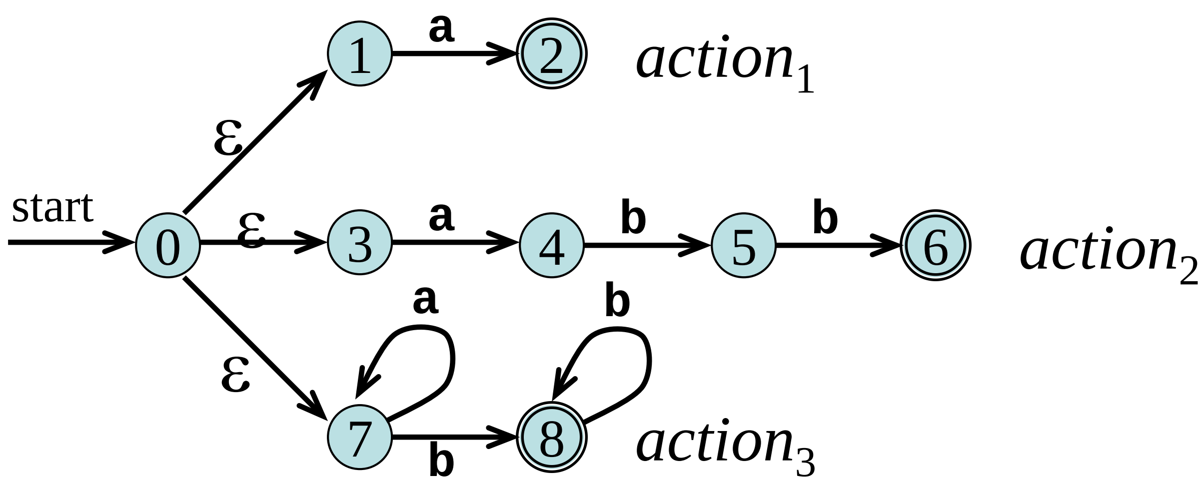
Simulating the Combined NFA Example 1

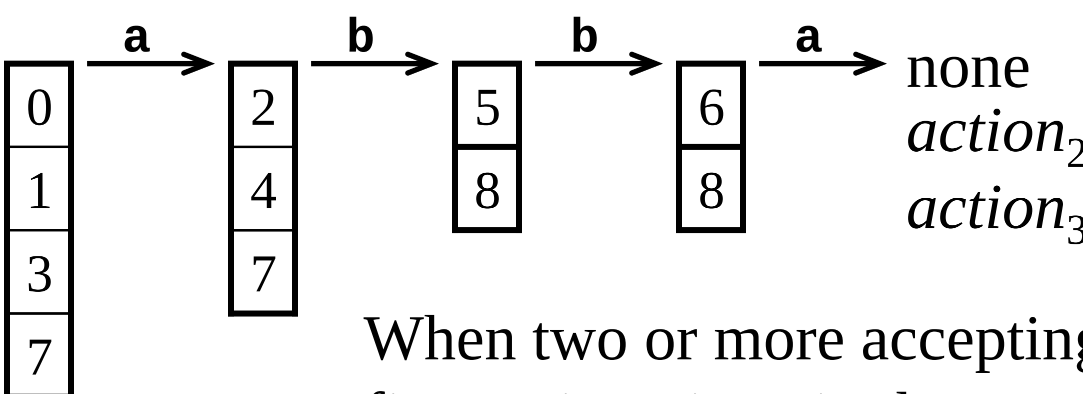




Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

Simulating the Combined NFA Example 2





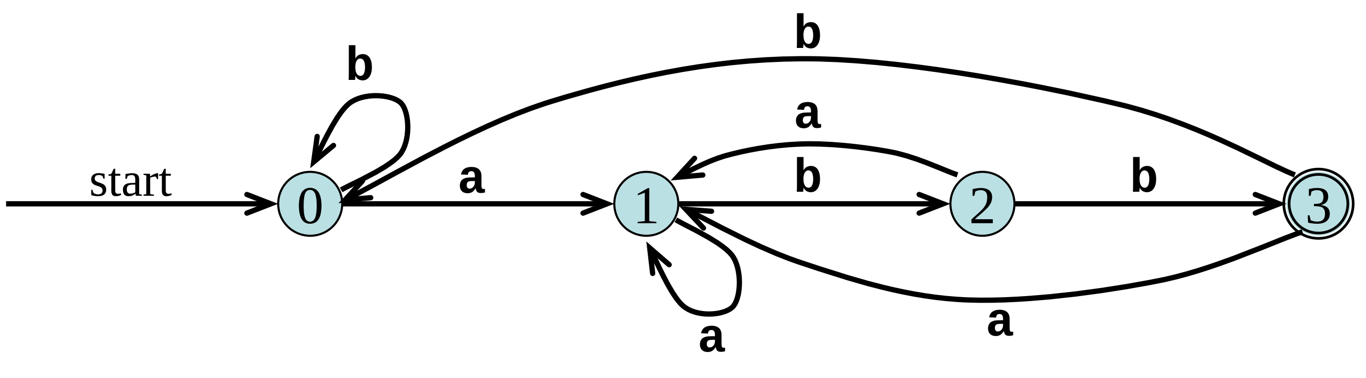
When two or more accepting states are reached, the first action given in the Lex specification is executed

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

Example DFA

A DFA that accepts (**a** | **b**)***abb**



Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:

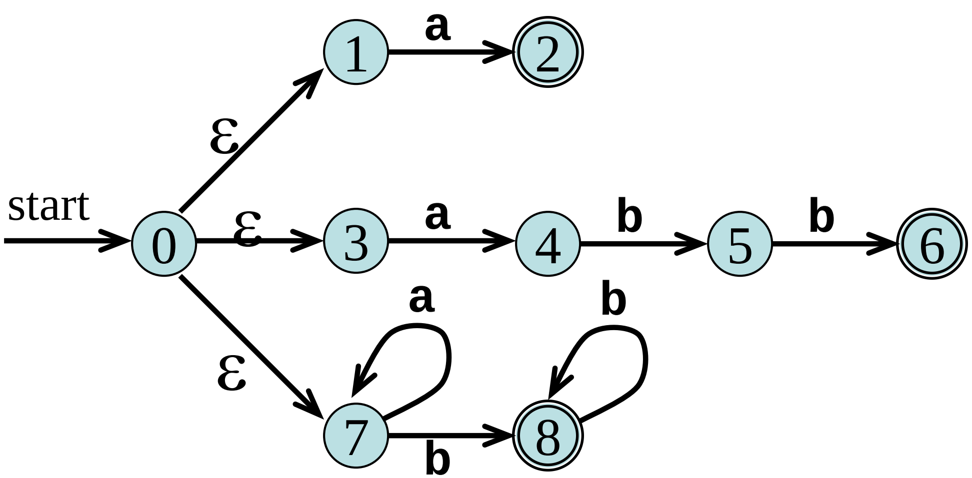
  $$\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \to_\varepsilon \ldots \to_\varepsilon t\}$$
  $$\varepsilon\text{-}closure(T) = \cup_{s \in T}\, \varepsilon\text{-}closure(s)$$
  $$move(T,a) = \{t \mid s \to_a t \text{ and } s \in T\}$$

- The algorithm produces:

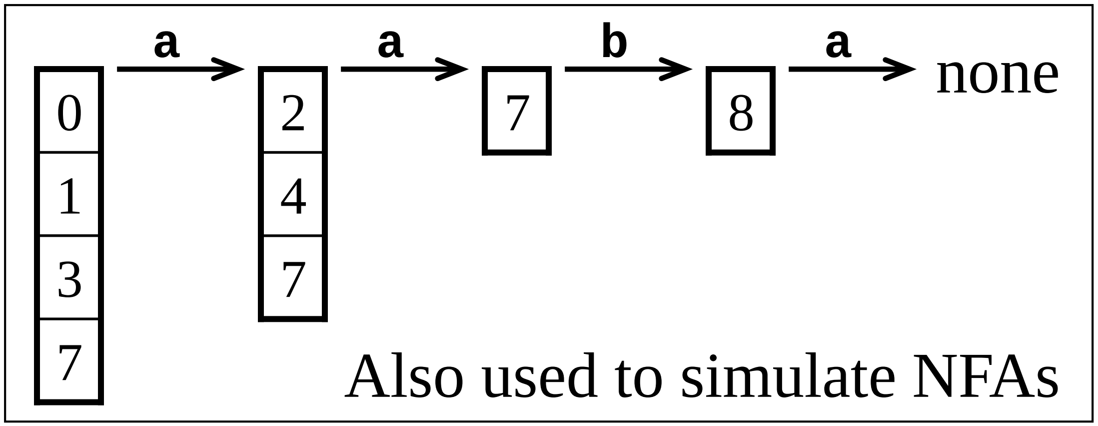
  Dstates is the set of states of the new DFA consisting of sets of states of the NFA

  Dtran is the transition table of the new DFA

ε-*closure* and *move* Examples



$\varepsilon\text{-}closure(\{0\}) = \{0,1,3,7\}$
$move(\{0,1,3,7\},\mathbf{a}) = \{2,4,7\}$
$\varepsilon\text{-}closure(\{2,4,7\}) = \{2,4,7\}$
$move(\{2,4,7\},\mathbf{a}) = \{7\}$
$\varepsilon\text{-}closure(\{7\}) = \{7\}$
$move(\{7\},\mathbf{b}) = \{8\}$
$\varepsilon\text{-}closure(\{8\}) = \{8\}$
$move(\{8\},\mathbf{a}) = \emptyset$



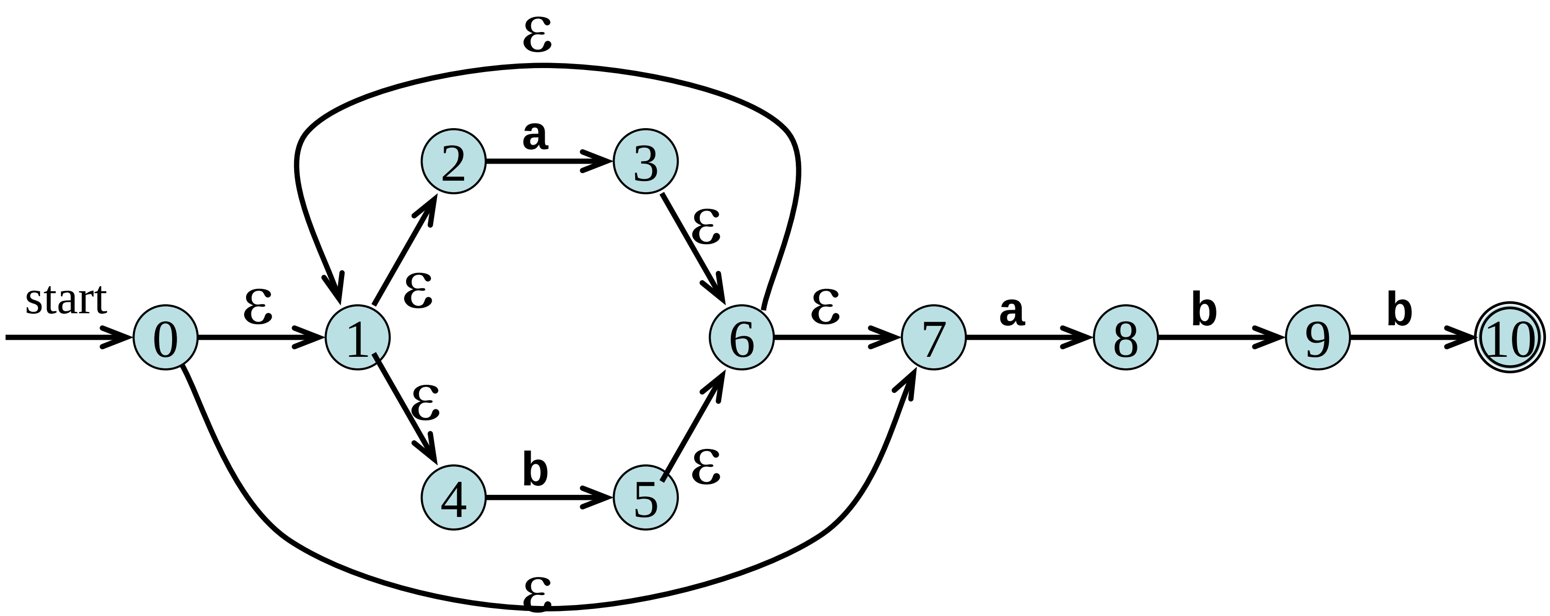
Simulating an NFA using ε-*closure* and *move*

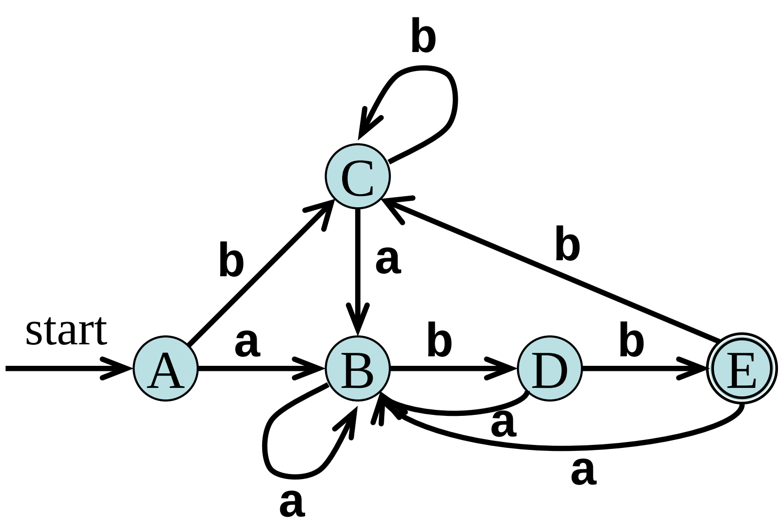
S := ε-*closure*($\{s_0\}$)
S_{prev} := $\emptyset$
a := *nextchar*()
while $S \neq \emptyset$ **do**
    S_{prev} := S
    S := ε-*closure*(*move*(S,a))
    a := *nextchar*()
end do
if $S_{prev} \cap F \neq \emptyset$ **then**
    execute *action in* S_{prev}
    return "yes"
else **return** "no"

The Subset Construction Algorithm

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}

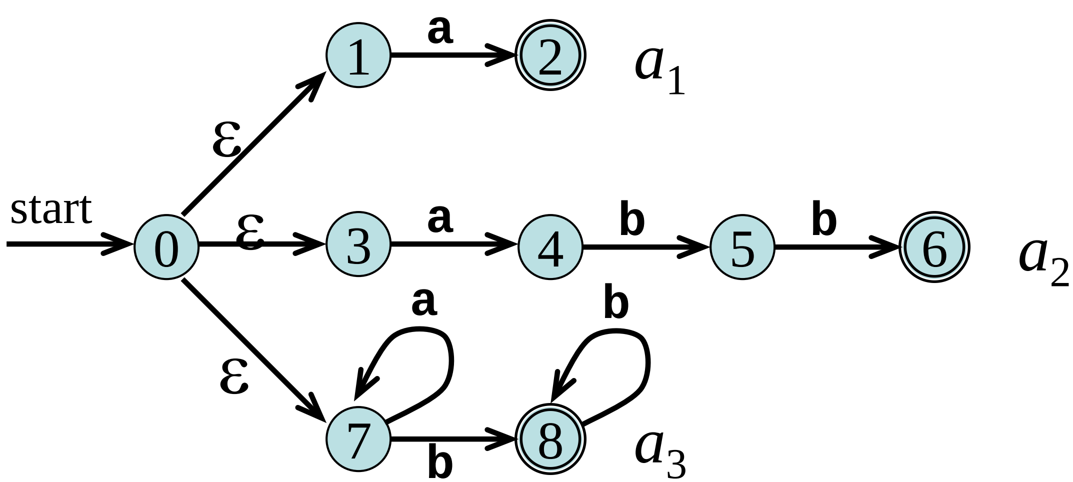
B = {1,2,3,4,6,7,8}

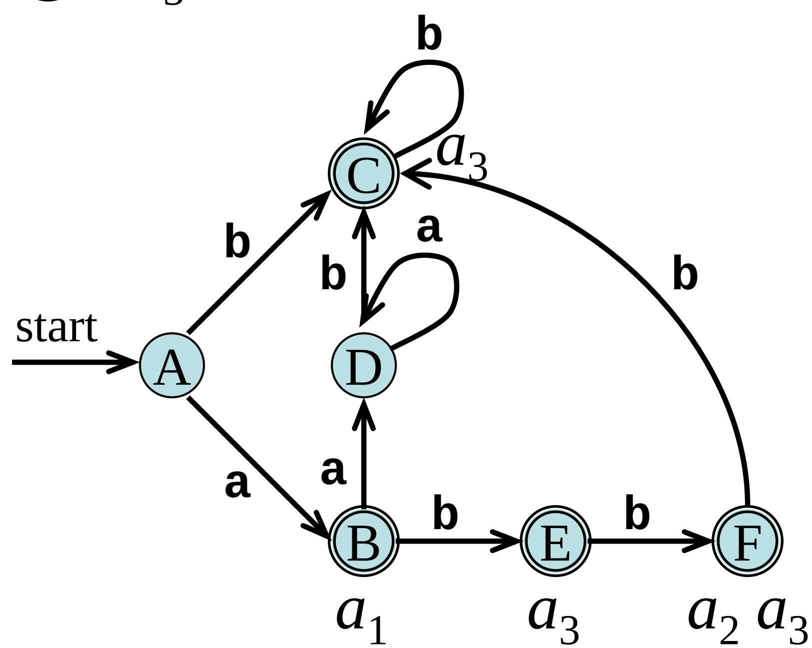
C = {1,2,4,5,6,7}

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}

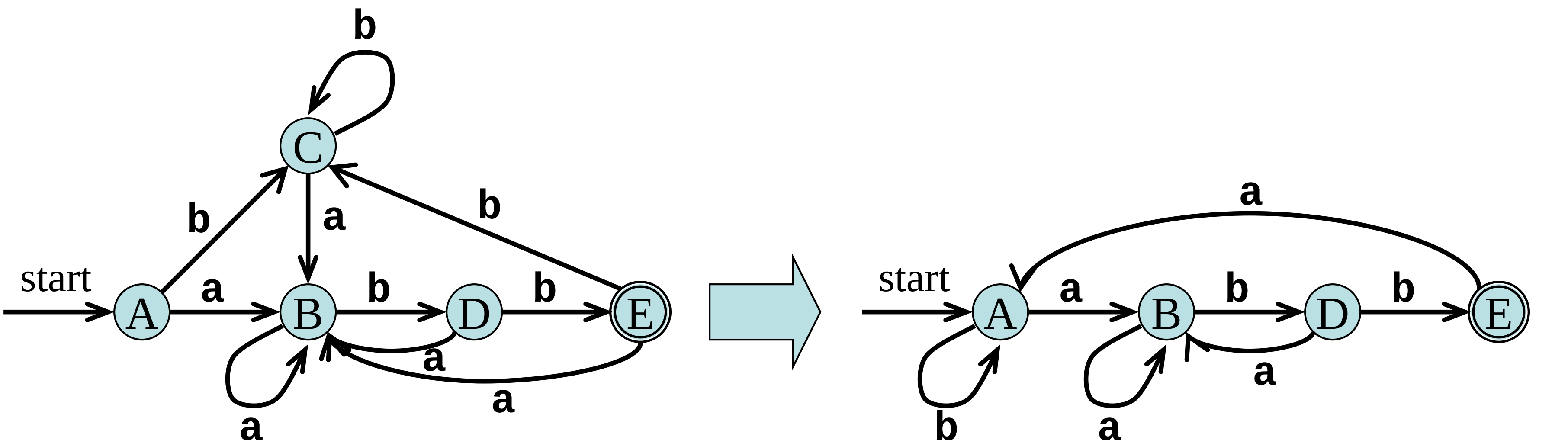
C = {8}

D = {7}

E = {5,8}

F = {6,8}

Minimizing the Number of States of a DFA (Hopcroft’s algorithm)



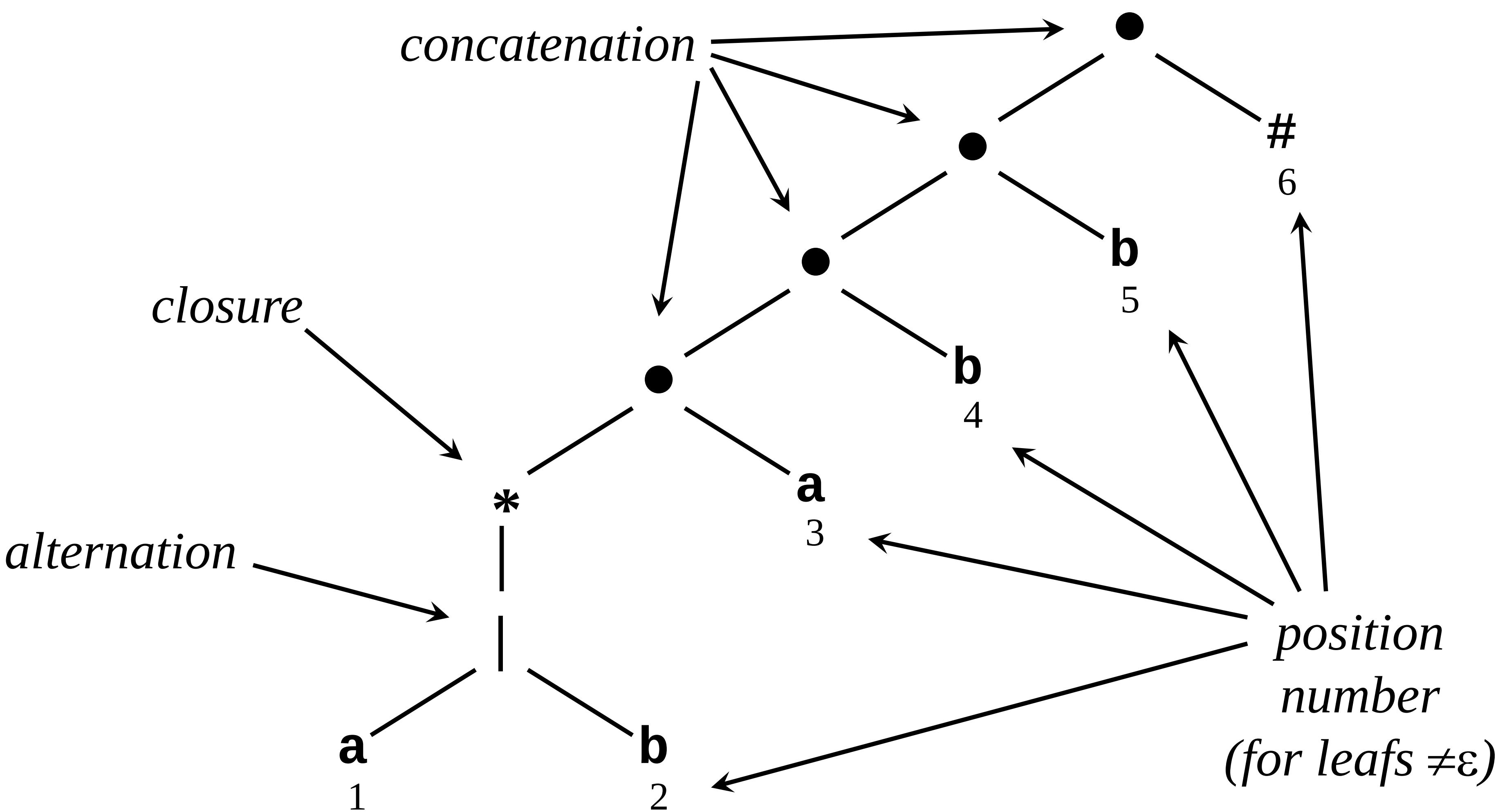
From Regular Expression to DFA Directly

- The "*important states*" of an NFA are those without an ε-transition, that is if $move(\{s\},a) \neq \emptyset$ for some a then s is an important state
- The subset construction algorithm uses only the important states when it determines $\varepsilon\text{-}closure(move(T,a))$

From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression *r* with a special end symbol # to make accepting states important: the new expression is *r*#
- Construct a syntax tree for *r*#
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

From Regular Expression to DFA Directly: Syntax Tree of (**a**|**b**)***abb#**



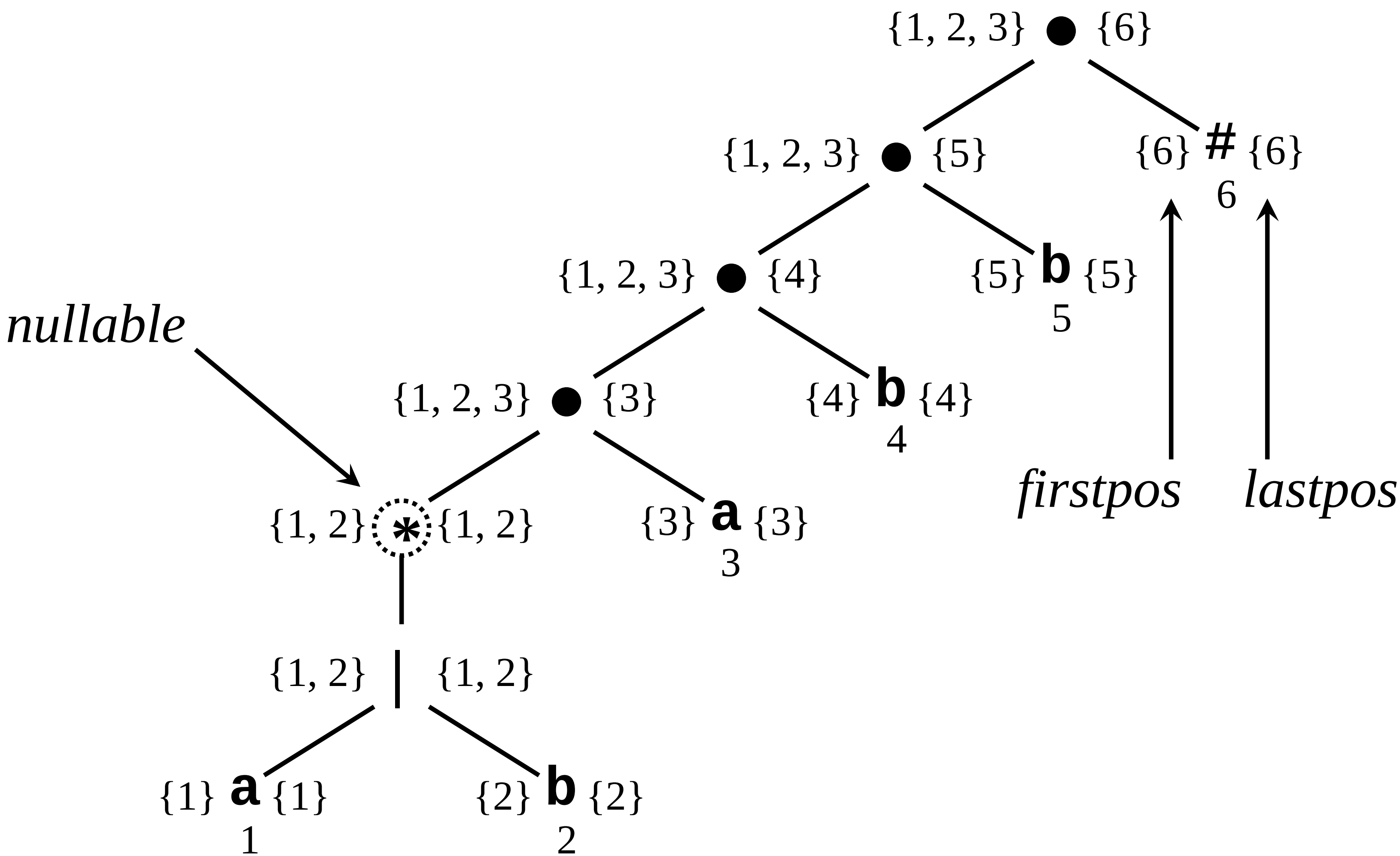
From Regular Expression to DFA Directly: Annotating the Tree

- *nullable*(*n*): the subtree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subtree at node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the subtree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree

From Regular Expression to DFA Directly: Annotating the Tree

Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
$\mid$ / \ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
$\bullet$ / \ c_1 c_2	$nullable(c_1)$ and $nullable(c_2)$	**if** $nullable(c_1)$ **then** $firstpos(c_1) \cup$ $firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \cup$ $lastpos(c_2)$ **else** $lastpos(c_2)$
* $\mid$ c_1	true	$firstpos(c_1)$	$lastpos(c_1)$

From Regular Expression to DFA Directly: Syntax Tree of (**a**|**b**)***abb#**



From Regular Expression to DFA Directly: *followpos*

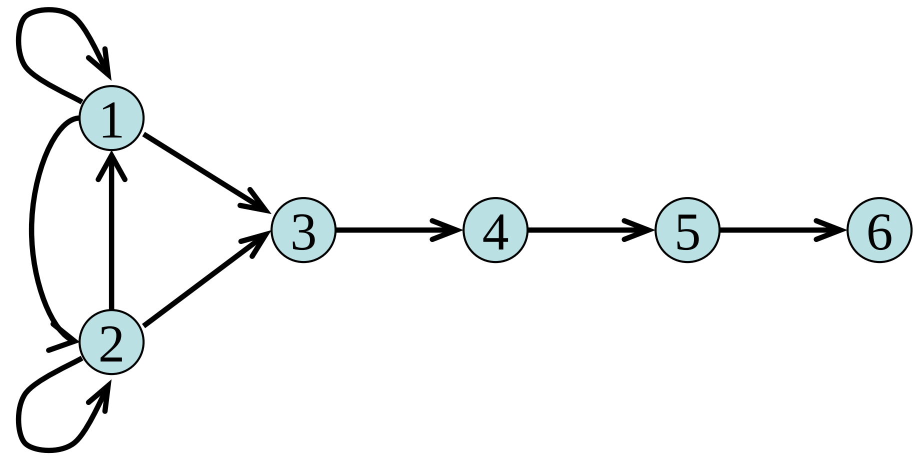
for each node n in the tree **do**
    if n is a cat-node with left child c_1 and right child c_2 **then**
        for each i in $lastpos(c_1)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(c_2)$
        end do
    else if n is a star-node
        for each i in $lastpos(n)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(n)$
        end do
    end if
end do

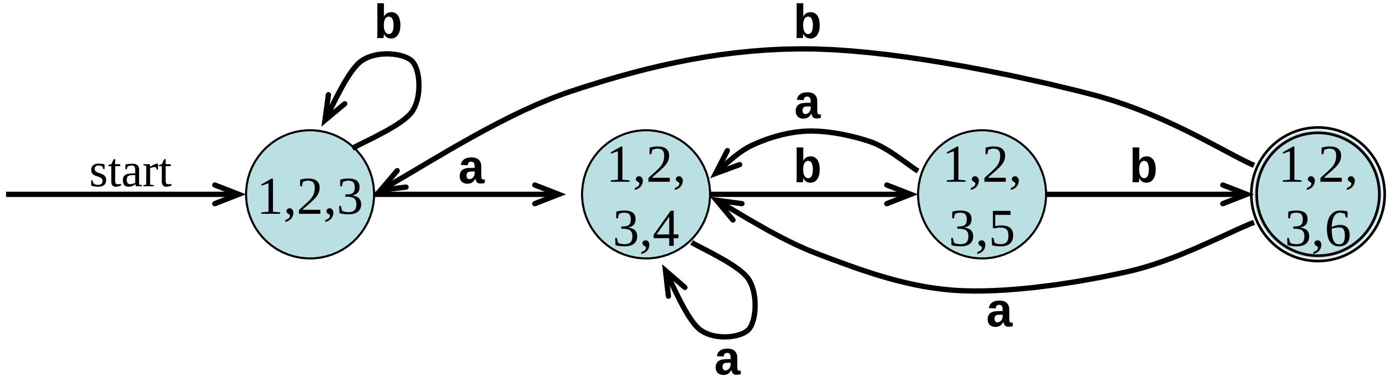
From Regular Expression to DFA Directly: Algorithm

s_0 := *firstpos*(*root*) where *root* is the root of the syntax tree
Dstates := {s_0} and is unmarked
while there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    for each input symbol $a \in \Sigma$ **do**
        let *U* be the set of positions that are in *followpos*(*p*)
            for some position *p* in *T*,
            such that the symbol at position *p* is *a*
        if *U* is not empty and not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        end if
        Dtran[*T*,*a*] := *U*
    end do
end do

From Regular Expression to DFA Directly: Example

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-





Time-Space Tradeoffs

Automaton	*Space (worst case)*	*Time (worst case)*
NFA	$O(\lvert r \rvert)$	$O(\lvert r \rvert \times \lvert x \rvert)$
DFA	$O(2^{\lvert r \rvert})$	$O(\lvert x \rvert)$

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall\ \alpha \in \Sigma$, transitions on α lead to equivalent states (DFA)
- α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition P of S

- Each $s \in S$ is in exactly one set $p_i \in P$
- The algorithm iteratively partitions the DFA's states

John Hopcroft

Initial partition, P_0, has two sets: $\{F\}$ & $\{Q-F\}$ $(D = (Q,\Sigma,\delta,q_0,F))$

Splitting a set ("partitioning a set by a")

- Assume q_a, & $q_b \in s$, and $\delta(q_a,\underline{a}) = q_x$, & $\delta(q_b,\underline{a}) = q_y$
- If q_x & q_y are not in the same set, then s must be split
- One state in the final DFA cannot have two transitions on a

DFA minimization: The idea

- **Equivalent states** : Two states *q and q' in a DFA M = (Q, Σ, δ, q0, F) are said to be* ***equivalent if for all strings u in Σ*, the states on which u ends on*** when read from *q and q' are both accept, or both non-accept.*

- **Remove *unreachable states*** *from start state.*

- **Remove *dead states**:* states that are not final and have transitions to themselves

The algorithm

- Input: DFA, S is the set of states, F is the set of final states.
- Output: minimized equivalent DFA.
- Steps:

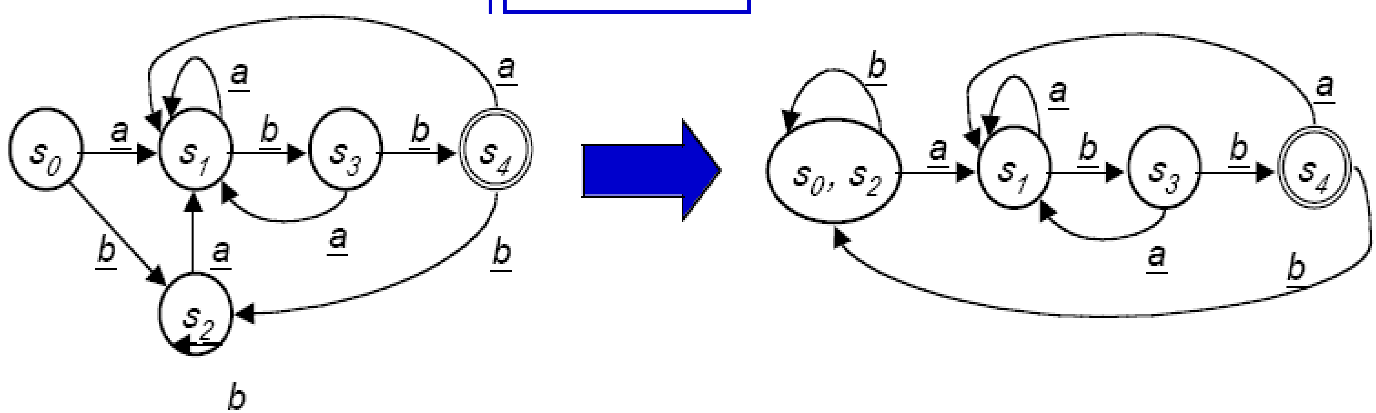
```
Π= (F) (S-F);
While (Π is changed) {
    for each group G of  Π do {
       partition G if there are distinguishable states in G;
       replace G by the subgroups found;
    }
}
Choose representative state for each group;
Remove dead states;
Remove states not reachable from the start state;
```

A Detailed Example

Applying the minimization algorithm to the DFA

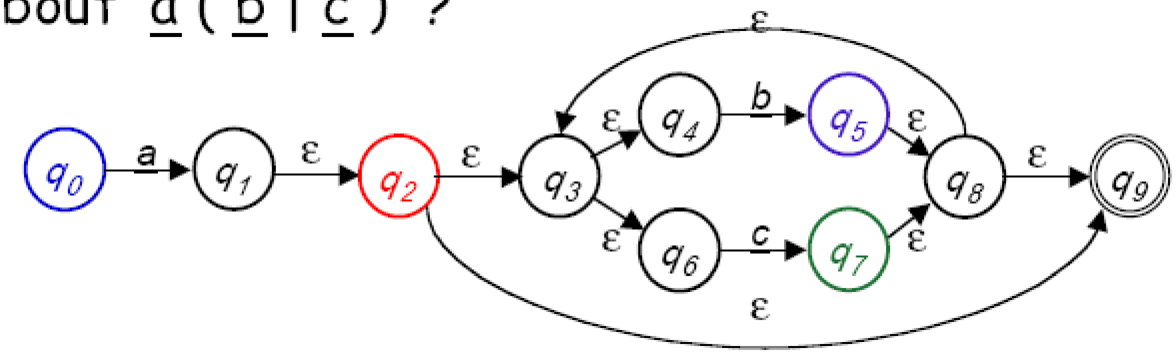
	Current Partition	*Worklist*	*s*	*Split on a*	*Split on b*
P_0	$\{s_4\}\ \{s_0,s_1,s_2,s_3\}$	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}\ \{s_3\}\ \{s_0,s_1,s_2\}$	$\{s_0,s_1,s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\ \{s_3\}\ \{s_1\}\ \{s_0,s_2\}$	$\{s_0,s_2\}\{s_1\}$	$\{s_1\}$	none	none

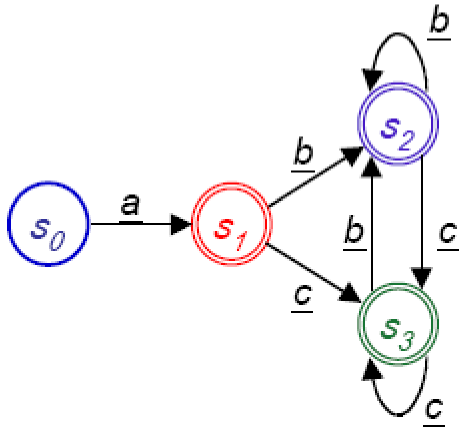
final state



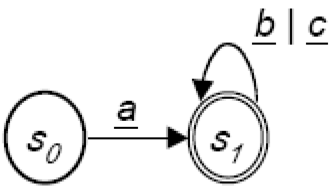
Example

What about a (b | c)* ?



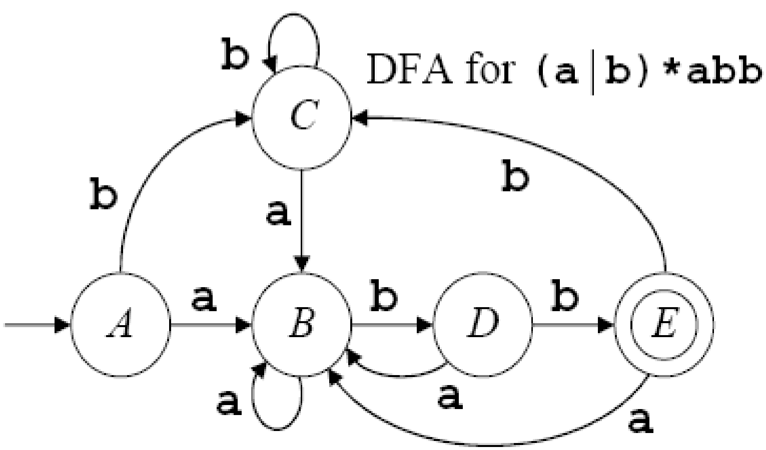


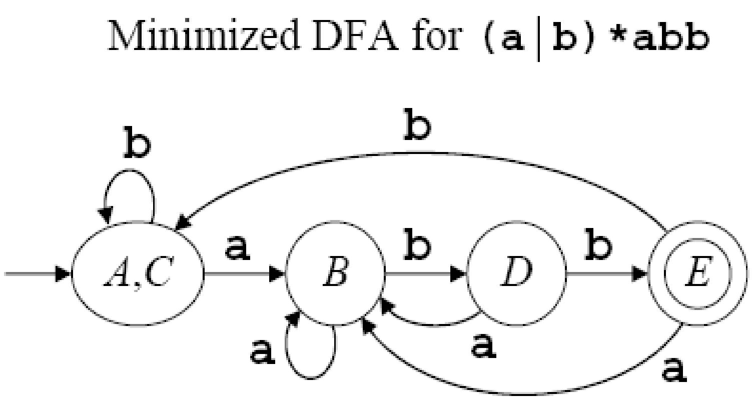
To produce the minimal DFA



Example on DFA Minimization

- Consider the DFA for `(a|b)*abb` obtained using subset construction algorithm
- Initial partition Π consists of 2 groups = $\{\{A, B, C, D\}, \{E\}\}$
- $\{A, B, C\}$-succ under $b \in \{A, B, C, D\}$, while D-succ under b is E
- Therefore, $\Pi_{new} = \{\{A, B, C\}, \{D\}, \{E\}\}$
- $\{A, C\}$-succ under b is C while B-succ under b is D
- Therefore, $\Pi_{new} = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$
- $\{A, C\}$-succ under a is B, and $\{A, C\}$-succ under b is C
- $\{A, C\}$ does not require further partitioning; states A and C can be merged
- Therefore, final $\Pi = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$





Building Faster Scanners from the DFA

Table-driven recognizers waste effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in *action()*
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

```
char ← next character;
state ← s0;
call action(state,char);
while (char ≠ eof)
   state ← δ(state,char);
   call action(state,char);
   char ← next character;

if T(state) = final then
   report acceptance;
else
   report failure;
```

Building Faster Scanners from the DFA

A direct-coded recognizer for <u>r</u> *Digit Digit**

```
     goto s0;
s0:  word ← Ø;
     char ← next character;
     if (char = 'r')
        then goto s1;
        else goto se;
s1:  word ← word + char;
     char ← next character;
     if ('0' ≤ char ≤ '9')
        then goto s2;
        else goto se;
s2:  word ← word + char;
     char ← next character;
     if ('0' ≤ char ≤ '9')
        then goto s2;
        else if (char = eof)
            then report success;
            else goto se;
se:  print error message;
     return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Minimizing the Number of States in a DFA

Smaller is better!

Minimal DFA

- Given any DFA, there is an equivalent DFA containing the minimum number of states
- The minimal DFA is unique
- It is possible to directly obtain the minimal DFA from any DFA
- The algorithm presented here is adapted from Aho, Sethi, and Ullman.

Minimal DFA Algorithm 1

- The algorithm starts by partitioning the states in the DFA into sets of states that will ultimately be combined into single states.
- The first partitioning creates 2 sets:
  - One set contains all the accepting states
  - The other set contains all the nonaccepting states
- The process now goes through one or more iterations where it considers the transitions on each character of the alphabet

Minimal DFA Algorithm 2

- Iterate until no further partitioning is possible:
  - For each set G of states in partition Π, consider the transitions for each input symbol **a** from any state in G.
  - Two states **s** and **t** belong in the same subgroup iff for all input symbols **a**, states **s** and **t** have transitions into states in the same subgroup of Π.
  - Replace G in Π by the set of subgroups formed.

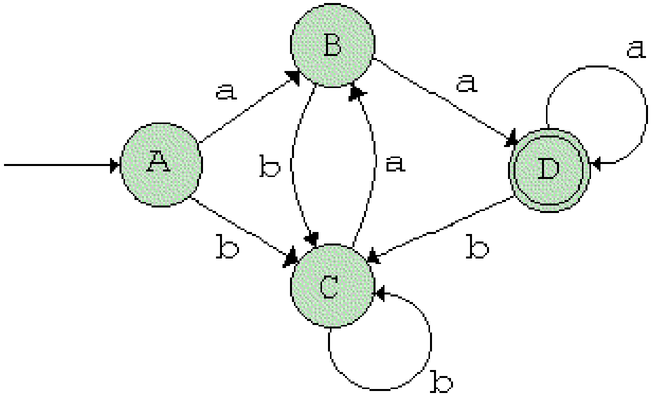
Minimal DFA Algorithm 3

- Choose one state in each group of the partition Π as the representative for that group. The representatives will be the states of the reduced DFA **M’**.
- The start state of **M’** will be the group that contains the start state of the original DFA.
- Any group that contains an accepting state from the original DFA will be an accepting state of the minimal DFA **M’.**

Minimal DFA Algorithm 4

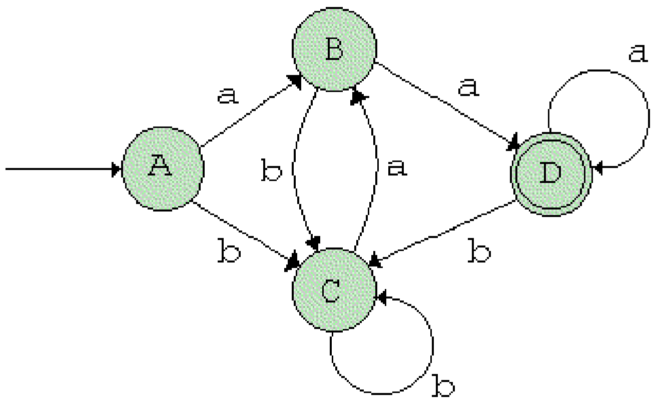
- Remove any dead state **d** from **M'.**
  - a dead state is one that has transitions to itself on all input symbols.
  - Any transitions to **d** from other states become undefined.
- Remove any states unreachable from the start state from **M'.**

Example 1: Minimize the DFA



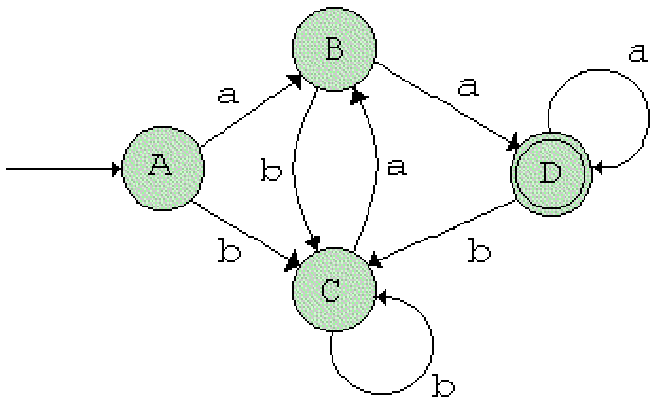
- We start with two groups
  - Accepting states: { D }
  - Nonaccepting states: { A, B, C }
- Since the singleton set { D } cannot be partitioned any further, we concentrate on { A, B, C }

Example 1, continued



- For input a and states in group { A, B, C }
  - T(A, a) = B
  - T(B, a) = D (maps into a different subgroup)
  - T(C, a) = B
- We must split the group { A, B, C } into two subgroups, { A, C } and { B}

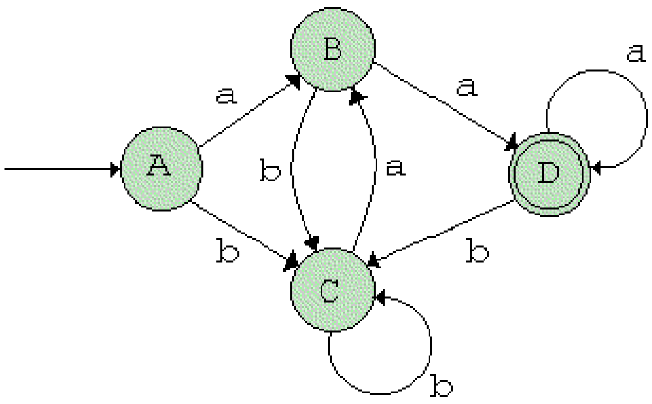
Example 1, continued



- At this point, Π = { A, C }, { B }, { D }
- Consider transitions from { A, C } on **a** and **b**
  - T(A, a) = B T(C, a) = B(same group)
  - T(A, b) = C T(C, b) = C (same group)
- No further partitioning is possible.

Example 1, continued

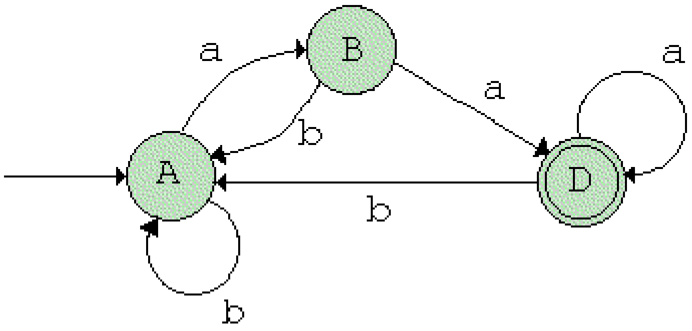
- Π = {A,C}, {B}, {D}
- Choose A as representative from {A,C}:
  - Remove row C from table
  - Replace all occurrences of C with A
- Resulting minimal DFA is shown on next slide



	a	b	
A	B	C	start
B	D	C	
C	B	C	
D	D	C	accept

Example 1, conclusion

- Minimal DFA



	a	b	
A	B	A	start
B	D	A	
D	D	A	accept

Checkpoint: Minimize the DFA

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B,C,D,F,G},{E,H}
- T({A,B,C,D,F,G}, a):
  - T(A, a) = G
  - T(B, a) = C
  - T(C, a) = B
  - T(D, a) = G
  - T(F, a) = A
  - T(G, a) = B
  - all map to same group—no repartitioning (yet)

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B,C,D,F,G},{E,H}
- T({A,B,C,D,F,G},b):
  - T(A, b) = F
  - T(B, b) = G
  - T(C, b) = D
  - T(D, b) = E (different group)
  - T(F, b) = D
  - T(G, b) = D
  - Partition {A,B,C,D,F,G} into {A,B,C,F,G}, {D}

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B,C,F,G},{D},{E,H}
- T({E,H}, a):
  - T(E, a) = B
  - T(H, a) = A
  - map into same group
- T({E,H}, b):
  - T(E, b) = H
  - T(H, b) = E
  - map into same group
- No repartitioning necessary (at least not yet)

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B,C,F,G},{D},{E,H }
- T({A,B,C,F,G}, a):
  - T(A, a) = G
  - T(B, a) = C
  - T(C, a) = B
  - T(F, a) = A
  - T(G, a) = B
  - all map to same group, so no repartitioning results from this

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B,C,F,G},{D},{E,H}
- T({A,B,C,F,G}, b):
  - T(A, b) = F
  - T(B, b) = G
  - T(C, b) = D
  - T(F, b) = D
  - T(G, b) = D
  - {A,B} and {C,F,G} map to different groups, so we repartition {A,B,C,F,G} into the groups {A,B} and {C,F,G}

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B},{C,F,G},{D},{E,H}
- T({A,B}, a) → {C,F,G}
- T({A,B}, b) → {C,F,G}
- T({C,F,G}, a) → {A,B}
- T({C,F,G}, b) → {D}
- T({D}, a) → {C,F,G}
- T({D}, b) → {E, H}
- T({E,H}, a) → {A, B}
- T({E,H}, b) → {E, H}
- No further partitioning is possible

Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- Π={A,B},{C,F,G},{D},{E,H}
- Representatives:
  - A = {A,B}
  - C = {C,F,G}
  - D = {D}
  - E = {E,H}
- The minimal DFA is shown on the next slide.

Checkpoint Solution

	a	b	
A	C	C	start
C	A	D	
D	C	E	
E	A	E	accept