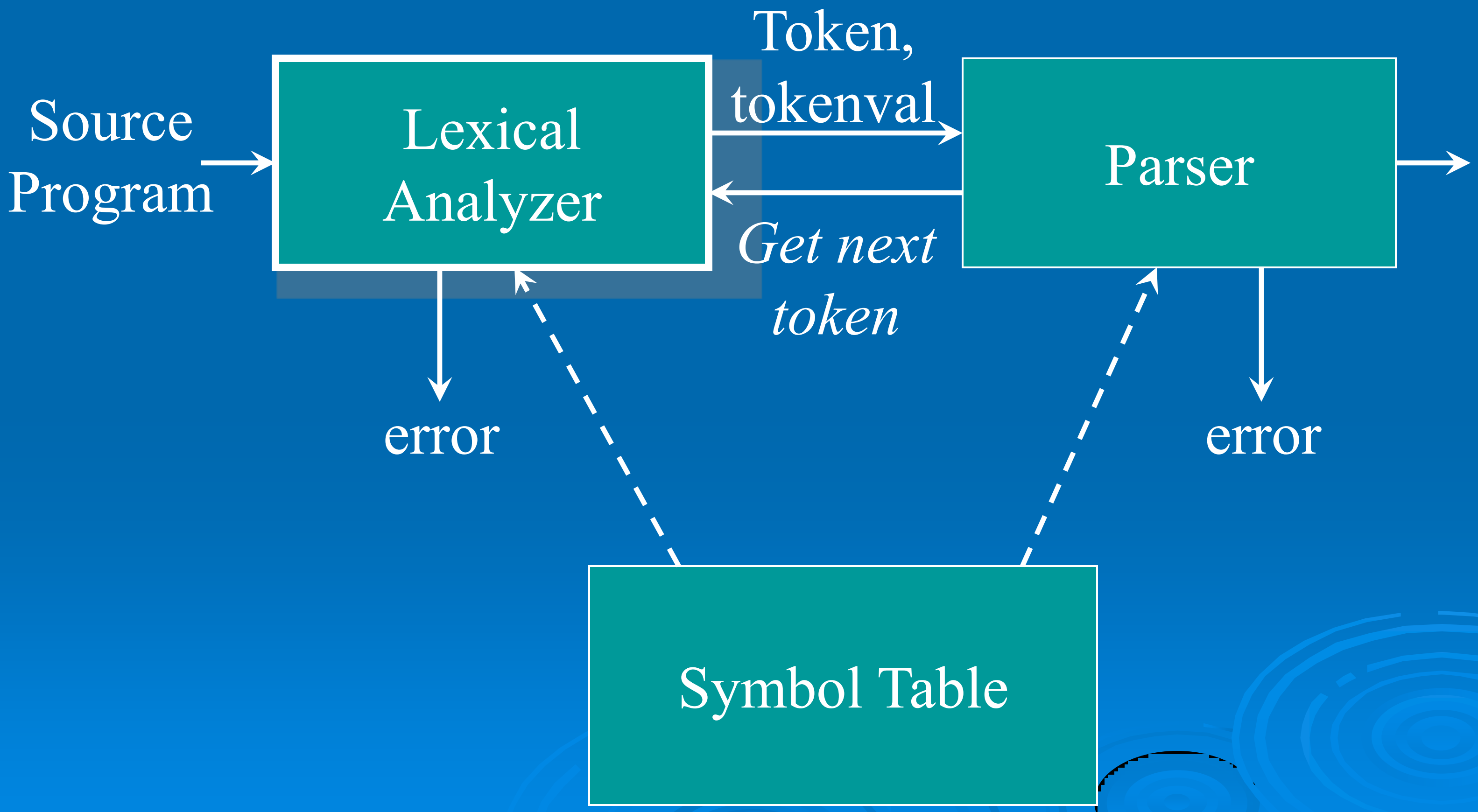
Lexical Analysis

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. UTF8, trigraphs)

Interaction of the Lexical Analyzer with the Parser



Source Program

Lexical Analyzer

Token, tokenval

Get next token

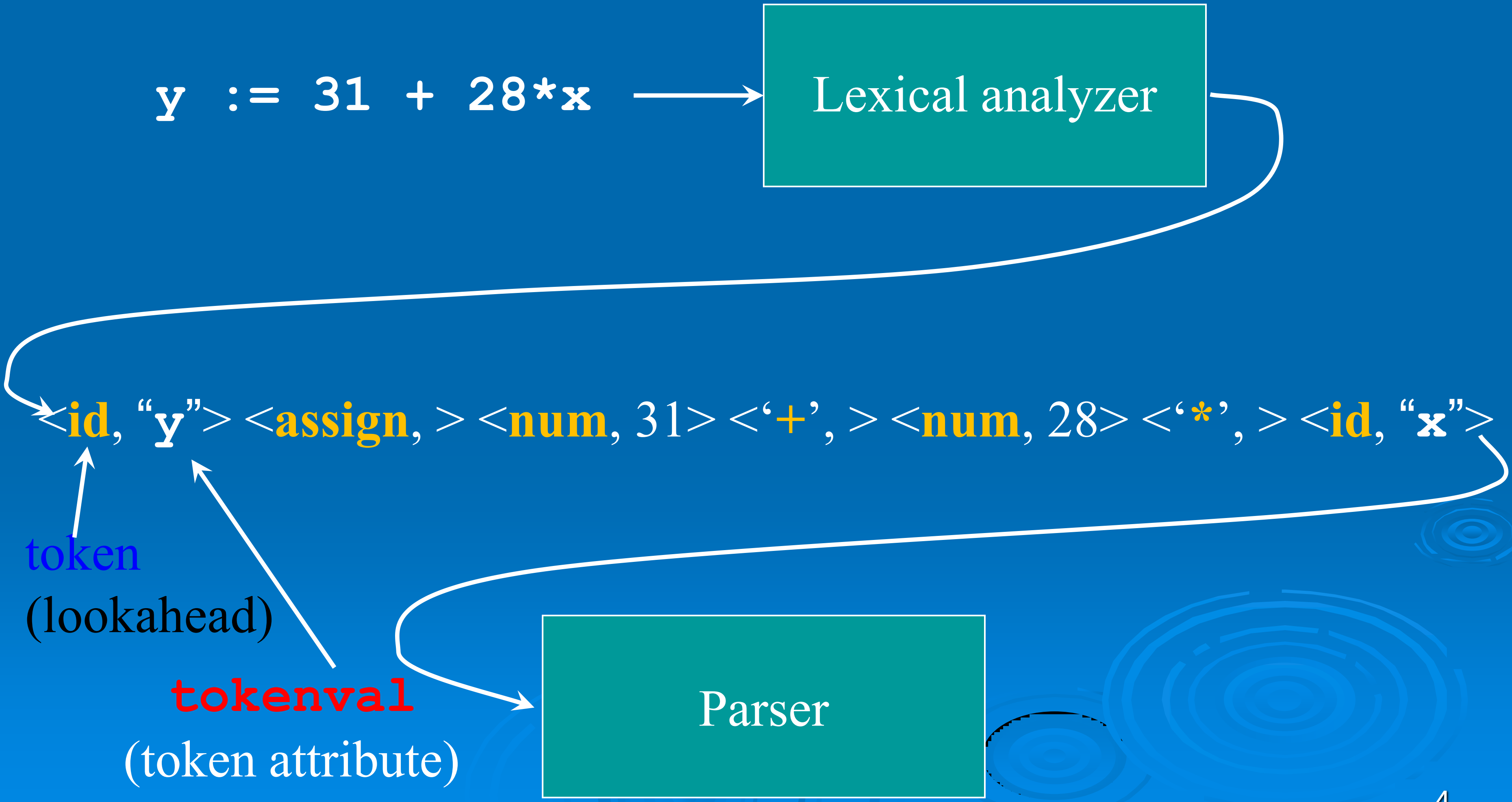
Parser

error

error

Symbol Table

Attributes of Tokens



`y := 31 + 28*x`

Lexical analyzer

<**id**, “**`y`**”> <**assign**, > <**num**, 31> <‘**+**’, > <**num**, 28> <‘*****’, > <**id**, “**`x`**”>

token
(lookahead)

`tokenval`
(token attribute)

Parser

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 *exponentation* 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 with name-to-regular-expression bindings:

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

  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:

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

- Regular definitions cannot be recursive:

$$\textbf{digits} \rightarrow \textbf{digit digits} \mid \textbf{digit} \qquad \textit{wrong!}$$

Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

$$r^{+} = rr^{*}$$

$$r? = r \mid \varepsilon$$

$$\texttt{[a-z]} = \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z}$$

- Examples:

  digit → [0-9]

  num → $\textbf{digit}^{+}$ (**.** $\textbf{digit}^{+}$)? (**E** (**+** | **-**)? $\textbf{digit}^{+}$)?

Regular Definitions and Grammars

Grammar

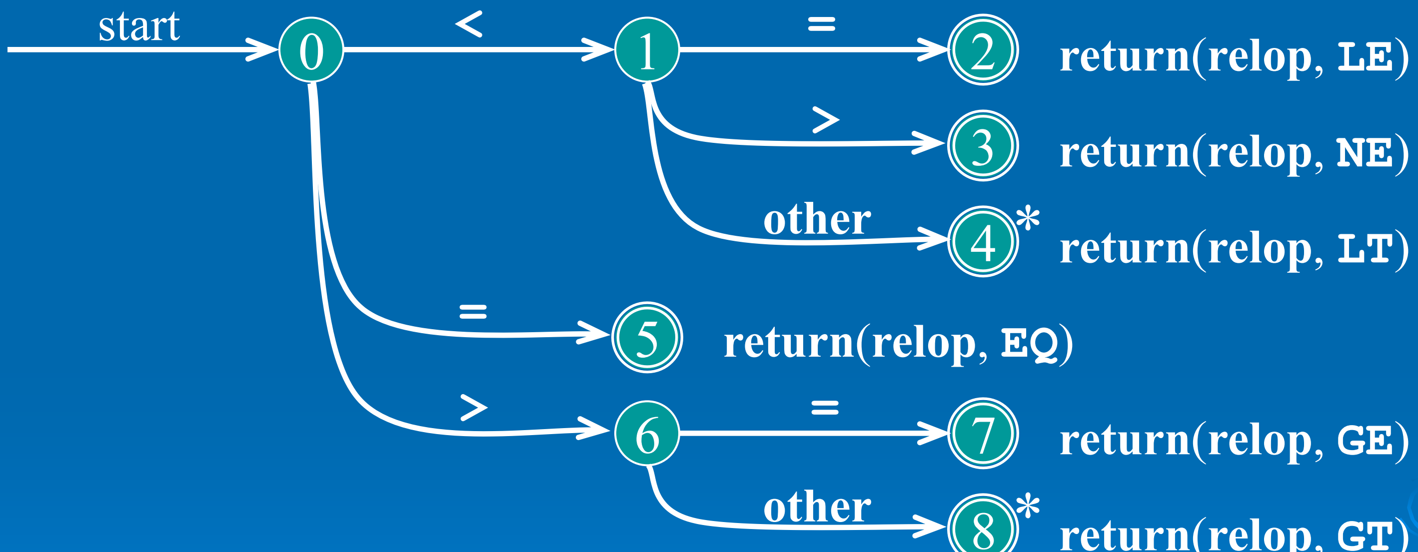
stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| ε
expr → *term* **relop** *term*
| *term*
term → **id**
| **num**

Regular definitions

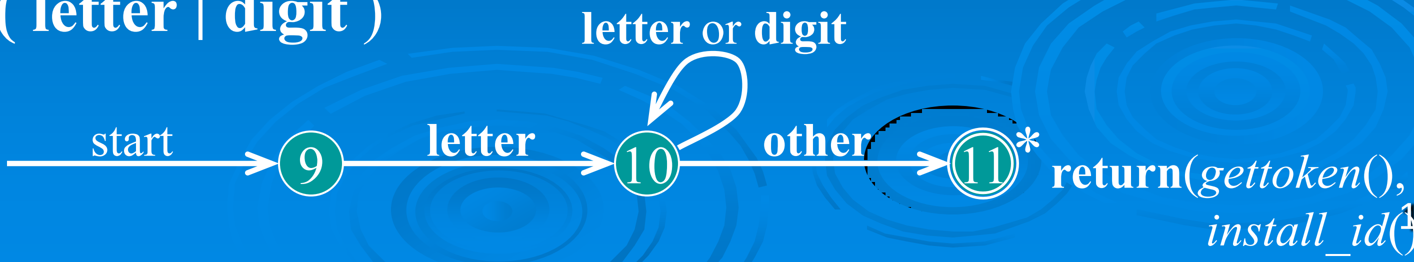
if → `if`
then → `then`
else → `else`
relop → `<` | `<=` | `<>` | `>` | `>=` | `=`
id → **letter (letter | digit)***
num → **digit**$^+$ (. **digit**$^+$)? (**E** (+ | -)? **digit**$^+$)?

Coding Regular Definitions in *Transition Diagrams*

relop → < | <= | <> | > | >= | =



id → **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;
     …
```

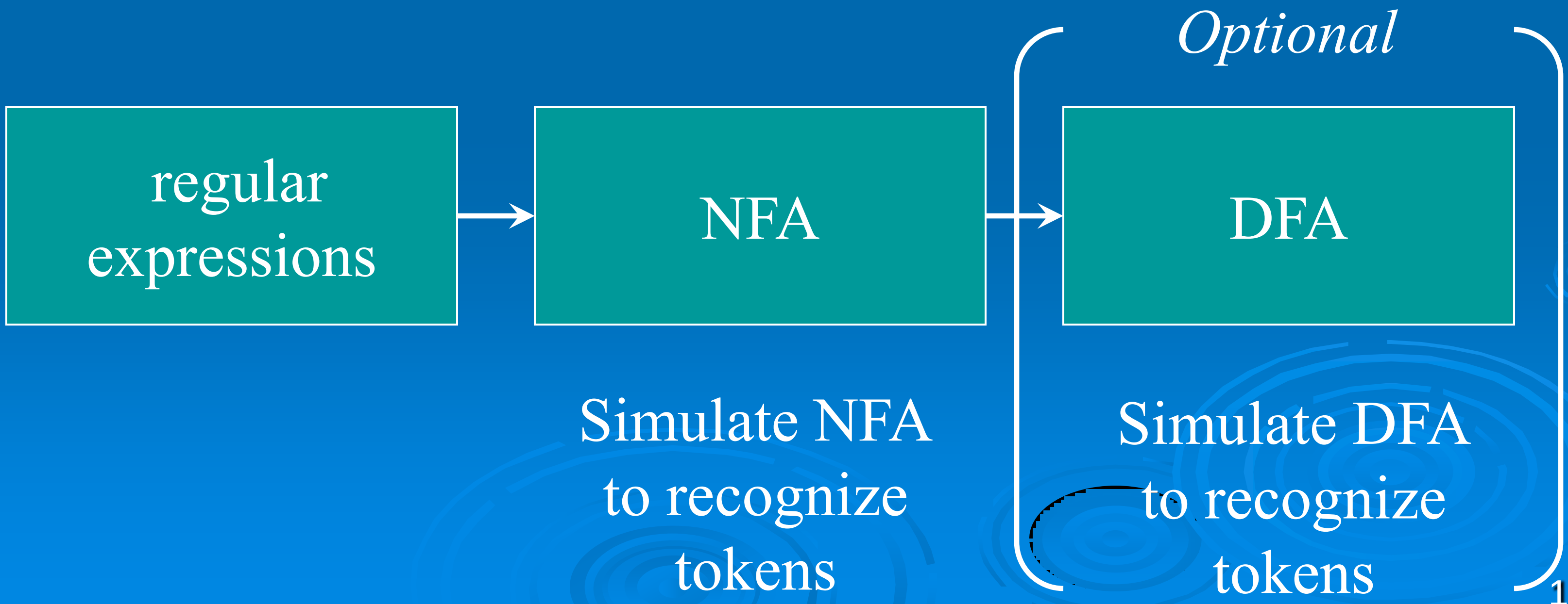
Decides the
next start state
to check

↓

```
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;
}
```

Design of a Lexical Analyzer Generator

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



Optional

regular expressions

NFA

DFA

Simulate NFA to recognize tokens

Simulate DFA to recognize tokens

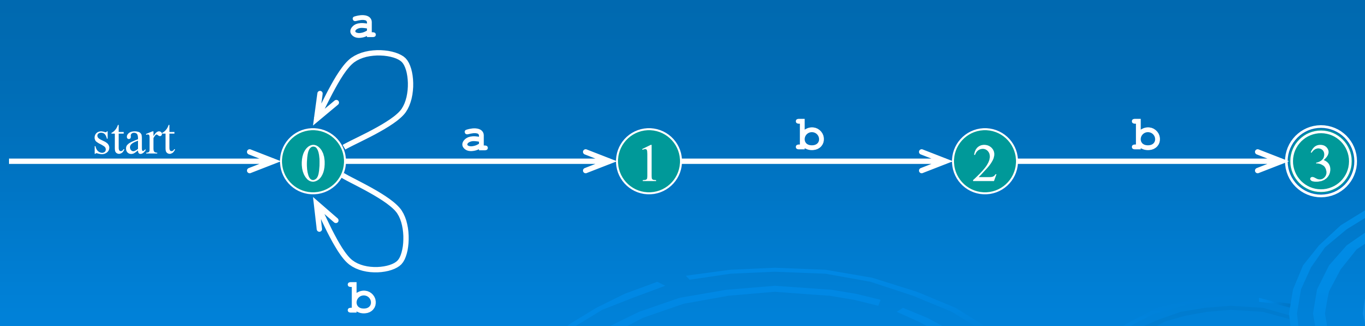
Nondeterministic Finite Automata

- 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*

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

→

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

The Language Defined by an NFA

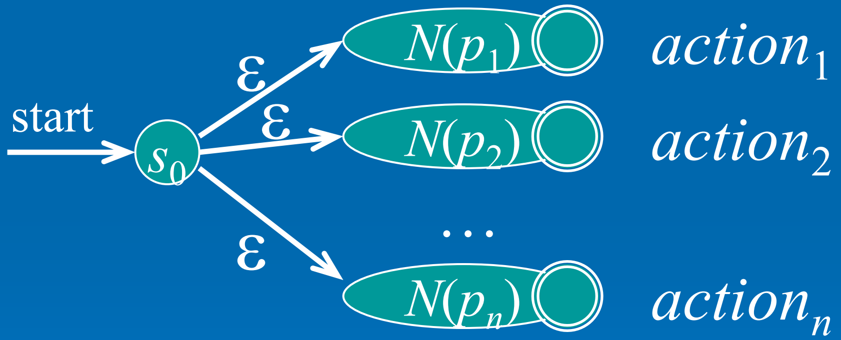
- An NFA *accepts* an input string *x* if and only if there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* 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

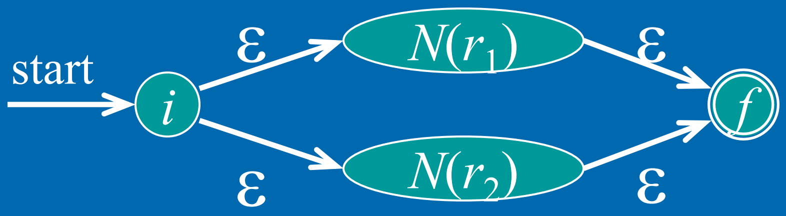
DFA

From Regular Expression to NFA (Thompson's Construction)

ε

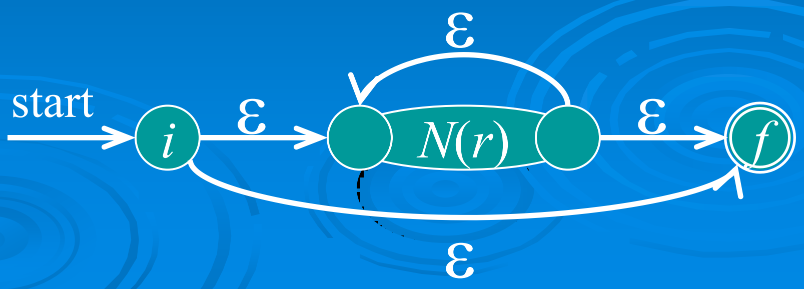
a

$r_1 \mid r_2$



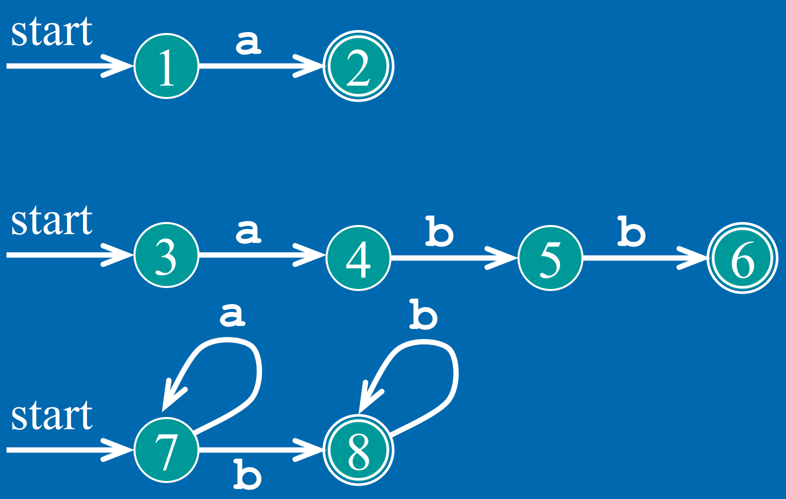
$r_1 r_2$

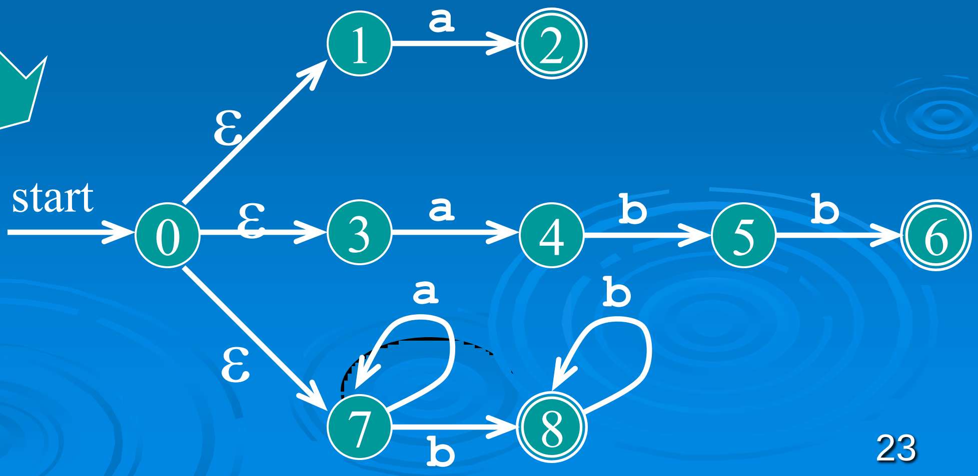
r^*



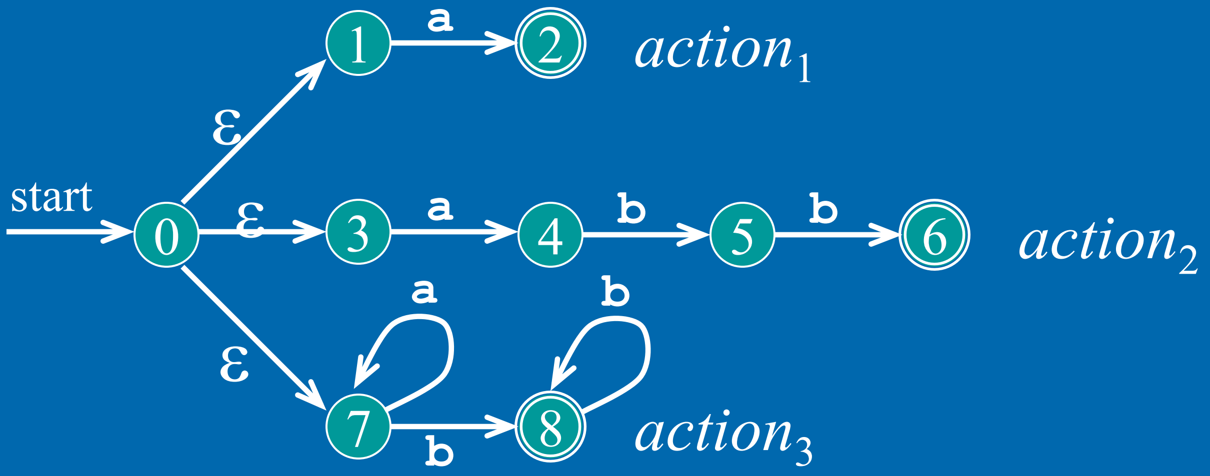
Combining the NFAs of a Set of Regular Expressions

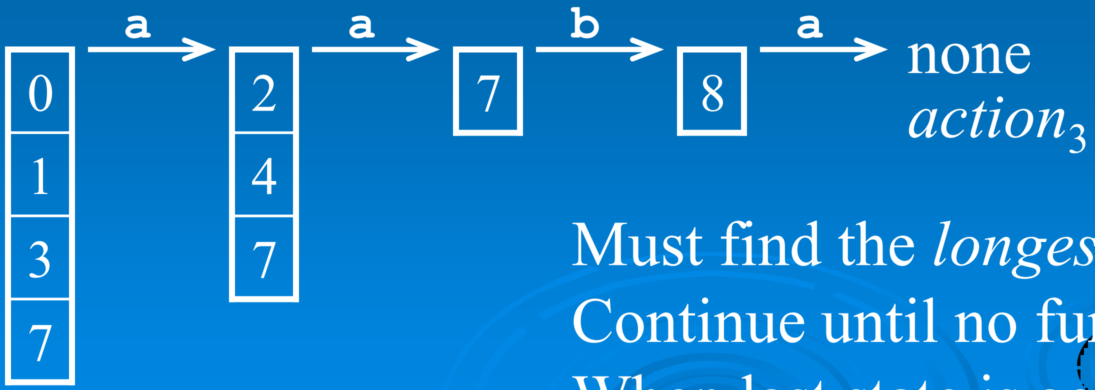
a	$\{ \mathit{action}_1 \}$
abb	$\{ \mathit{action}_2 \}$
a*b+	$\{ \mathit{action}_3 \}$





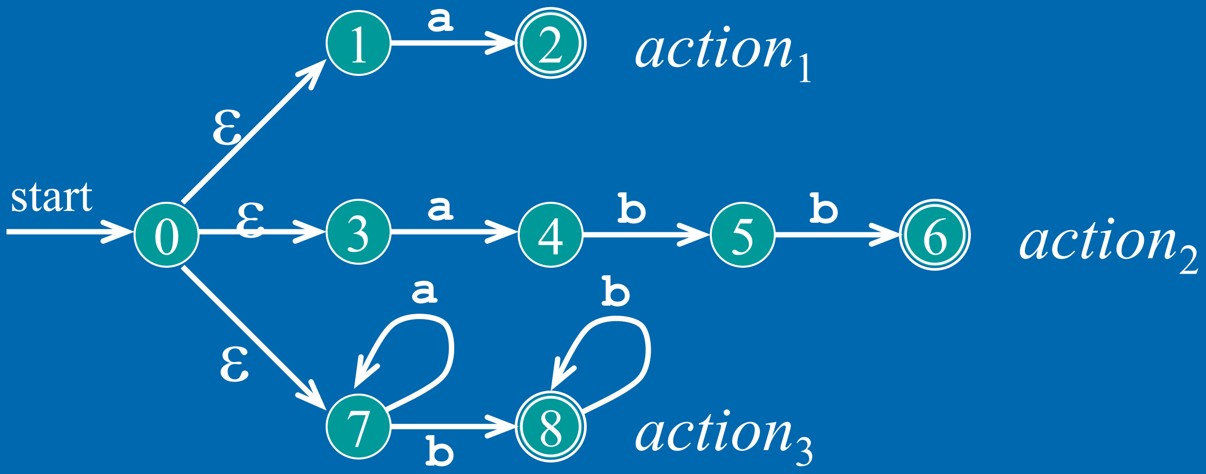
Simulating the Combined NFA Example 1

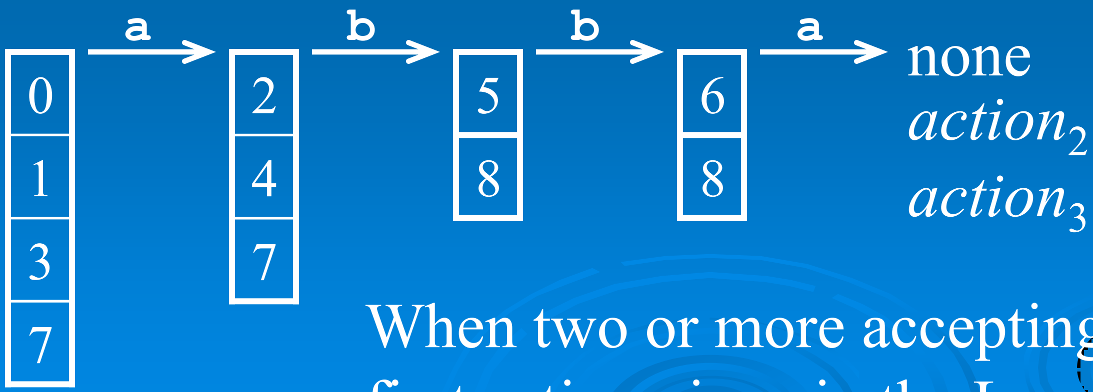




Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

Simulating the Combined NFA Example 2





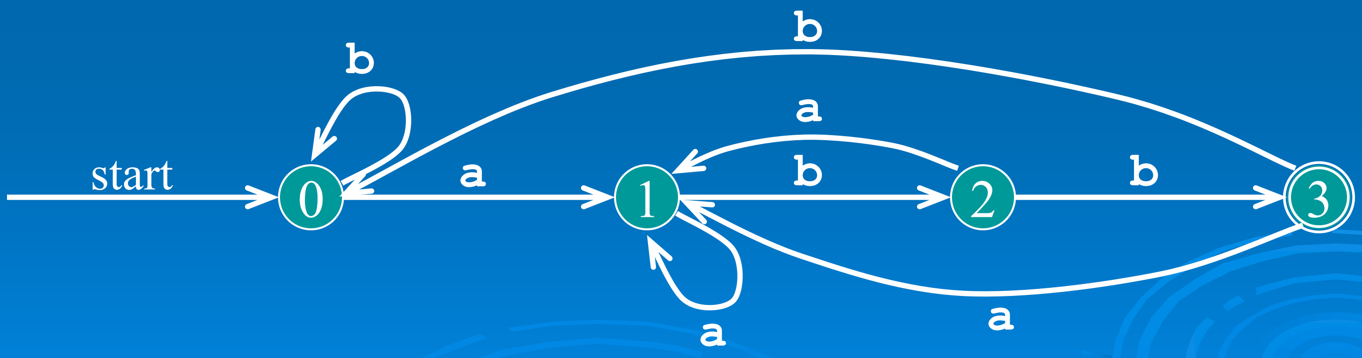
When two or more accepting states are reached, the first action given in the Lex specification is executed

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**



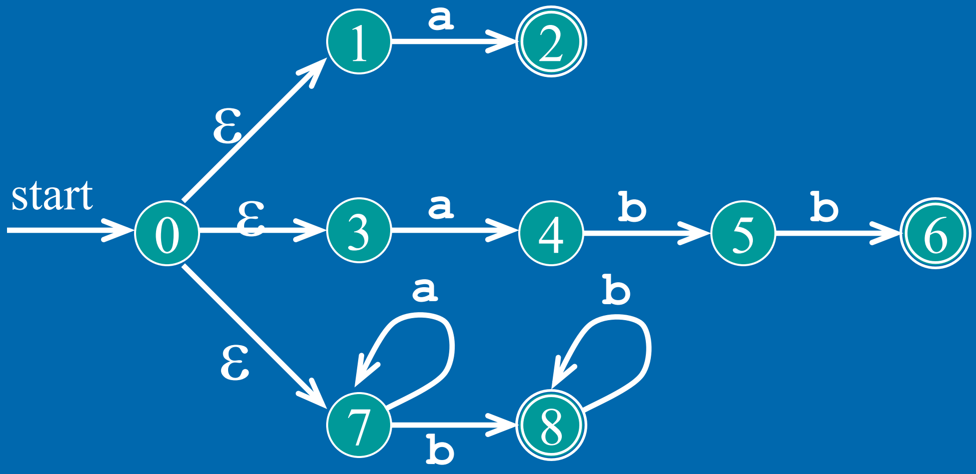
Conversion of an NFA into a DFA

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

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

- 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



$\varepsilon\text{-}closure(\{0\}) = \{0,1,3,7\}$

$move(\{0,1,3,7\},\mathbf{a}) = \{2,4,7\}$

$\varepsilon\text{-}closure(\{2,4,7\}) = \{2,4,7\}$

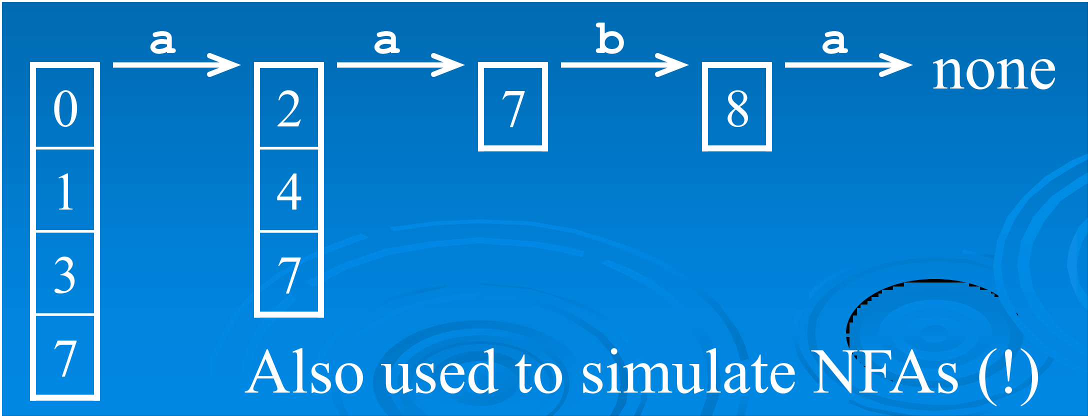
$move(\{2,4,7\},\mathbf{a}) = \{7\}$

$\varepsilon\text{-}closure(\{7\}) = \{7\}$

$move(\{7\},\mathbf{b}) = \{8\}$

$\varepsilon\text{-}closure(\{8\}) = \{8\}$

$move(\{8\},\mathbf{a}) = \varnothing$



Also used to simulate NFAs (!)

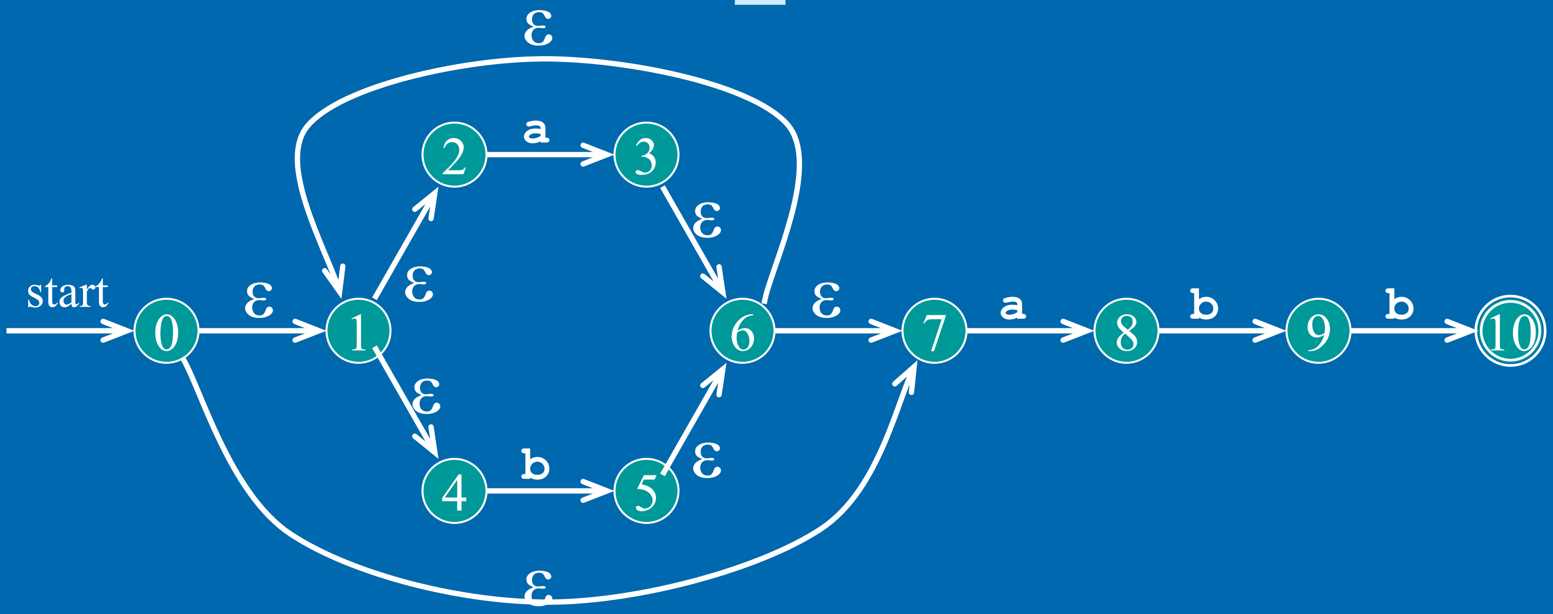
Simulating an NFA using ε-*closure* and *move*

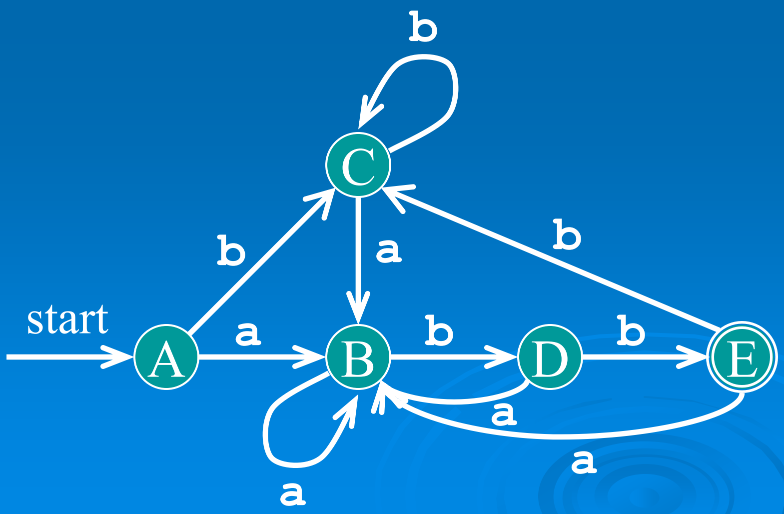
```
$S := \varepsilon\text{-}closure(\{s_0\})$
$S_{prev} := \emptyset$
$a := nextchar()$
while $S \neq \emptyset$ do
        $S_{prev} := S$
        $S := \varepsilon\text{-}closure(move(S,a))$
        $a := nextchar()$
end do
if $S_{prev} \cap F \neq \emptyset$ then
        execute $action\ in\ S_{prev}$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

The Subset Construction Algorithm

```
Initially, $\varepsilon$-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
        $U := \varepsilon\text{-}closure(move(T,a))$
        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}

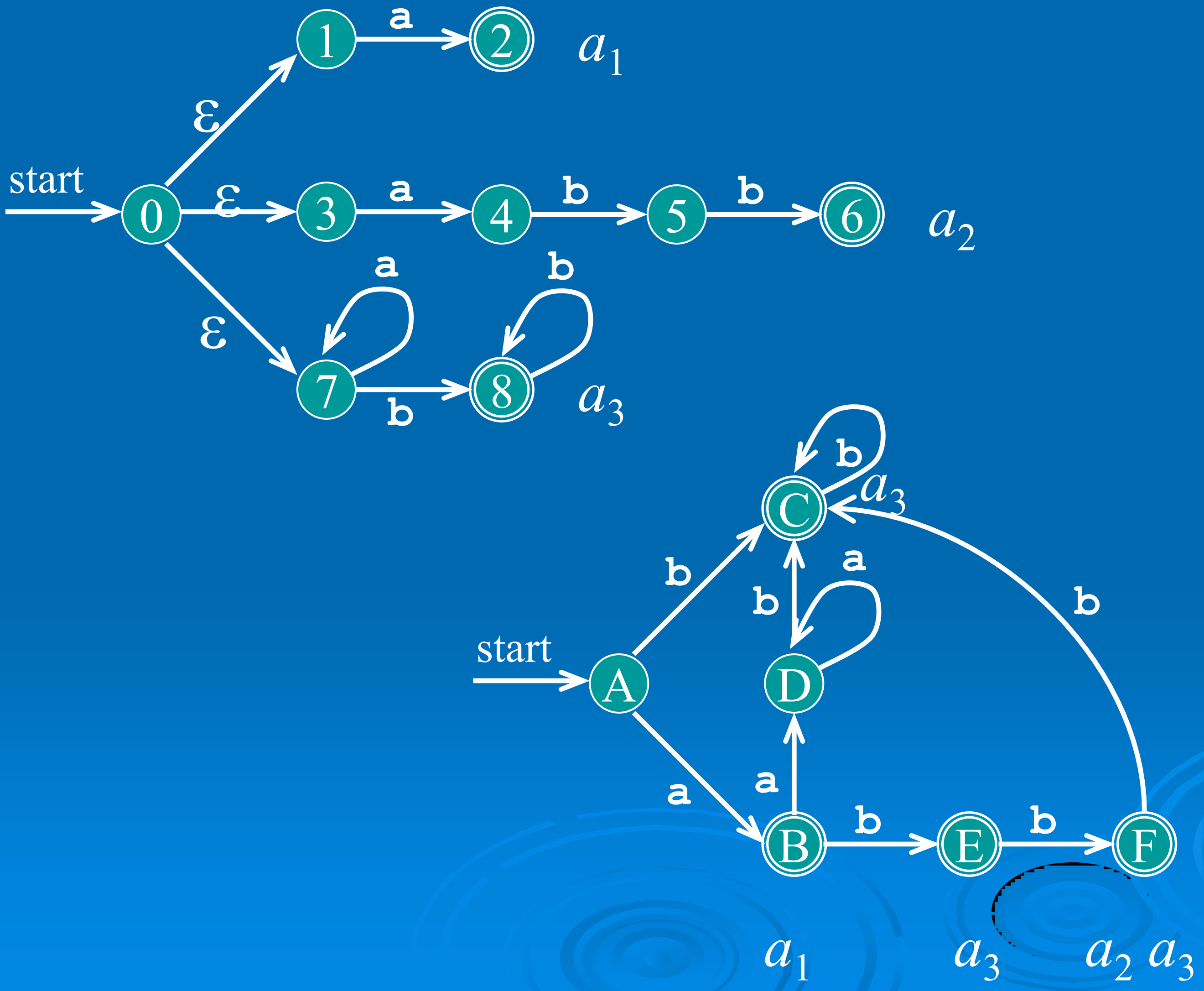
B = {1,2,3,4,6,7,8}

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}

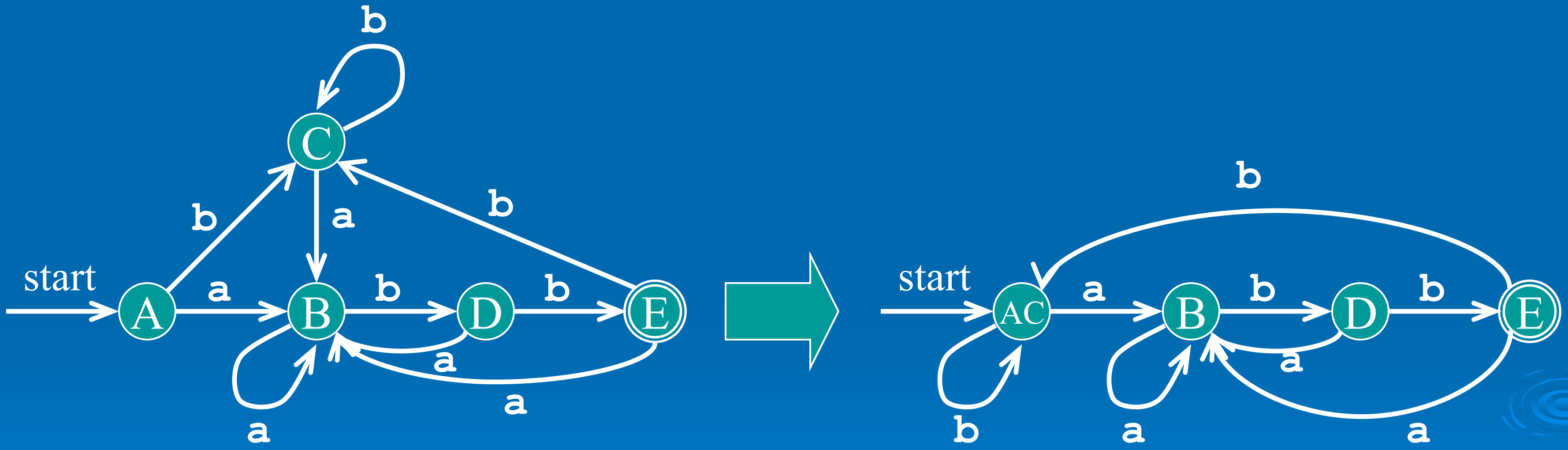
C = {8}

D = {7}

E = {5,8}

F = {6,8}

Minimizing the Number of States of a DFA



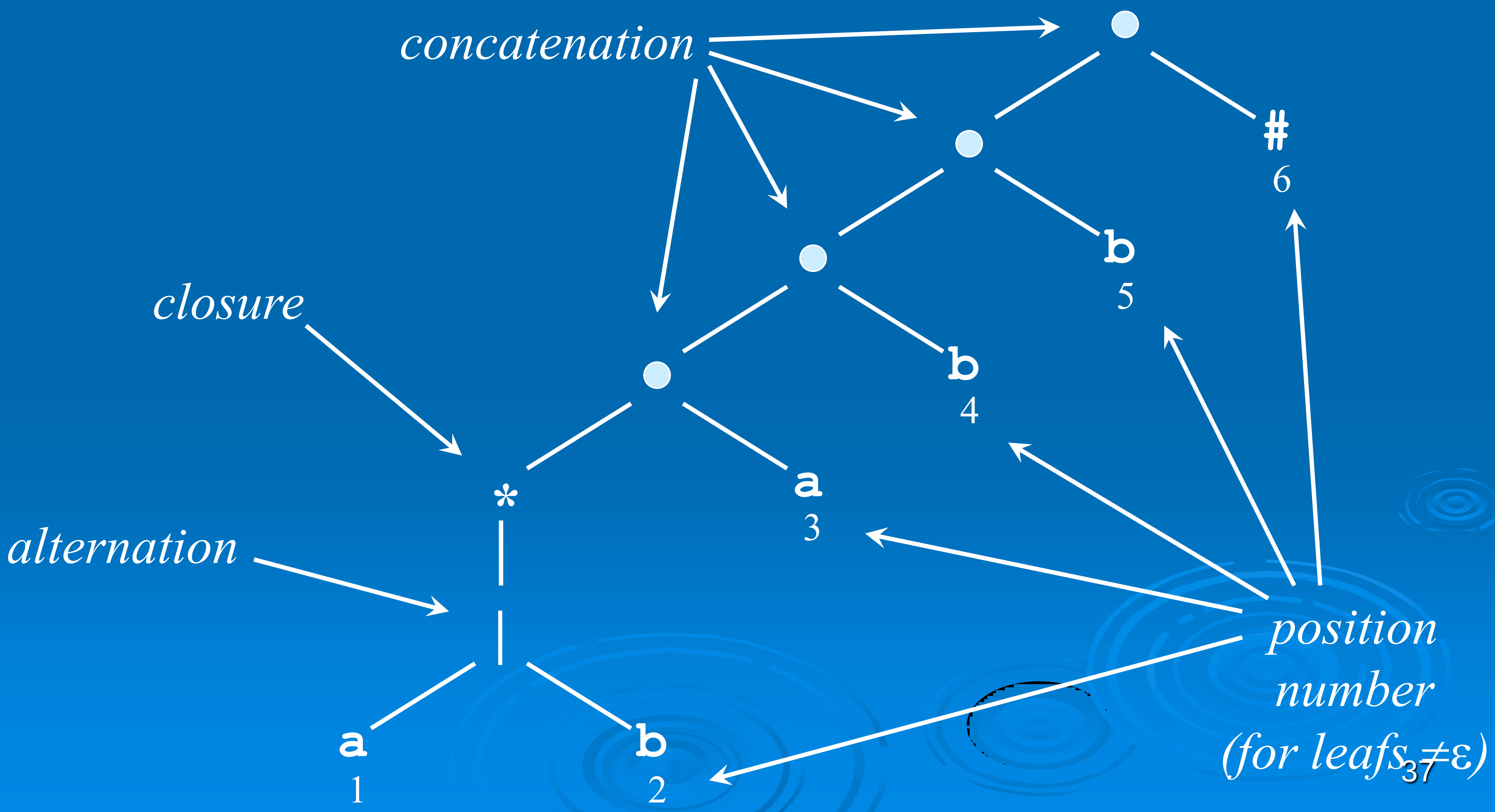
From Regular Expression to DFA Directly

- 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 *ε-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#`**



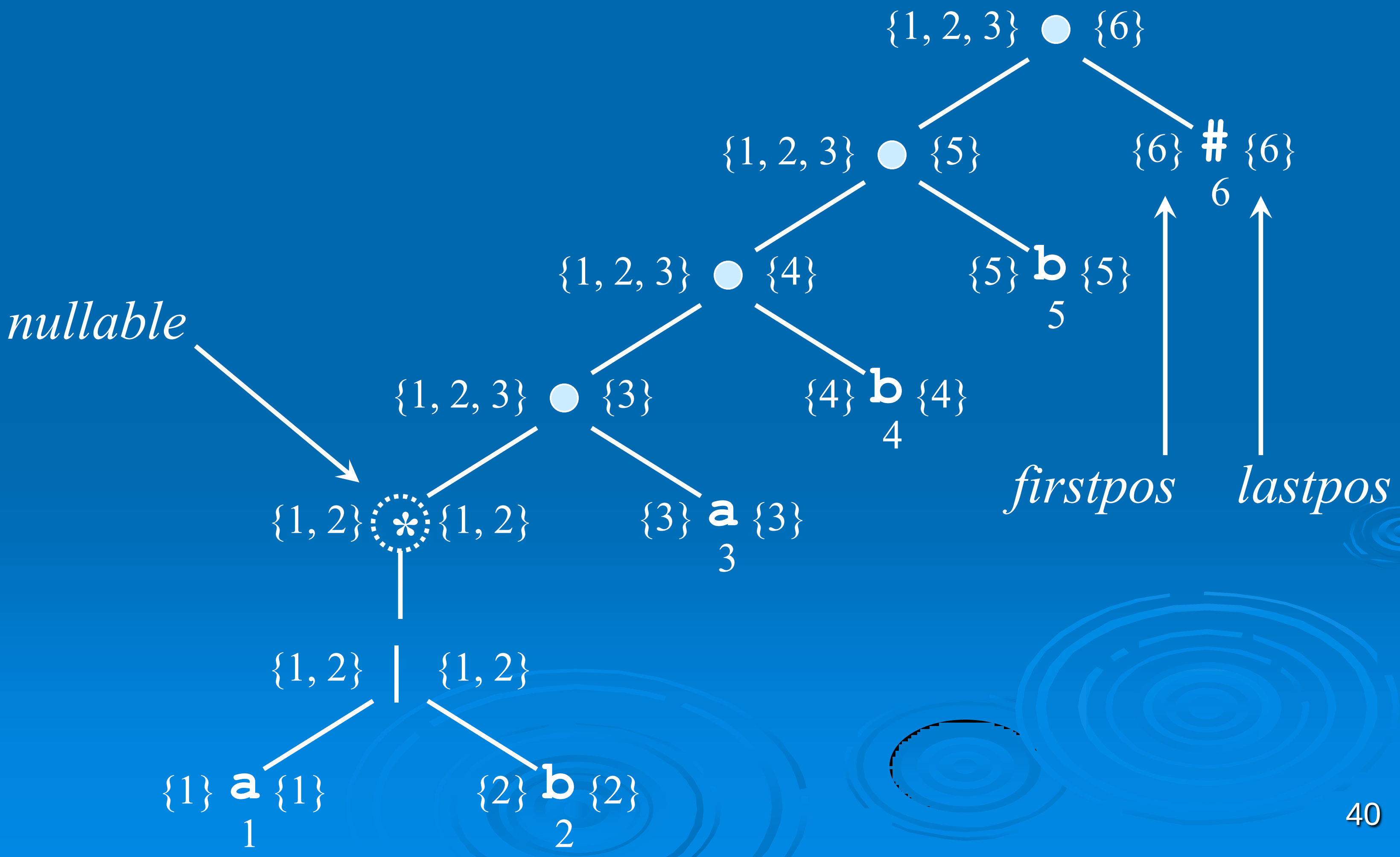
From Regular Expression to DFA Directly: Annotating the Tree

- *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	∅	∅
Leaf *i*	false	{*i*}	{*i*}
| / \\ c_1 c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
• / \\ c_1 c_2	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
* | 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*

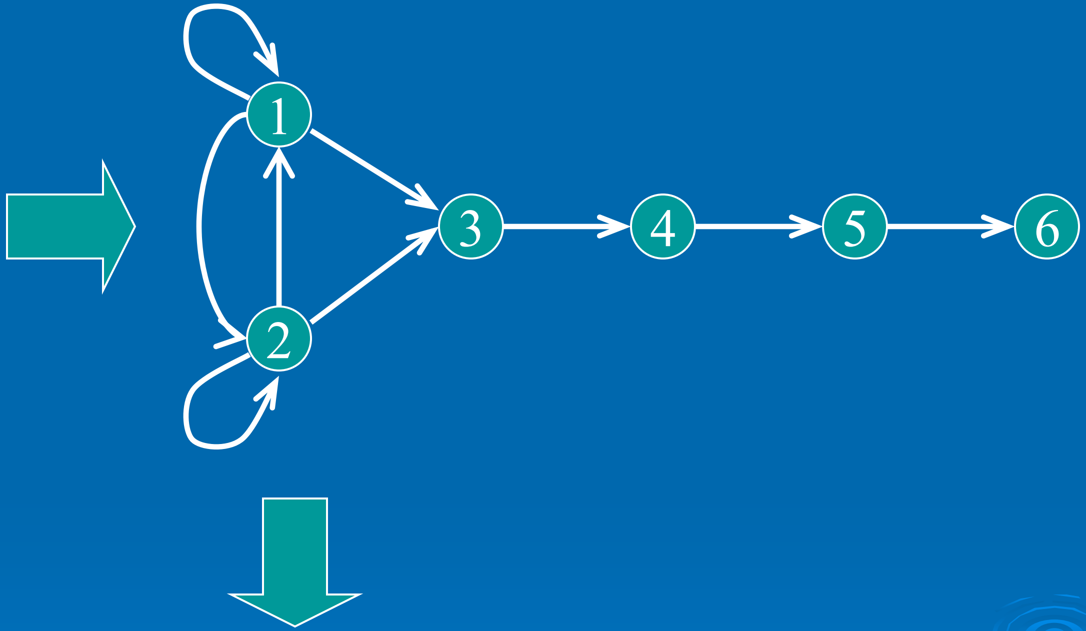
```
for each node n in the tree do
    if n is a cat-node with left child $c_1$ and right child $c_2$ then
        for each i in lastpos($c_1$) do
            followpos(i) := followpos(i) $\cup$ firstpos($c_2$)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
         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

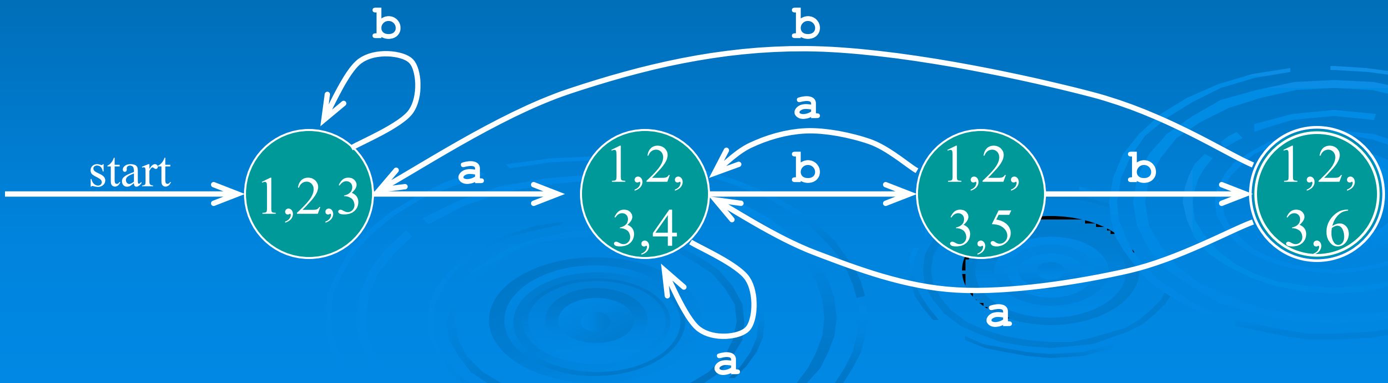
From Regular Expression to DFA Directly: Algorithm

```
$s_0$ := 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	$O(2^{\lvert r \rvert})$	$O(\lvert x \rvert)$

Thank you