CS 4300: Compiler Theory

Chapter 3
Lexical Analysis

Xuejun Liang

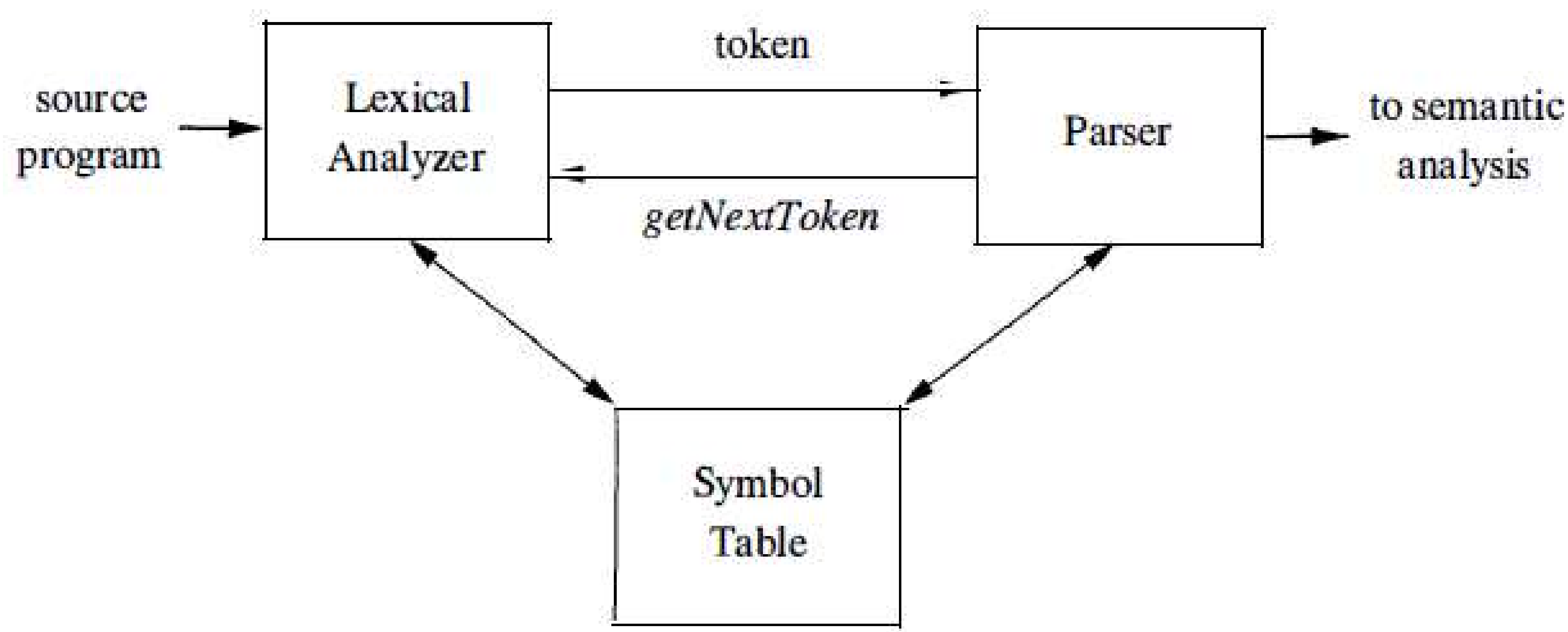
2019 Fall

Outlines (Sections)

1. The Role of the Lexical Analyzer
2. Input Buffering (Omit)
3. Specification of Tokens
4. Recognition of Tokens
5. The Lexical -Analyzer Generator Lex
6. Finite Automata
7. From Regular Expressions to Automata
8. Design of a Lexical-Analyzer Generator
9. Optimization of DFA-Based Pattern Matchers

1. The Role of the Lexical Analyzer

- As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.



Why Lexical Analysis and Parsing (Syntax Analysis) are Separate

- 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)

Tokens, Patterns, and Lexemes

- A *token* is a pair consisting of a token name and an optional attribute value
  - The token name is an abstract symbol representing a kind of lexical unit
  - 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*"

Examples of Tokens

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Token Classes:

1. One token for each keyword
2. Tokens for the operators
3. One token representing all identifiers
4. One or more tokens representing constants
5. Tokens for each punctuation symbol

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched
- Examples: lexemes, token names and associated attribute values for the following statements.

```
printf ( "Total = %d\n",  score ) ;
```

```
E = M * C ** 2
```

3. 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

String Operations

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

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

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

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

Example:
Compute
$L \cup D$
LD
D^4
D^*
$L(L \cup D)^*$
D^+

where
$L = \{A, B, ..., Z, a, b, ..., z\}$
and $D = \{0, 1, ... 9\}$

Regular Expressions Over Some Alphabet Σ

- 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 $L(s)$ respectively, then
  - $r \mid s$ is a regular expression denoting $L(r) \cup L(s)$
  - rs is a regular expression denoting $L(r)\, L(s)$
  - 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*

Algebraic laws for regular expressions

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Example 3.4 : Let Σ = {a, b}, what are languages denoted by The following regular expressions:

a|b, **(a|b)(a|b)**, **a***, **(a|b)***, **a|a*b**

Regular Definitions Over Some Alphabet Σ

- Regular definitions introduce a naming convention with name to regular expression bindings:

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

where:

  - Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
  - each r_i is a regular expression over

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

Regular Definitions: Examples

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

$$
\begin{aligned}
\mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} &\rightarrow \mathit{digit} \; \mathit{digit}^{*} \\
\mathit{optionalFraction} &\rightarrow \texttt{.} \; \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\
\mathit{number} &\rightarrow \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent}
\end{aligned}
$$

Numbers: 5280, 0.01234, 6.336E4, or 1.89E-4.

Regular Definitions: Extensions

- The following shorthands are often used:

One or more instances: +	$r^+ = rr^*$
Zero or one instance: ?	$r? = r \mid \varepsilon$
Character classes:	$[\mathtt{a\text{-}z}] = \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \ldots \mid \mathtt{z}$

- Examples:

$$\begin{array}{rcl} letter\_ & \rightarrow & [\mathtt{A\text{-}Za\text{-}z\_}] \\ digit & \rightarrow & [0\text{-}9] \\ id & \rightarrow & letter\_\ (\ letter \mid digit\ )^* \end{array}$$

$$\begin{array}{rcl} digit & \rightarrow & [0\text{-}9] \\ digits & \rightarrow & digit^+ \\ number & \rightarrow & digits\ (.\ \ digits)?\ (\ \mathtt{E}\ [+-]?\ \ digits\ )? \end{array}$$

4. Recognition of Tokens

Example 3.8: A Grammar for branching statements

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

The terminals of the grammar, which are **if**, **then**, **else**, **relop** , **id**, and **number**, are the names of tokens for lexical analyzer.

Patterns for tokens of Example 3.8

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Tokens, patterns, and attribute values

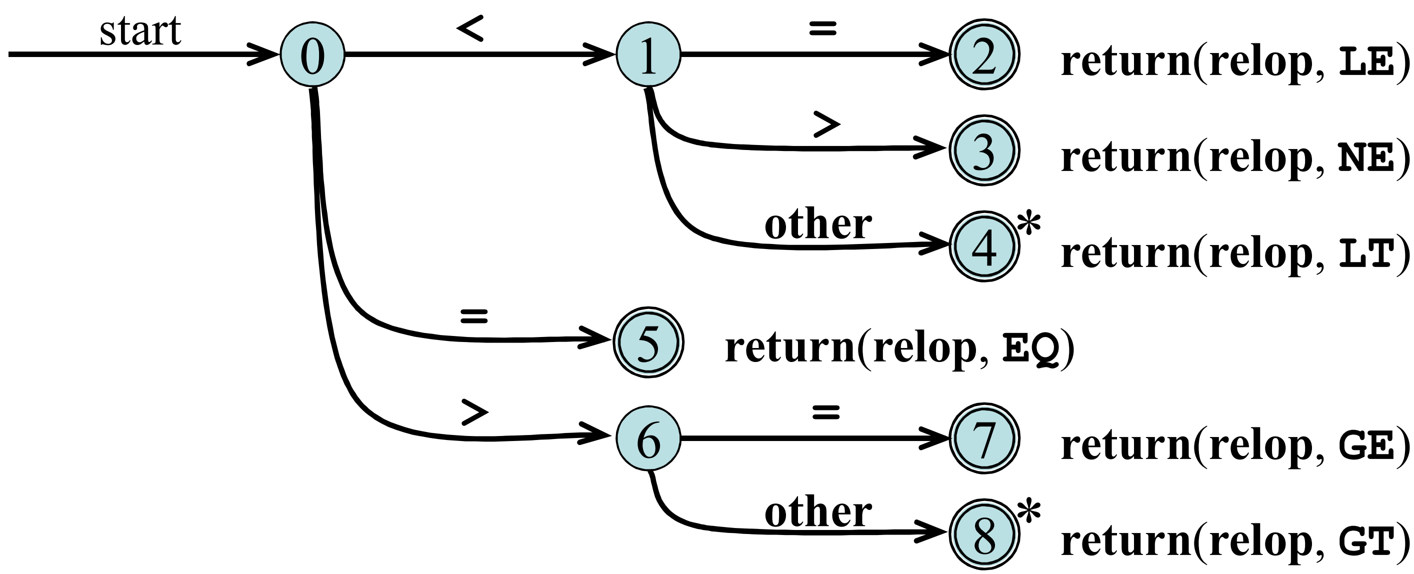
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

whitespace

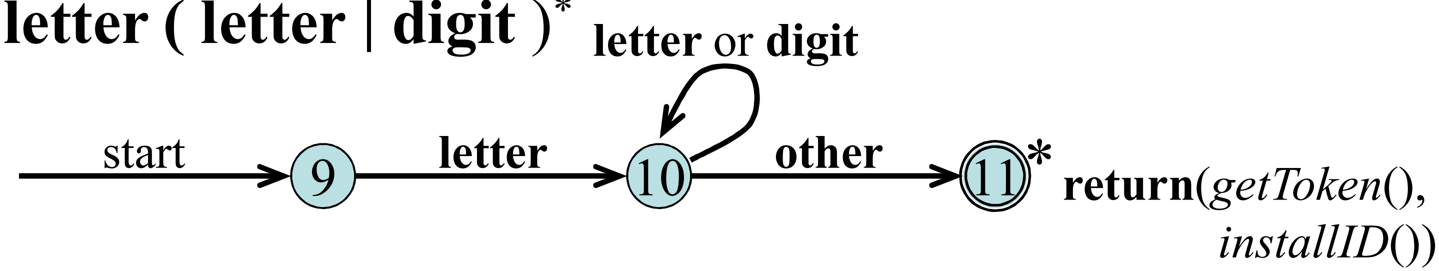
$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^+$$

Transition Diagrams

$\textbf{relop} \to \texttt{<} \mid \texttt{<=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=}$

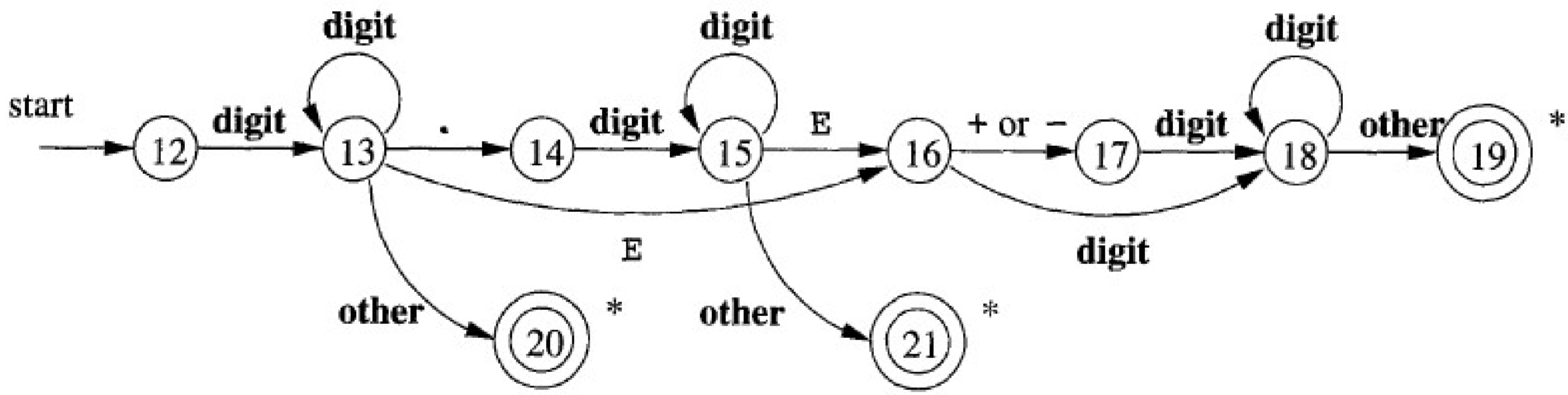


$\textbf{id} \to \textbf{letter} \ (\ \textbf{letter} \mid \textbf{digit} \ )^*$

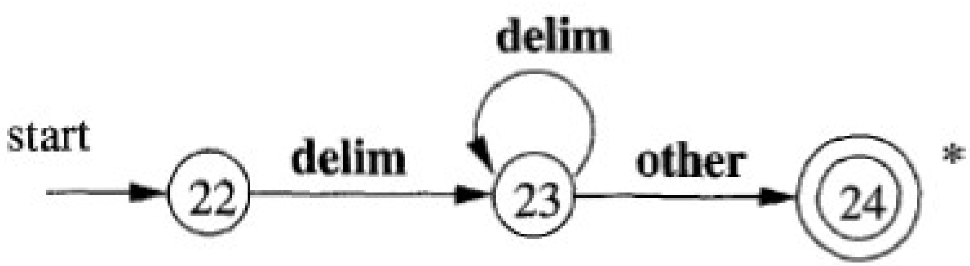


Transition Diagrams (Cont.)

Unsigned number



Whitespace



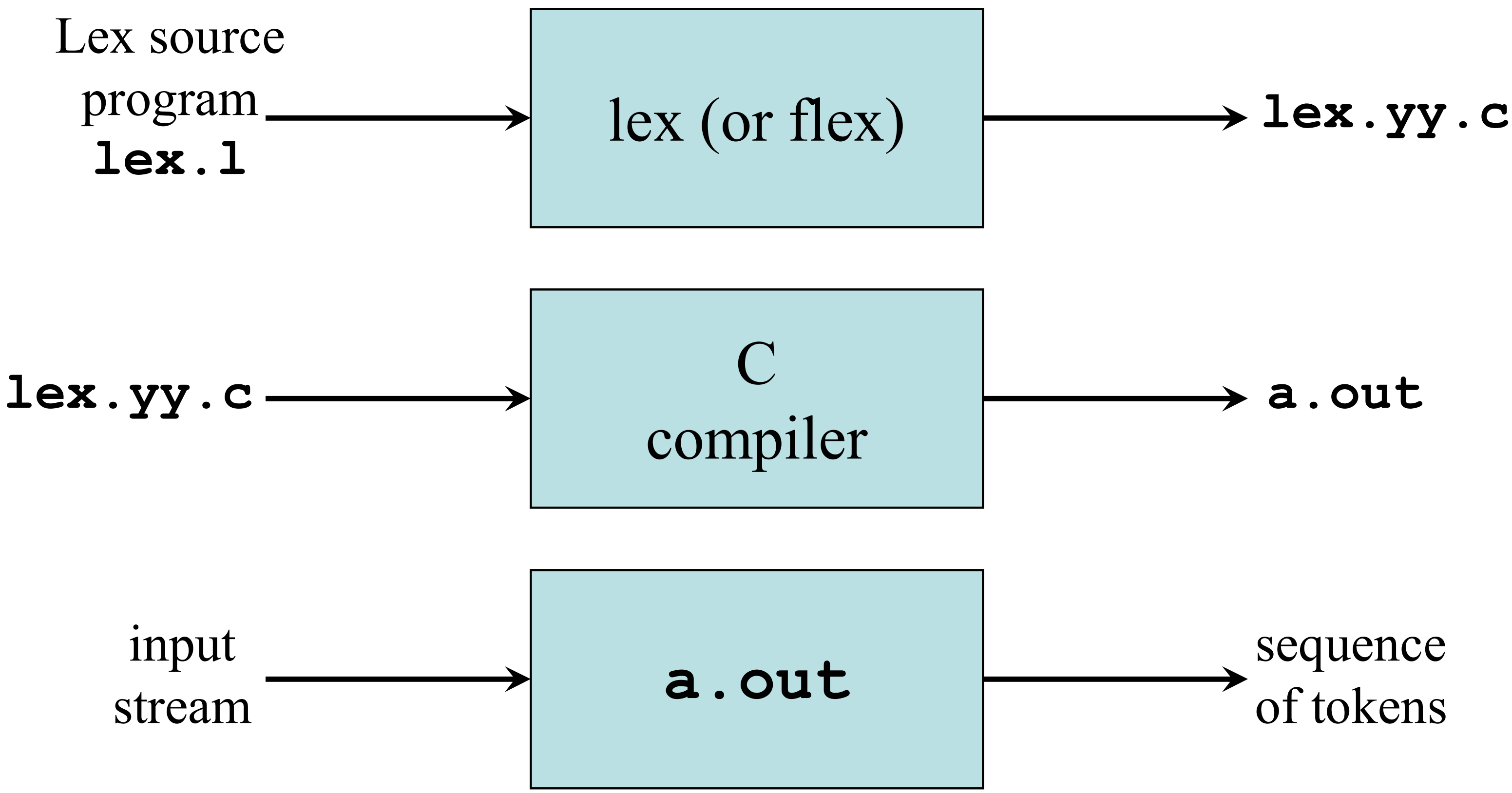
Sketch of implementation of relop transition diagram


```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

5. Lexical-Analyzer Generator: Lex and Flex

- *Lex* and its newer cousin *flex* are *scanner generators*
- 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



Structure of Lex Programs

- A Lex program consists of three parts:

  declarations
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures

- declarations
  - *C declarations in* `%{ %}`
  - *regular definitions*
- The translation rules are of the form:

  $pattern_1$ { $action_1$ }
  $pattern_2$ { $action_2$ }
  …
  $pattern_n$ { $action_n$ }

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

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();
}
```

Lex Specification: Example 3.8

```
%{ /* 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

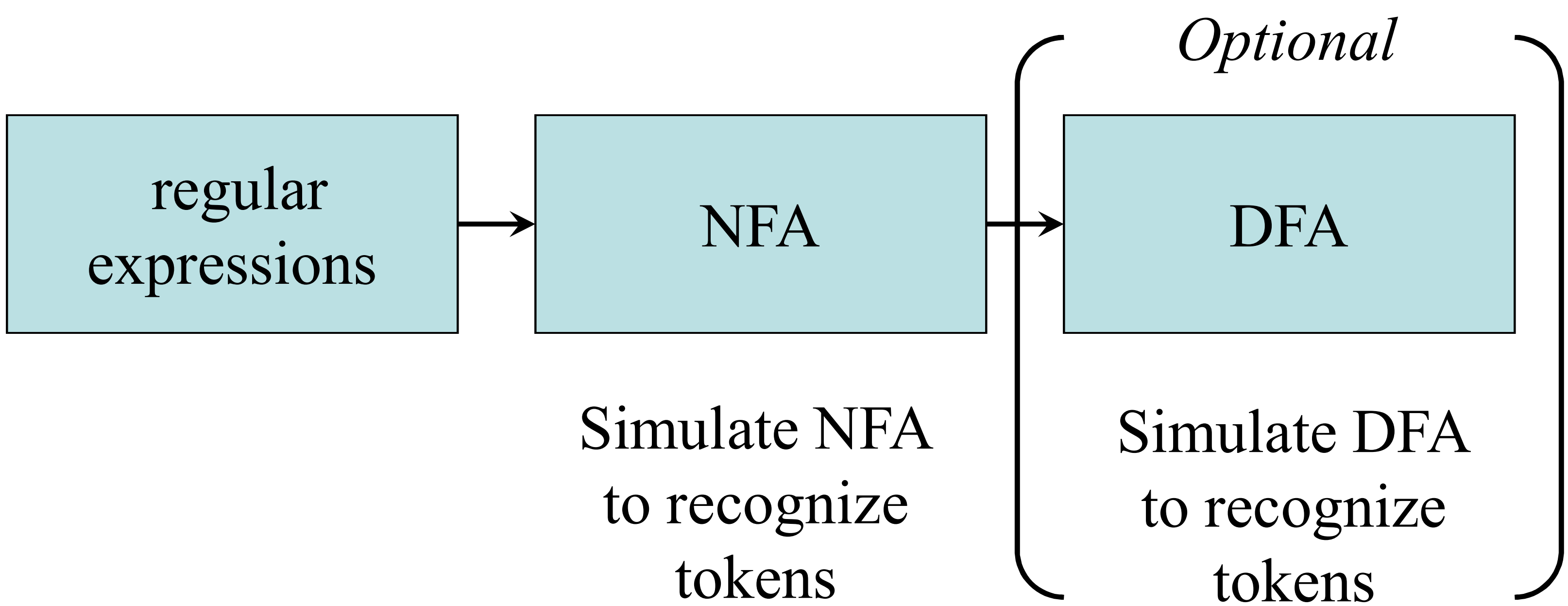
Install **`yytext`** as identifier in symbol table

Conflict Resolution in Lex

- Two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:
  1. Always prefer a longer prefix to a shorter prefix.
  2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

6. Finite Automata

- Design of a Lexical Analyzer Generator
  - Translate regular expressions to NFA
  - Translate NFA to an efficient DFA



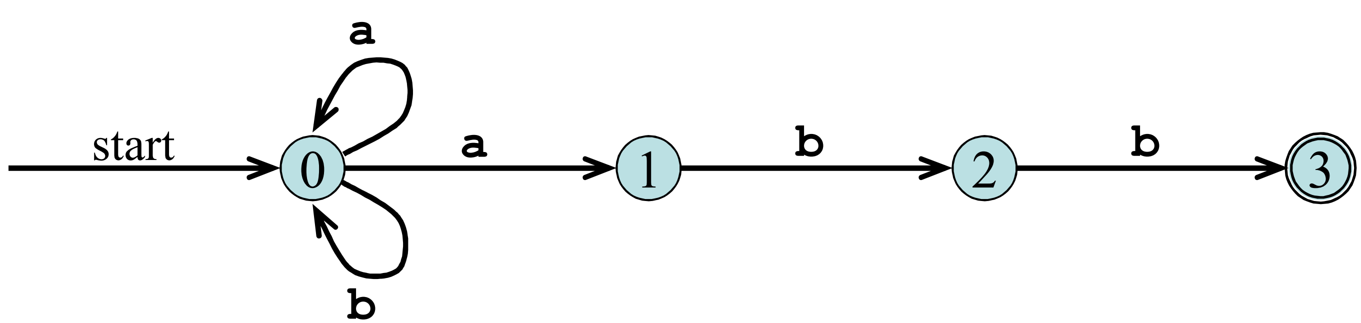
Nondeterministic Finite Automata

- 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*
- Example
  - an NFA recognizing the language of regular expression (**a**|**b**) * **abb**



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

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

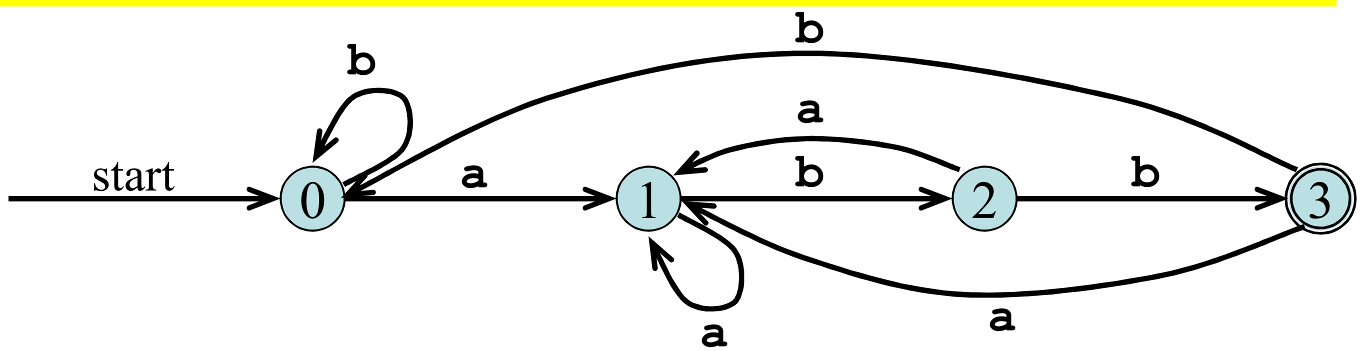
Deterministic Finite Automata

- A deterministic finite automaton (DFA) is a special case of 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

Simulating a DFA

$s = s_0$;
$c = nextChar()$;
while (c != **eof**) {
    $s = move(s, c)$;
    $c = nextChar()$;
}
if (s is in F) **return "yes"**;
else return "no";

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



7. From Regular Expressions to Automata
Conversion of an NFA into a DFA

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

The Subset Construction Algorithm

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

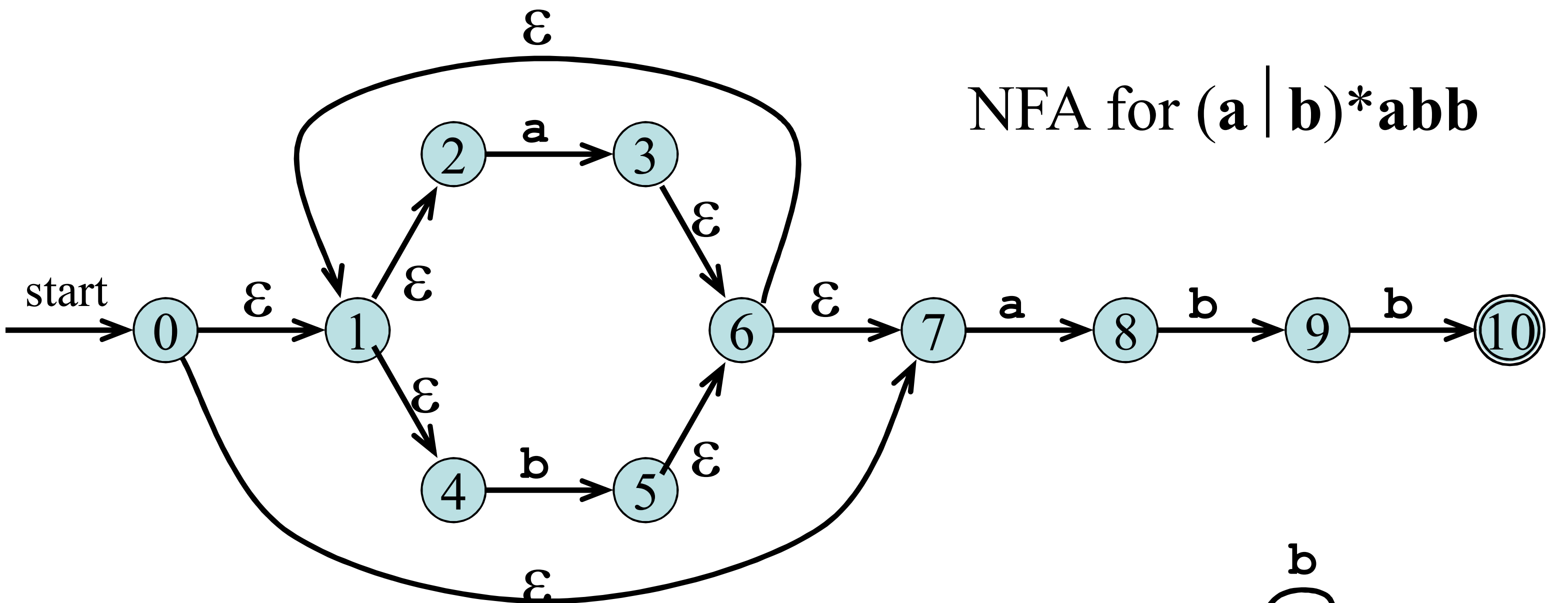
while (there is an unmarked state *T* in *Dstates)* **{**
    mark *T*
    for (each input symbol $a \in \Sigma$) {
        U = ε-*closure*(*move*(*T*,*a*))
        if (*U* is not in *Dstates)*
            add *U* as an unmarked state to *Dstates*
        Dtran[*T*,*a*] := *U*
    }
}

Computing ε-closure(*T*)

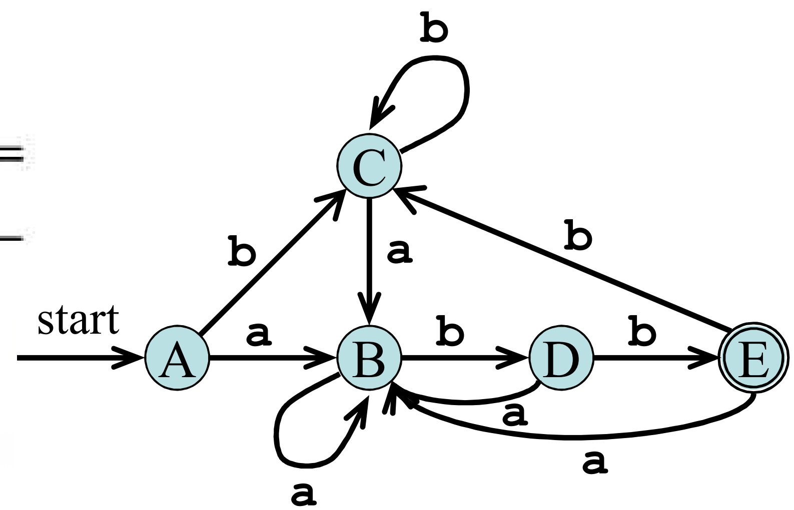
```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
            if ( u is not in ε-closure(T) ) {
                    add u to ε-closure(T) ;
                    push u onto stack;
            }
}
```

Subset Construction Example 1

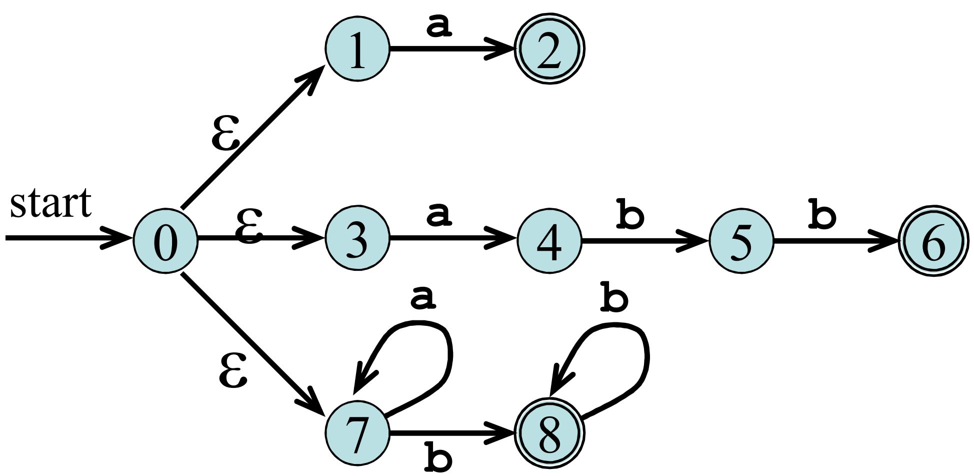
NFA for **(a | b)*abb**

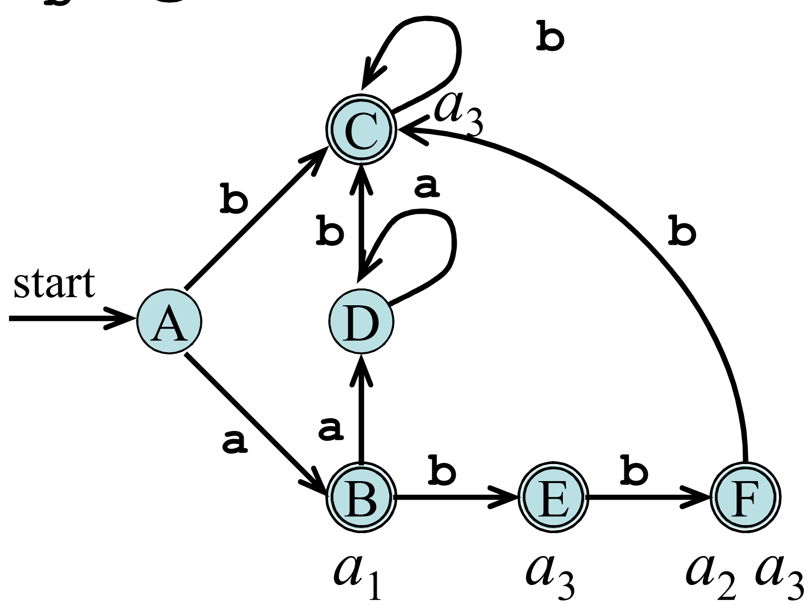


NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C



Subset Construction Example 2





Dstates

A = {0,1,3,7}

B = {2,4,7}

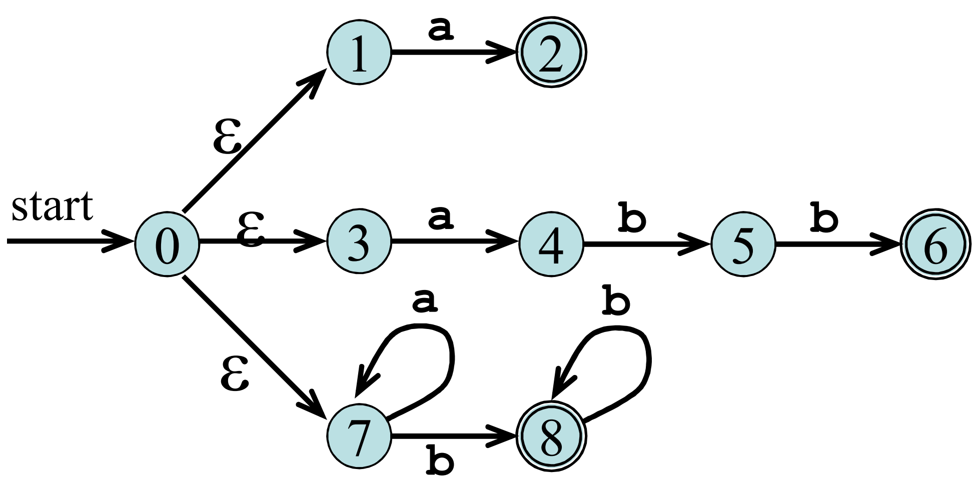
C = {8}

D = {7}

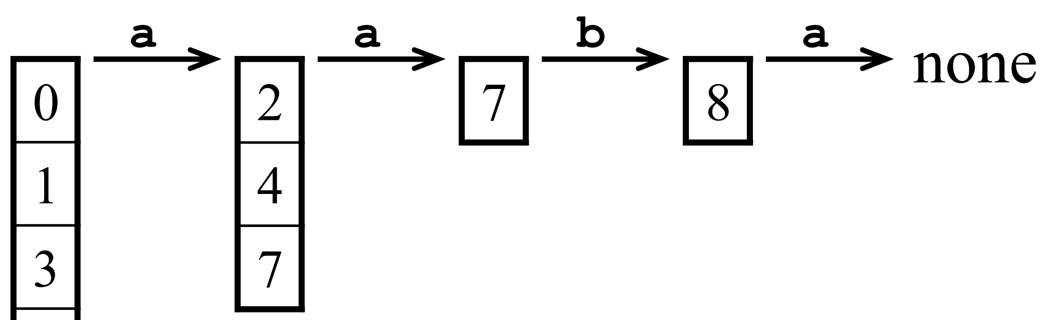
E = {5,8}

F = {6,8}

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**) = ∅

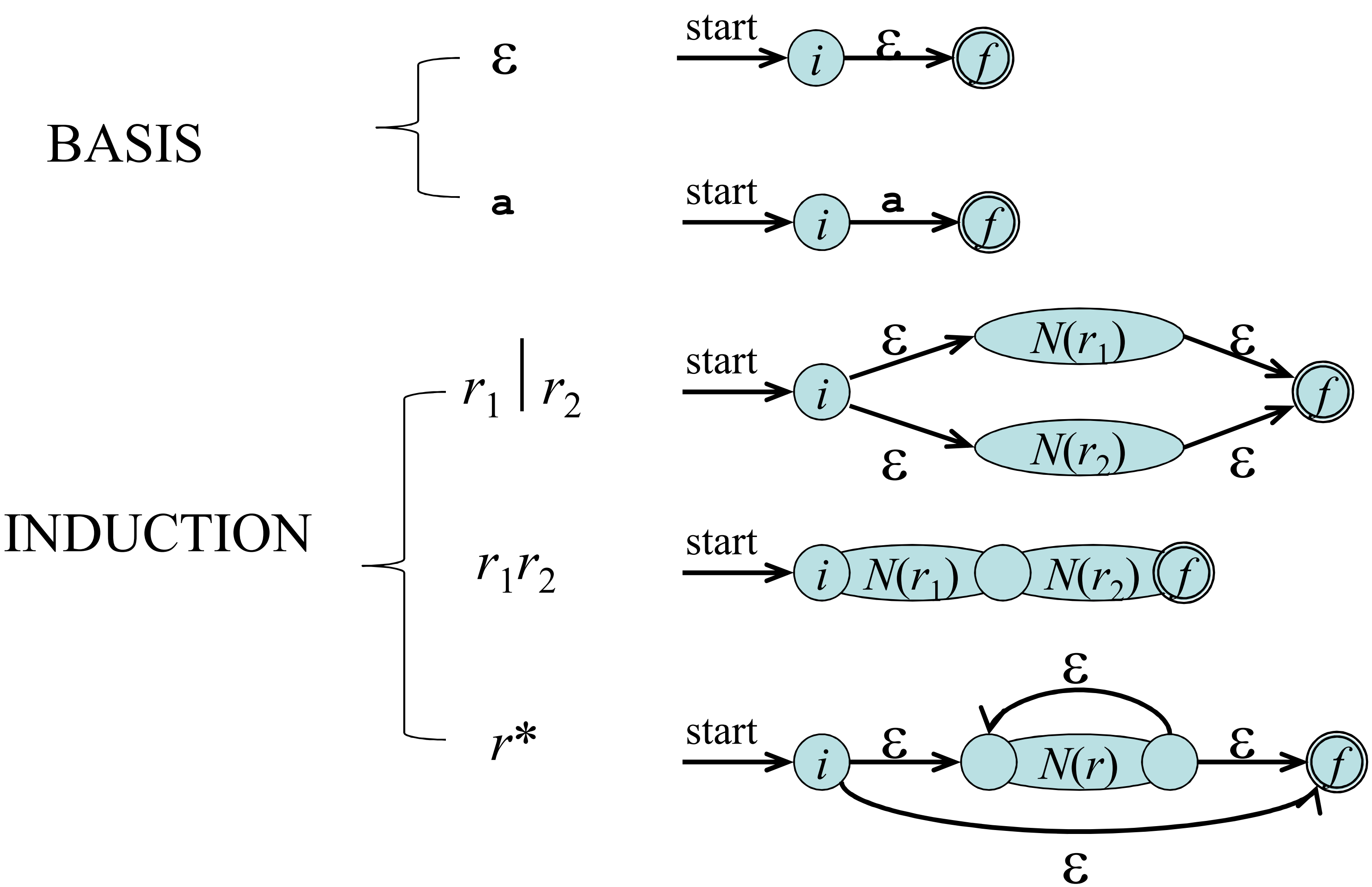


Also used to simulate NFAs (!)

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

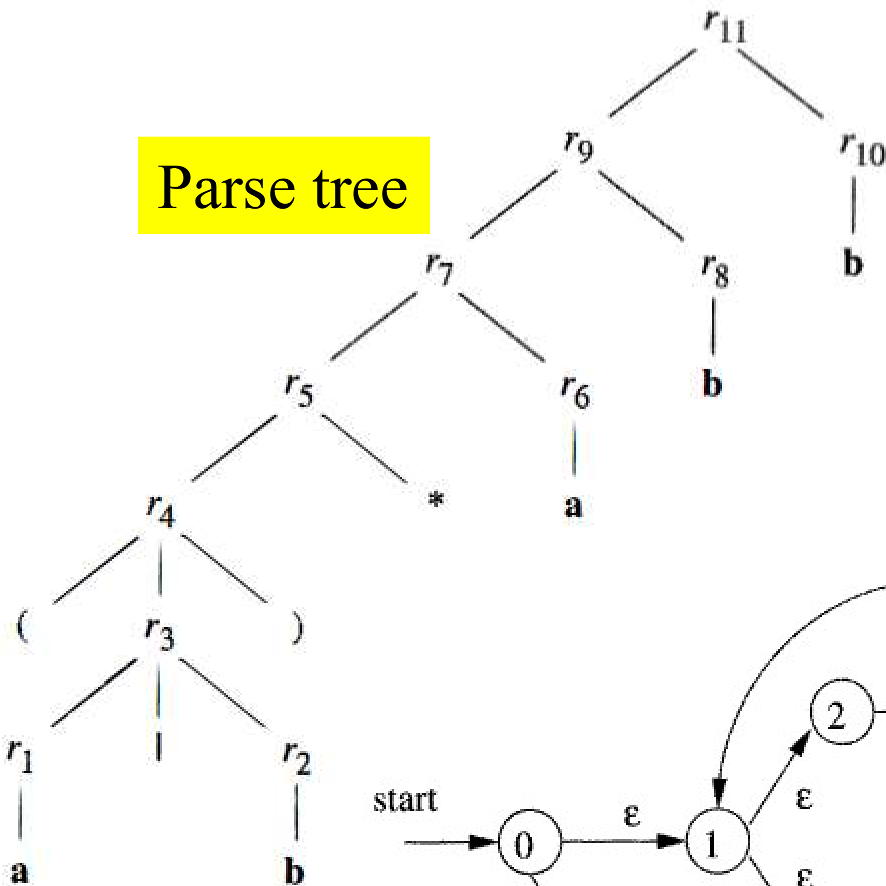
$S = \epsilon\text{-}closure(s_0);$
$c = nextChar();$
while (c != **eof**) {
    $S = \epsilon\text{-}closure(move(S, c));$
    $c = nextChar();$
}
if ($S \cap F \neq \emptyset$) **return "yes";**
else return "no";

From Regular Expression to NFA (Thompson’s Construction)

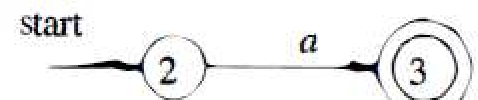


Construct an NFA for r = (a|b)*abb

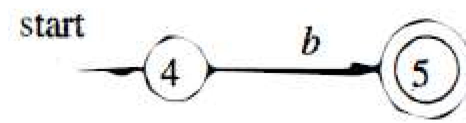
Parse tree

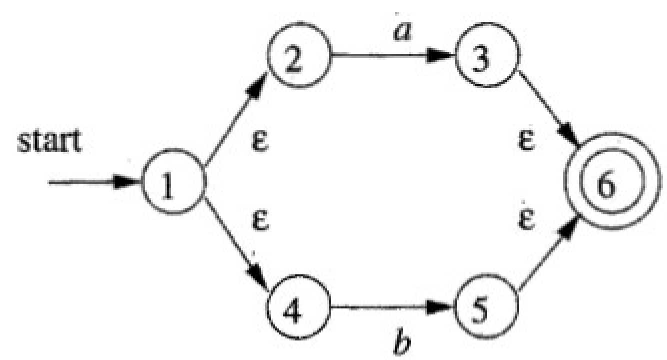


$r_1 = a$



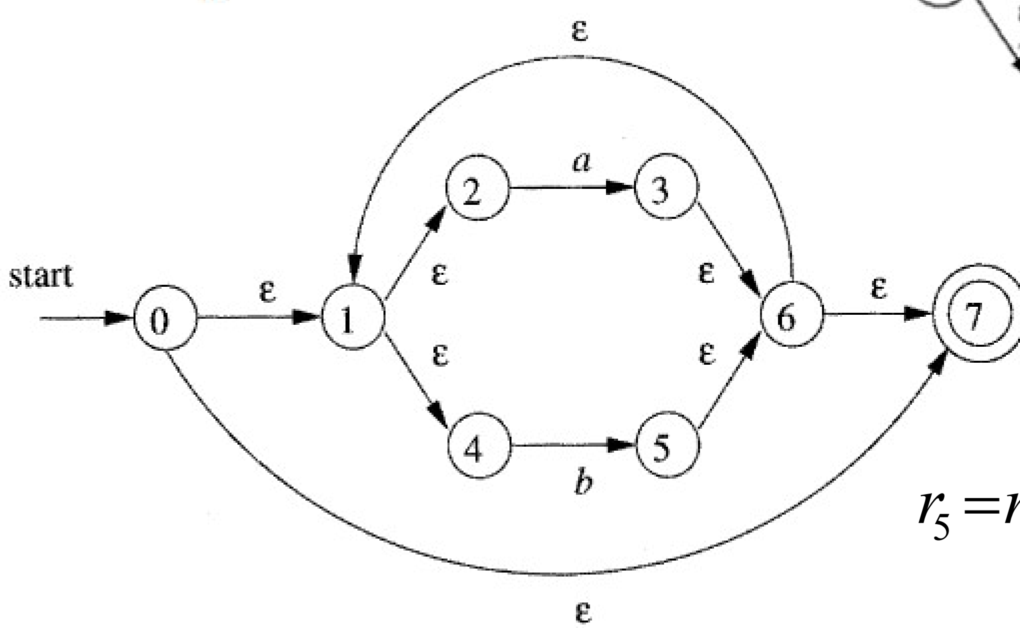
$r_2 = b$





$r_3 = r_1 / r_2$

$r_4 = (r_3)$



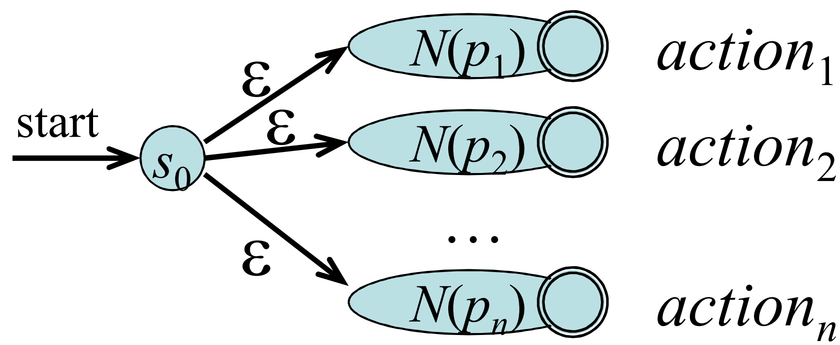
$r_5 = r_4 *$

8. Design of a Lexical-Analyzer Generator Construct an NFA from a Lex Program

Lex specification with regular expressions

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

NFA

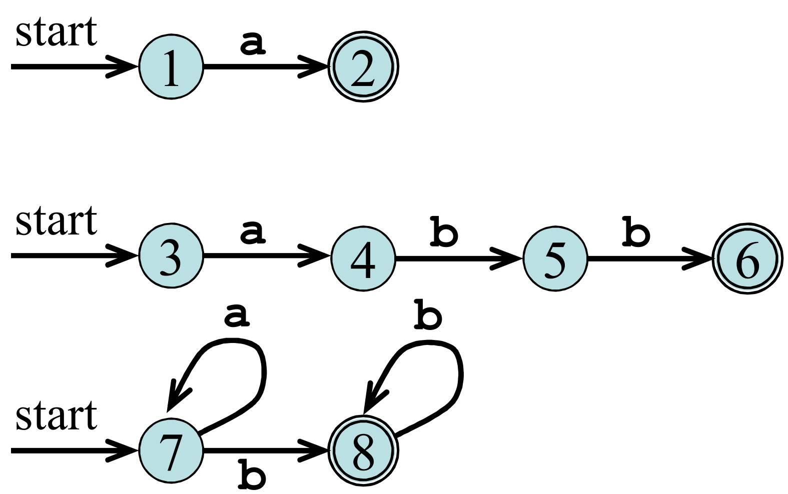


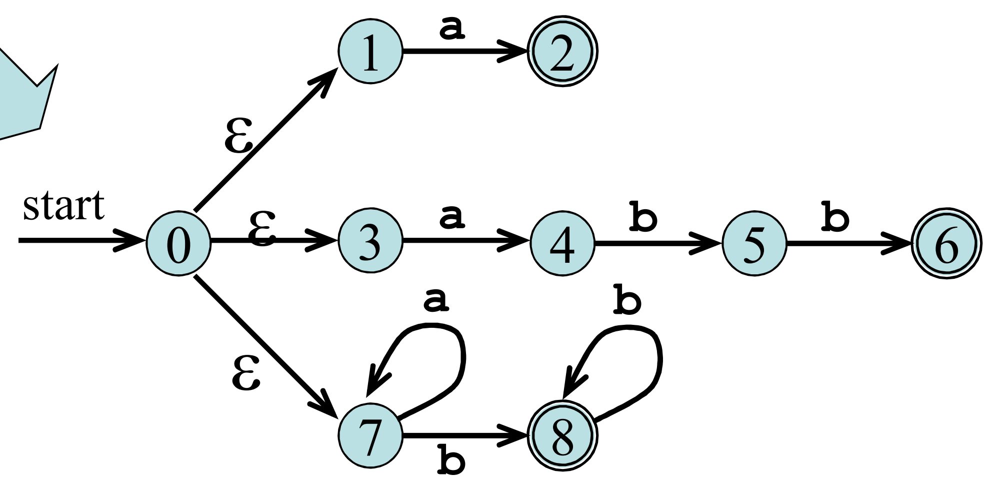
Subset construction

DFA

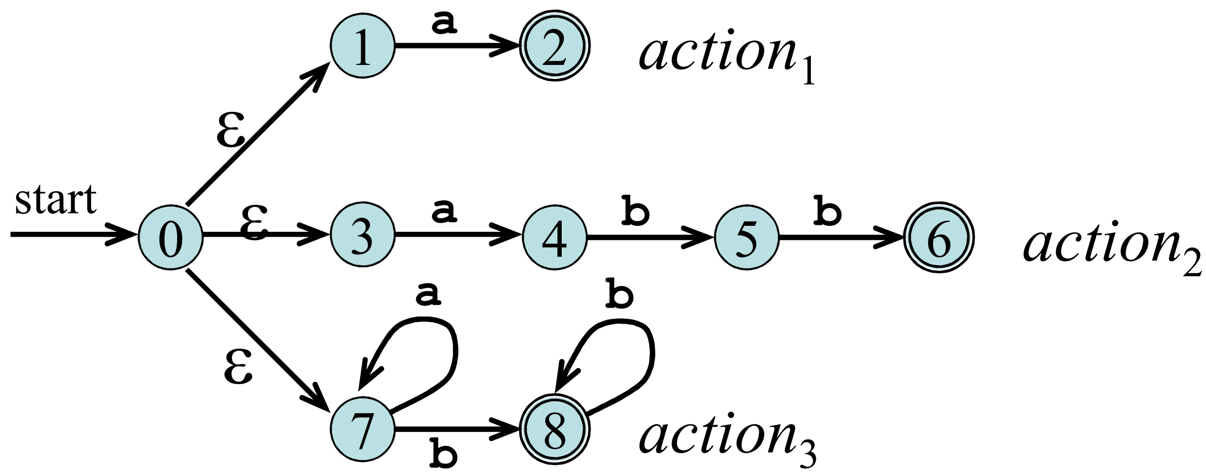
Combining the NFAs of a Set of Regular Expressions

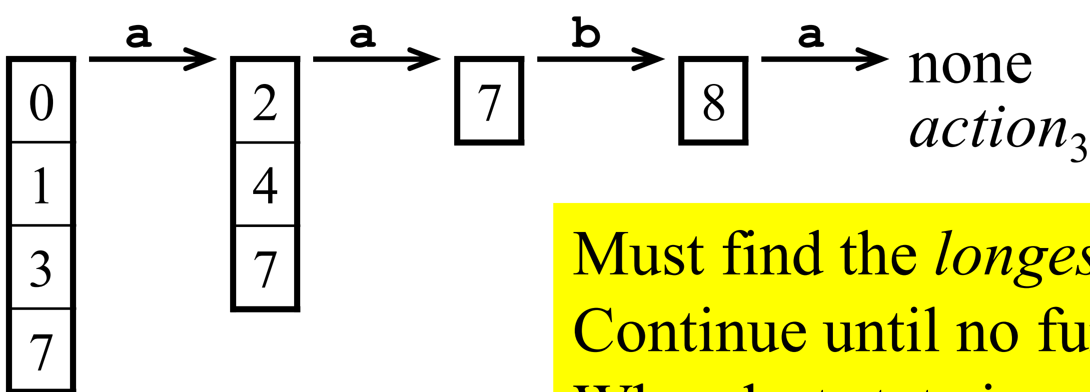
a { $action_1$ }
abb { $action_2$ }
a*b+ { $action_3$ }





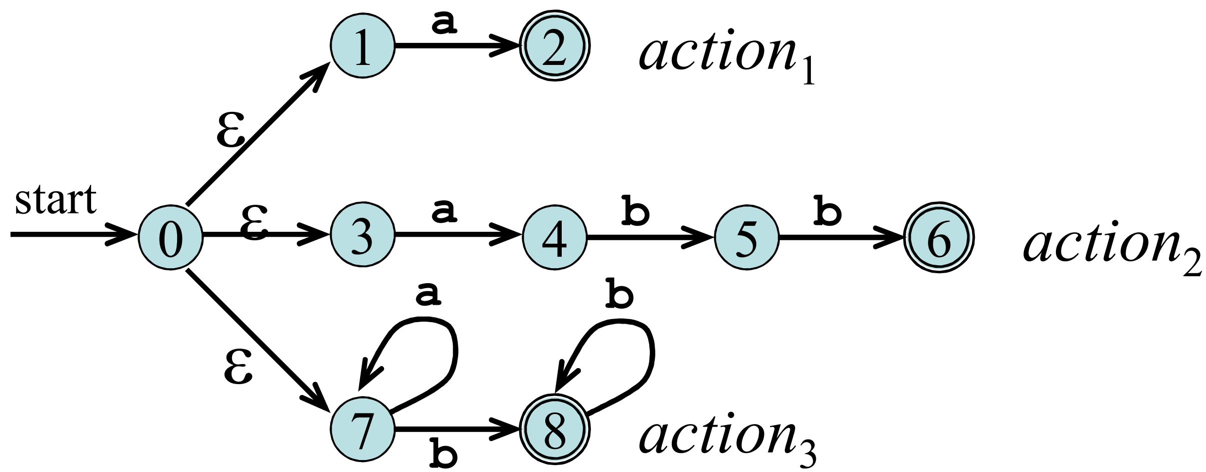
Simulating the Combined NFA Example 1

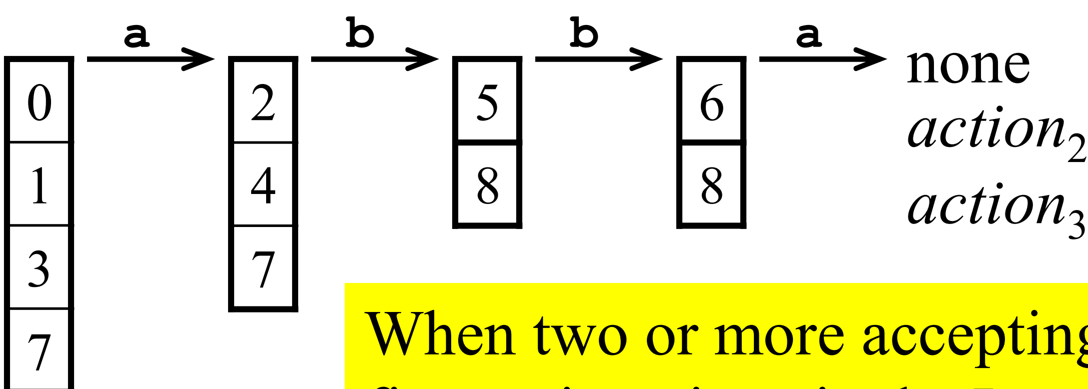




Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

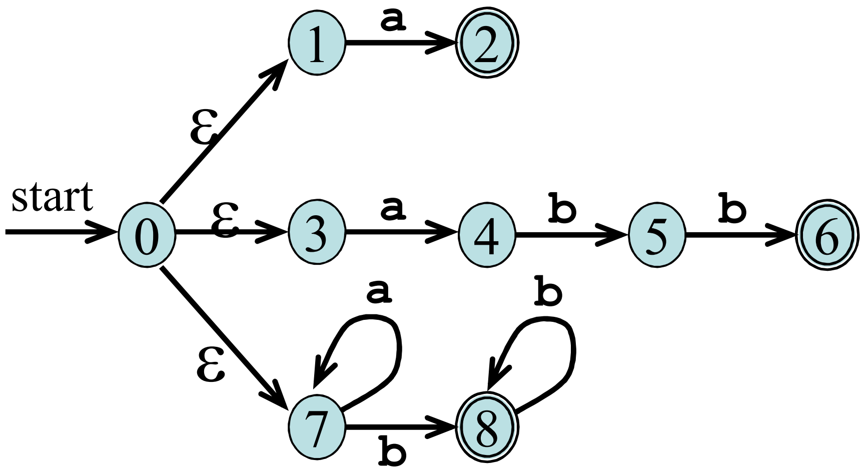
Simulating the Combined NFA Example 2





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

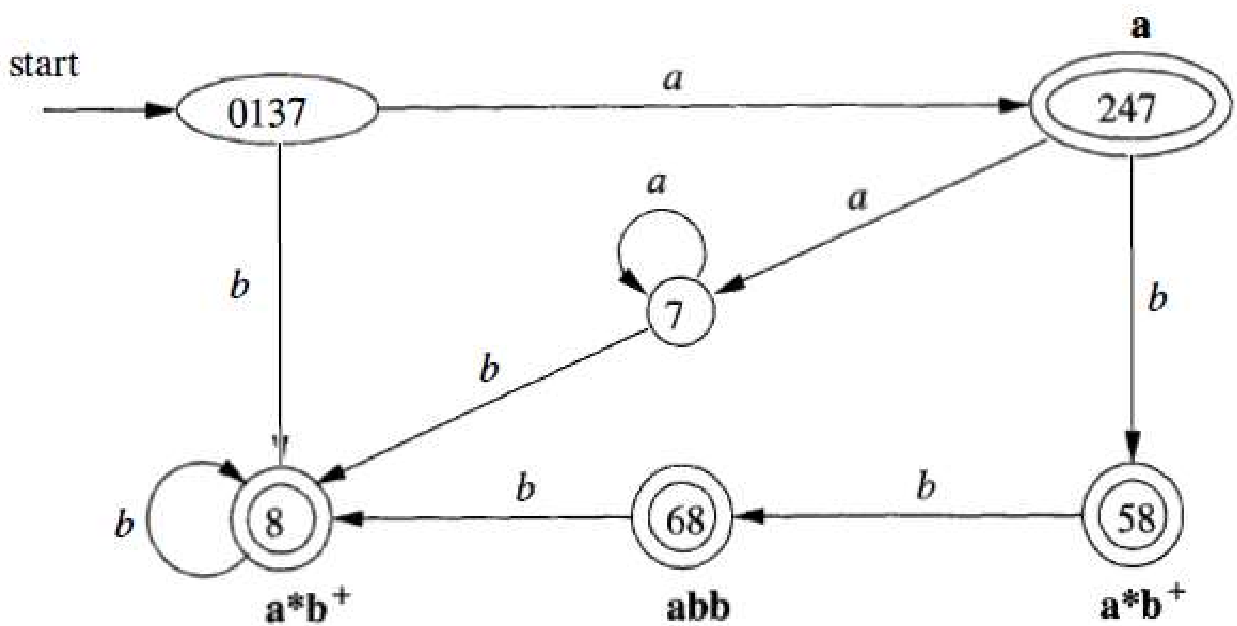
DFA's for Lexical Analyzers



NFA

Subset construction

DFA



Examples

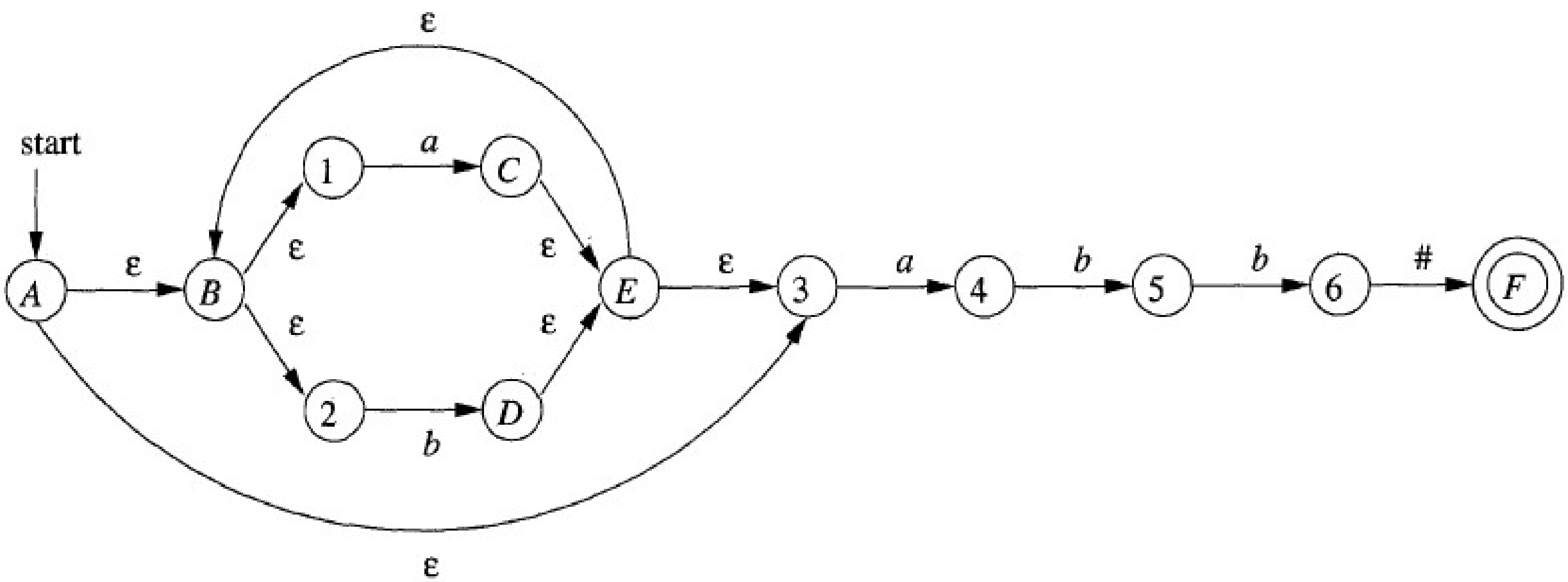
a → a → b → a →

a → b → b → a →

9. From RE 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 $\varepsilon\text{-}closure(move(T,a))$

NFA Constructed for (**a**|**b**)***abb#**



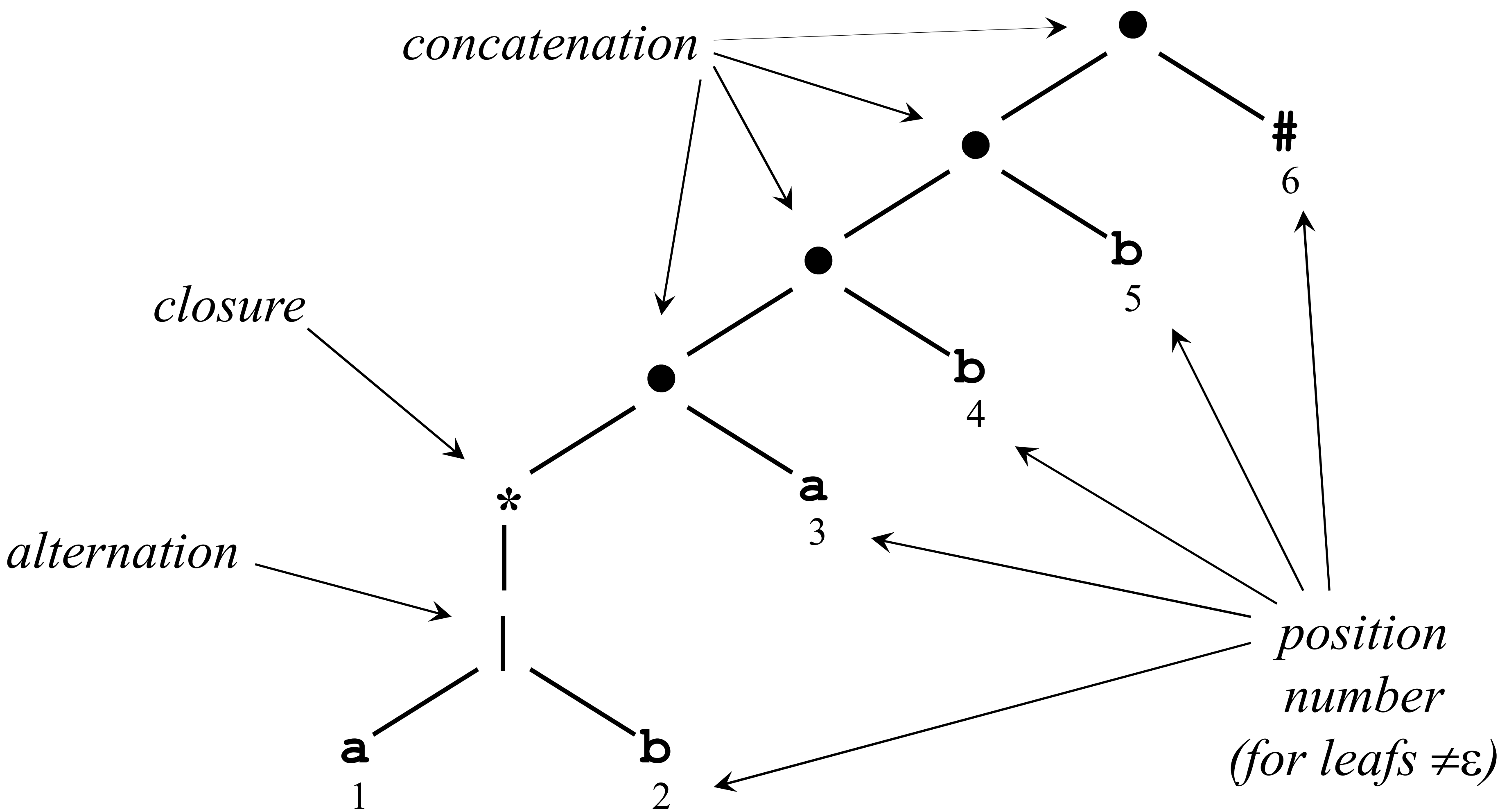
Note:
1. The NFA is constructed by Thompson's Algorithm
2. The important states in the NFA are numbered

Algorithm:

INPUT : A regular expression r.
OUTPUT: A DFA D that recognizes L(r) .

- Augment the regular expression *r* with a special end symbol # to make accepting states important: the new expression is *r#*
- Construct a syntax tree T from *r#*
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
- Construct *Dstates*, the set of states of DFA D, and *Dtran*, the transition function for D.
- The start state of D is *firstpos*(n_0), where node n_0 is the root of T. The accepting states are those containing the position for the end marker symbol #.

Syntax Tree of (**a**|**b**)***abb#**



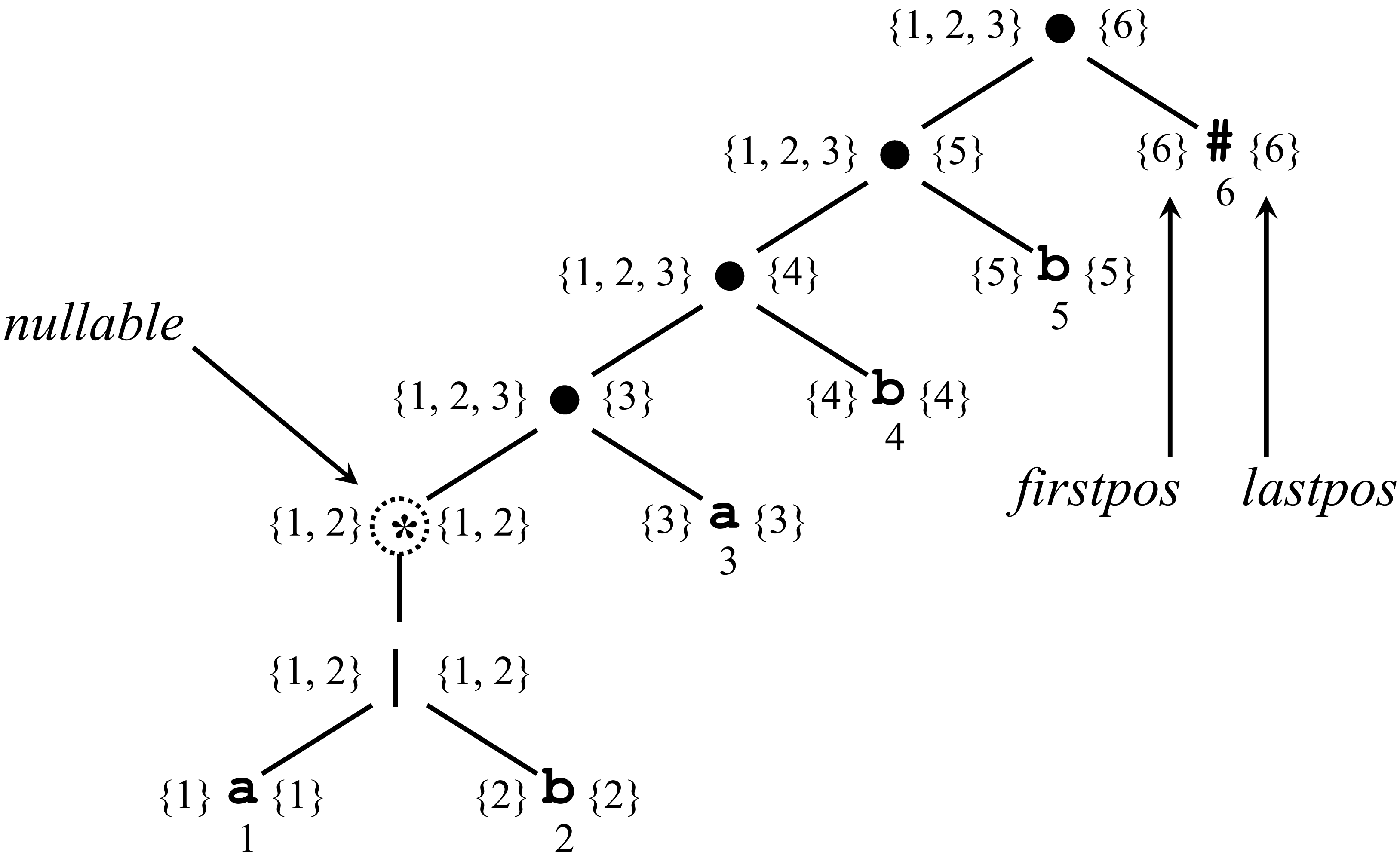
Annotating the Syntax Tree

- *nullable*(*n*): is true for a syntax-tree node *n* if and only if the subexpression represented by *n* has ε in its language.
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subexpression represented by node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the subexpression represented by node *n*
- *followpos*(*p*): the set of positions that can follow position *p* in the syntax-tree

Annotating the Syntax Tree (Cond.)

Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
| / \ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
• / \ 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** $lastpos(c_2)$
* | c_1	true	$firstpos(c_1)$	$lastpos(c_1)$

Annotated Syntax Tree of (a|b)*abb#



Algorithm: *followpos*

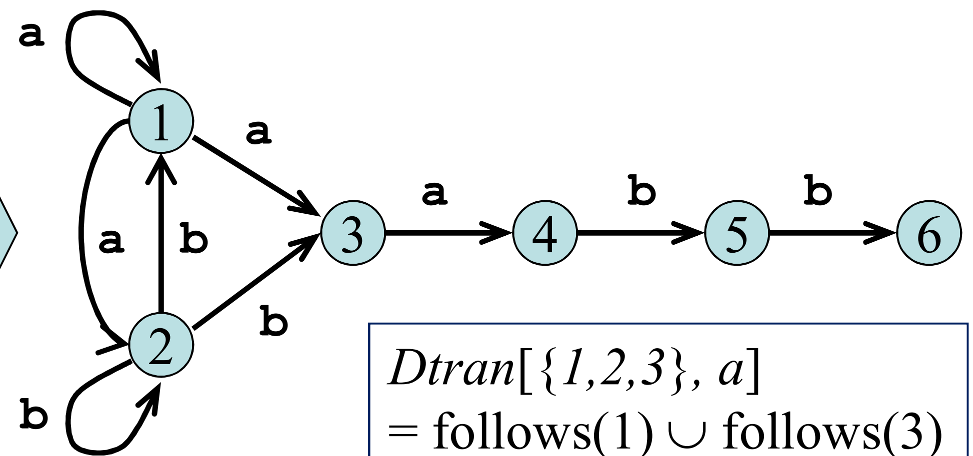
for each node n in the tree **{**
    if n is a cat-node with left child c_1 and right child c_2
        for each i in $lastpos(c_1)$ **{**
            $followpos(i) := followpos(i) \cup firstpos(c_2)$
        }
    else if n is a star-node
        for each i in $lastpos(n)$ **{**
            $followpos(i)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

Algorithm: Construct *Dstates*, and *Dtran*

s_0 = *firstpos*(n_0) where n_0 is the root of the syntax tree
Dstates := {s_0} and s_0 is unmarked
while (there is an unmarked state *S* in Dstates) {
    mark *S;*
    for each input symbol $a \in \Sigma$ **{**
        let U be the union of *followpos*(p) for all p
            in S that correspond to a;
        if (*U* not in *Dstates*)
            add *U* as an unmarked state to *Dstates*
        Dtran[*S,a*] = *U*
    }
}

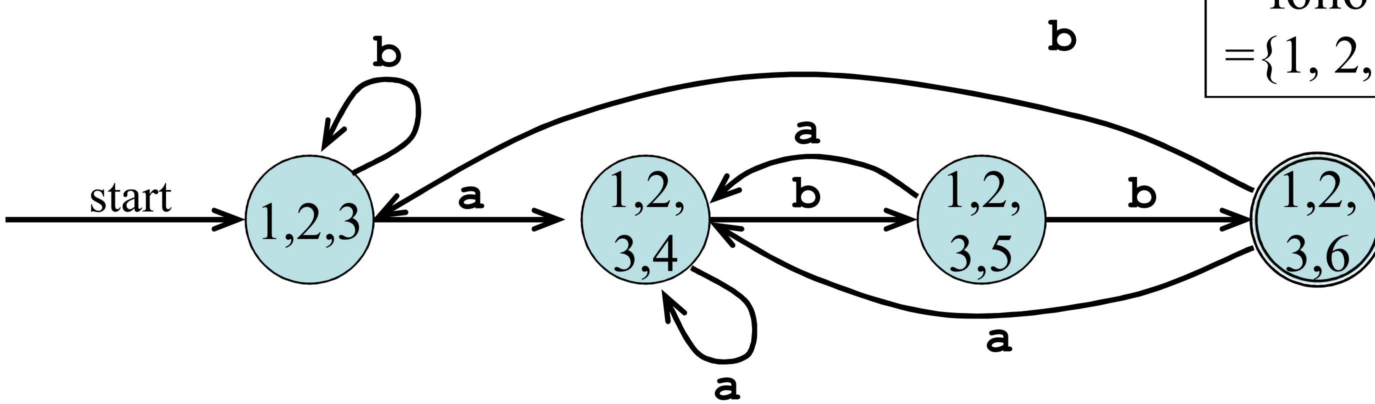
From RE to DFA Directly: Example

Node	*followpos*
1(a)	{1, 2, 3}
2(b)	{1, 2, 3}
3(a)	{4}
4(b)	{5}
5(b)	{6}
6(#)	-



Dtran[{*1,2,3*}, *a*]
= follows(1) ∪ follows(3)
={1, 2, 3, 4}

Dtran[{*1,2,3*}, *b*]
= follows(2)
={1, 2, 3, 4}



Minimize the Number of States of a DFA