北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Chapter 2: Lexical Analysis

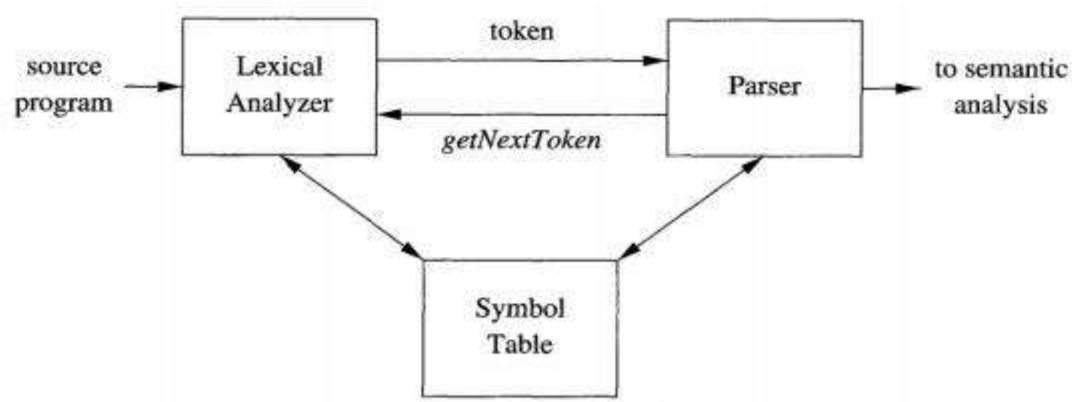
Yanhui Guo
yhguo@bupt.edu.cn

Outline

- **The Role of Lexical Analyzer**
- **Specification of Tokens (Regular Expressions)**
- **Recognition of Tokens (Transition Diagrams)**
- **The Lexical-Analyzer Generator**
- **Finite Automata**

The Role of Lexical Analyzer

- Read the input characters of the source program, group them into lexemes, and produces a sequence of tokens
- Add lexemes into the symbol table when necessary
- Strip out comments and whitespace (blank, newline, tab etc.)
- Associate error messages with a line number (tracking newlines)



Why Separated Lexical Analysis?

- Simplicity of compiler design. The lexical analyzer can perform simple tasks such as dealing with comments and whitespace

- Improved compiler efficiency. Lexical analysis is much simplercomparing to syntax analysis

- Higher portability of compilers. Only lexical analyzer needs to deal with input-device-specific peculiarities (e.g., line separator)

Tokens, Patterns, and Lexemes

- A ***token*** is a pair <token name, attribute value>
  - Token name is an abstract symbol representing a kind of lexical unit
  - Attribute value (optional) points to the symbol table
- A ***pattern*** is a description of the form that the lexemes of a token may take.
- A ***lexeme*** is a sequence of characters in the source program that matches the pattern for a token.
  - It is identified by the lexical analyzer as an instance of the token.

Examples

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"`

Consider the C statement: `printf("Total = %d\n", score);`

Lexeme	printf	score	"Total = %d\n"	(	...
Token	**id**	**id**	**literal**	**left_parenthesis**	**...**

Attributes for Tokens

- When more than one lexeme match a pattern, the lexical analyzer must provide additional information, named *attribute values*, to the subsequent compiler phases
  - Token names influence parsing decisions
  - Attribute values influence semantic analysis, code generation etc.
- For example, an **id** token is often associated with: (1) its lexeme,(2) type, and (3) the location at which it is first found. Token attributes are stored in the symbol table.

A = B * 2 →

<**id**, pointer to symbol-table entry for A>
<**assign_op**>
<**id**, pointer to symbol-table entry for B> <**mult_op**>
<**number**, integer value 2>

Lexical Errors

- The lexical analyzer is unable to proceed: none of the patterns for tokens match any prefix of the remaining input
- Example: int 3a = a * 3;

Outline

- The Role of Lexical Analyzer
- Specification of Tokens (Regular Expressions)
- Recognition of Tokens (Transition Diagrams)
- The Lexical-Analyzer Generator
- Finite Automata

Specification of Tokens

- Regular expression (正则表达式, regexp for short) is an important notation for specifying lexeme patterns

- Content of this part
  - Strings and Languages (串和语言)
  - Operations on Languages (语言上的运算)
  - Regular Expressions
  - Regular Definitions (正则定义)
  - Extensions of Regular Expressions

Strings and Languages

- Alphabet (字母表): any finite set of symbols
  - Examples of symbols: letters, digits, and punctuations
  - Examples of alphabets: {1, 0}, ASCII, Unicode
- A string (串) over an alphabet is a finite sequence of

symbols drawn from the alphabet

  - The length of a string s, denoted |s|, is the number of
- occurrences of symbols in s (i.e., cardinality)
  - Empty string (空串): the string of length 0, ε

Strings and Languages Cont.

- String-related terms (using banana for illustration)
  - Prefix (前缀) of string s: any string obtained by removing 0 or more symbols from the end of s (ban, banana, ϵ)
  - Proper prefix (真前缀): a prefix that is not ϵ and not equal to s itself (ban)
  - Suffix (后缀): any string obtained by removing 0 or more symbols from the beginning of s (nana, banana, ϵ).
  - Proper suffix (真后缀): a suffix that is not ϵ and not equal to s itself (nana)

Strings and Languages Cont.

- String-related terms (using banana for illustration)
  - Substring (子串) of s: any string obtained by removing any prefix and any suffix from s (banana, nan, ϵ)
  - Proper substring (真子串): a substring that is not ϵ and not equal to s itself (nan)
  - Subsequence (子序列): any string formed by removing 0 or more not necessarily consecutive symbols from s

How may substrings does banana have?

Strings and Languages Cont.

- String-related operations (串的运算)
  - Concatenation (连接): the concatenation of two strings x and y, denoted xy, is the string formed by appending y to x
    - x = dog, y = house, xy = doghouse
  - Exponentiation (幂/指数运算): $s^0=\varepsilon, s^1=s, s^i=s^{i-1}s$
    - x = dog, $x^0 = \varepsilon$, x^1 = dog, x^3 = dogdogdog

Strings and Languages Cont.

- A language (语言) is any countable set[1] of strings over some fixed alphabet
  - The set containing only the empty string, that is $\{\epsilon\}$, is a language, denoted $\emptyset$
  - The set of all grammatically correct English sentences
  - The set of all syntactically well-formed C programs

[1] In mathematics, a countable set is a set with the same cardinality (number of elements) as some subset of the set of natural numbers. A countable set is either a finite set or a countably infinite set.

Strings and Languages Cont.

- Operations on Languages (语言的运算)
  - 并，连接，Kleene闭包，正闭包

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

Strings and Languages Cont.

- Operations on Languages (Example)
  - $L = \{A, B, \ldots, Z, a, b, \ldots, z\}$
  - $D = \{0, 1, \ldots, 9\}$
  - $L \cup D = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
  - LD: the set of 520 strings of length two, each consisting of one letter followed by one digit
  - L^4: the set of all 4-letter string
  - L^*: the set of all strings of letters, including ϵ
  - $L(L \cup D)^*$: ?
  - D^+: ?

Regular Expressions

Rules that define regexps over an alphabet Σ:

- BASIS: two rules form the basis:
  - ϵ is a regexp, $L(\epsilon) = \{\epsilon\}$
  - If a is a symbol in Σ, then a is a regexp, and $L(a) = \{a\}$
- INDUCTION: Suppose r and s are regexps denoting the languages $L(r)$ and $L(s)$
  - $(r)|(s)$ is a regexp denoting the language $L(r) \cup L(s)$
  - $(r)(s)$ is a regexp denoting the language $L(r)L(s)$
  - $(r)^*$ is a regexp denoting $(L(r))^*$
  - (r) is a regexp denoting $L(r)$. Additional parentheses do not change the language an expression denotes.

Regular Expressions Cont.

- Following the rules, regexps often contain unnecessary pairs of parentheses. We may drop some if we adopt the conventions:
  - Precedence: closure * > concatenation > union |
  - Associativity: All three operators are left associative, meaning that operations are grouped from the left, e.g., a | b | c would be interpreted as (a | b) | c
- Example: (a) | ((b)*(c)) = a | b*c

Regular Expressions Cont.

- Examples: Let $\Sigma = \{a, b\}$
  - $a|b$ denotes the language $\{a, b\}$
  - $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
  - a^* denotes $\{\epsilon, a, aa, aaa, \ldots\}$
  - $(a|b)^*$ denotes the set of all strings consisting of 0 or more a's or b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$
  - $a|a^*b$ denotes the string a and all strings consisting of 0 or more a's and ending in b: $\{a, b, ab, aab, aaab, \ldots\}$

Regular Expressions Cont.

- A regular language (正则语言) is a language that can be defined by a regexp
- If two regexps r and s denote the same language, they are equivalent, written as r = s
- Each algebraic law below asserts that expressions of two different forms are equivalent

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

(a|b)(a|b)

=

aa|ab|ba|bb

Is it true???

Regular Definitions

- For notational convenience, we can give names to certain regexps and use those names in subsequent expressions
- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\ldots \\
d_n &\rightarrow r_n
\end{aligned}
$$

where:

- Each d_i is a new symbol not in Σ and not the same as the other d's
- Each r_i is a regexp over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Regular Definitions - Examples

- Regexp for C identifiers

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

_hello valid?
3times valid?

- Regexp for C identifiers

(A|B|...|Z|a|b|...|z|_)((A|B|...|Z|a|b|...|z|_)|(0|1|...|9))*

Extensions of Regular Expressions

- **Basic operators**: union $|$, concatenation, and Kleene closure*(proposed by Kleene in 1950s)

- A few **notational extensions**:
  - One of more instances: the unary, postfix operator +
    - $r^+ = rr^*$, $r^* = r^+ \mid \epsilon$
  - Zero or one instance: the unary postfix operator ?
    - $r? = r \mid \epsilon$
  - Character classes: shorthand for a logical sequence
    - $[a_1a_2\ldots a_n] = a_1 \mid a_2 \mid \ldots \mid a_n$
    - $[a\text{-}e] = a \mid b \mid c \mid d \mid e$

- The extensions are only for notational convenience, they do not change the descriptive power of regexps

Outline

- The Role of Lexical Analyzer
- Specification of Tokens (Regular Expressions)
- Recognition of Tokens (Transition Diagrams)
- The Lexical-Analyzer Generator
- Finite Automata

Recognition of Tokens

- Lexical analyzer examines the input string and finds a prefix that matches one of the tokens
- The first thing when building a lexical analyzer is to define the patterns of tokens using regular definitions
- **A special token: ws → (blank | tab | newline)$^+$**
  - When the lexical analyzer recognizes the whitespace token, it does not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace (the next token gets returned to the parser)

Example: Patterns and Tokens

digit	→	[0-9]
digits	→	$digit^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

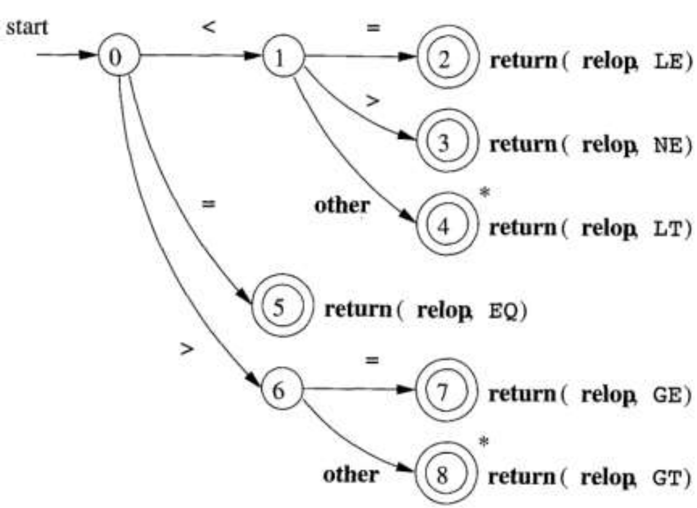
Patterns for tokens

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

Tokens, their patterns, and attribute values

Transition Diagrams (状态转换图)

- An important step in constructing a lexical analyzer is to convert patterns into "transition diagrams"
- Transition diagrams have a collection of nodes, called states (状态) and edges (边) directed from one node to another

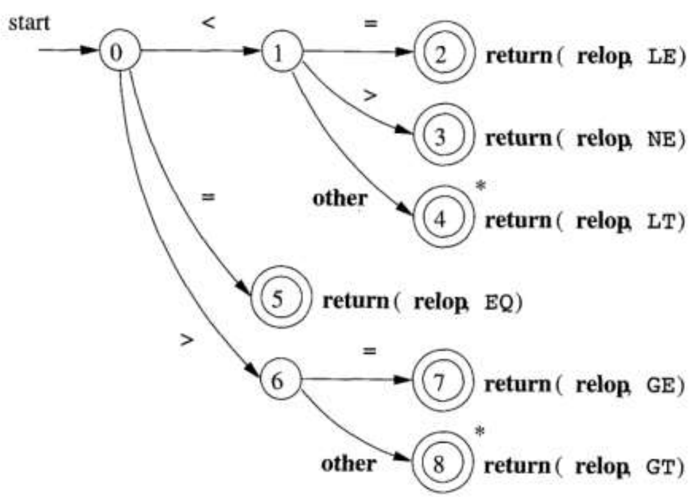


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

The transition diagram in the left is for recognizing **relop** tokens

States

- Represent conditions that could occur during the process of scanning the input for recognizing lexemes of tokens (what characters we have seen)
- The start state (开始状态), or initial state, is indicated by an edge labeled "start", which enters from nowhere
- Certain states are said to be accepting (接受状态), or final, indicating that a lexeme has been found

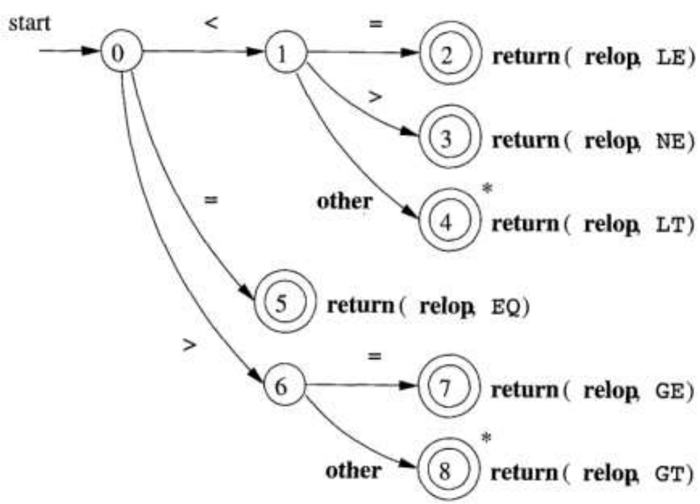


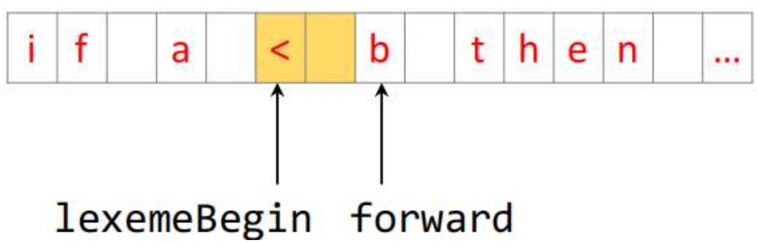
States 2, 3, 4, 5, 7, 8 are accepting.

By convention, we indicate accepting states by double circles

States Cont.

- At certain accepting states, the found lexeme may not contain all characters that we have seen from the start state
- Such states are annotated with *
- When entering * states, it is necessary to retract (回退) the forward pointer, which points to the next char in the input string)

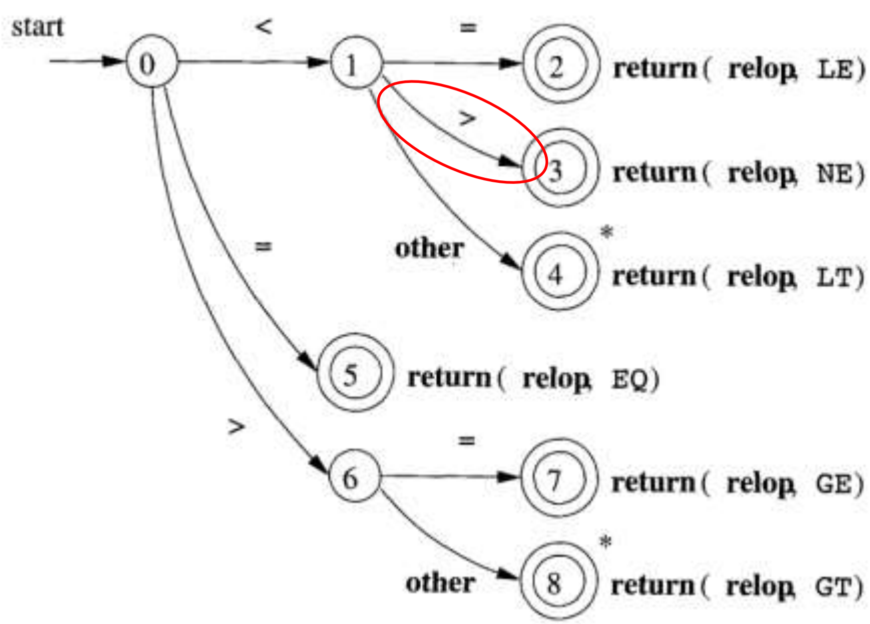


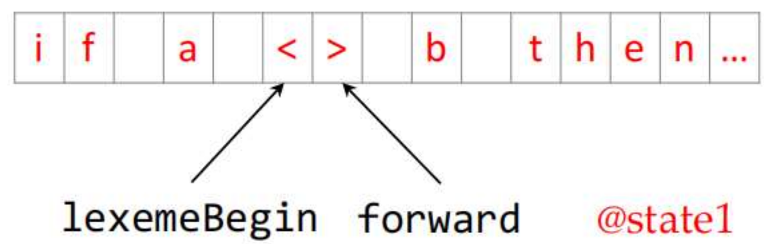


We should retract forward one position in the above case

Edges

- Edges are directed from one state to another
- Each edge is labeled by a symbol or set of symbols

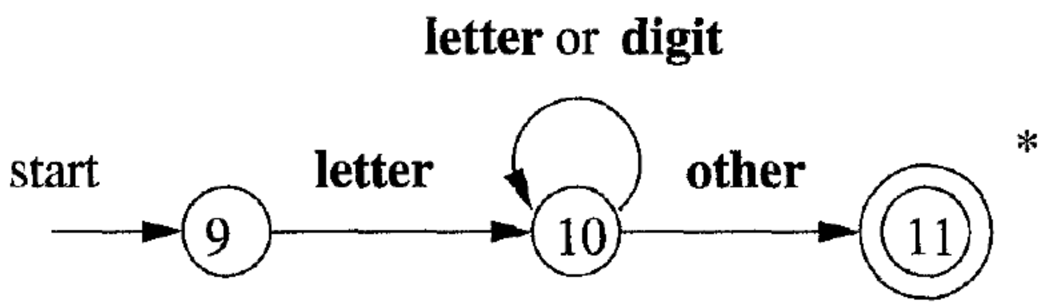




In the above case, we should follow the circled edge to enter state 3 and advance the forward pointer

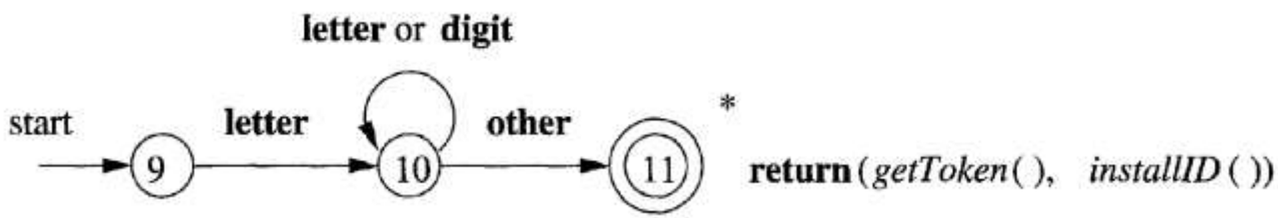
Recognition of Reserved Words and Identifiers (保留字和标识符的识别)

- In many languages, **reserved** words or **keywords** (e.g., then) also match the pattern of identifiers
- Problem: the transition diagram that searches for identifiers can also recognize reserved words

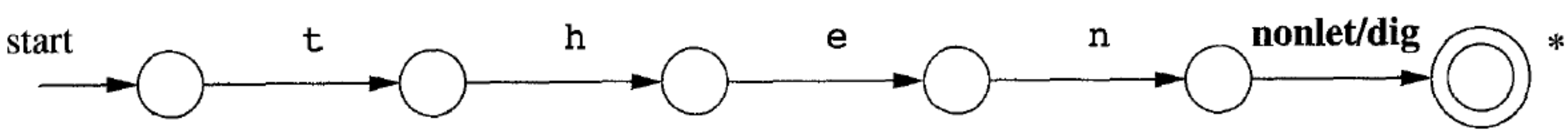


Handling Reserved Words

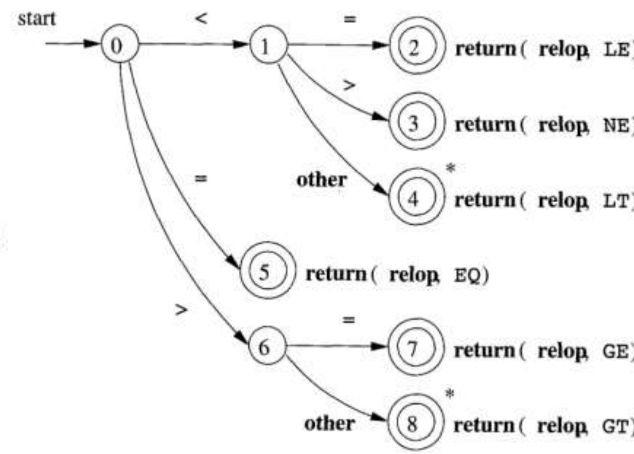
- Strategy 1: Preinstall the reserved words in the symbol table. Put a field in the symbol-table entries to indicate that these strings are not ordinary identifiers



- Strategy 2: Create a separate transition diagram with a high priority for each keyword

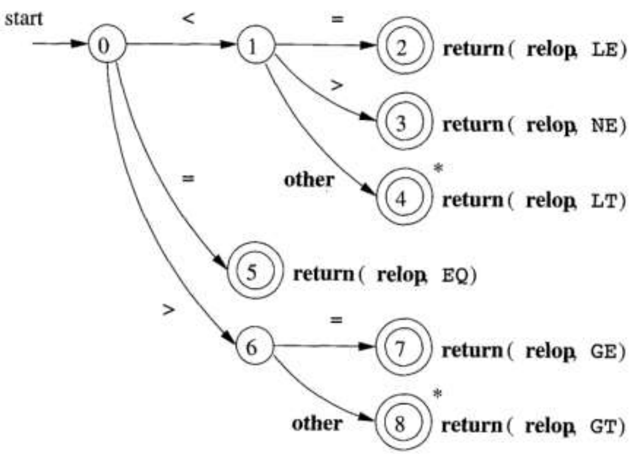


Building a Lexical Analyzer from Transition Diagrams




```
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);
        }
    }
}
```

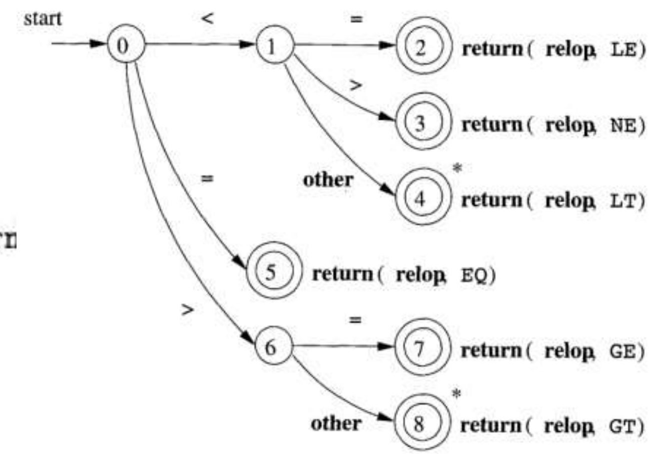
Building a Lexical Analyzer from Transition Diagrams




```
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);
        }
    }
}
```

Use a variable state to record the current state

Building a Lexical Analyzer from Transition Diagrams

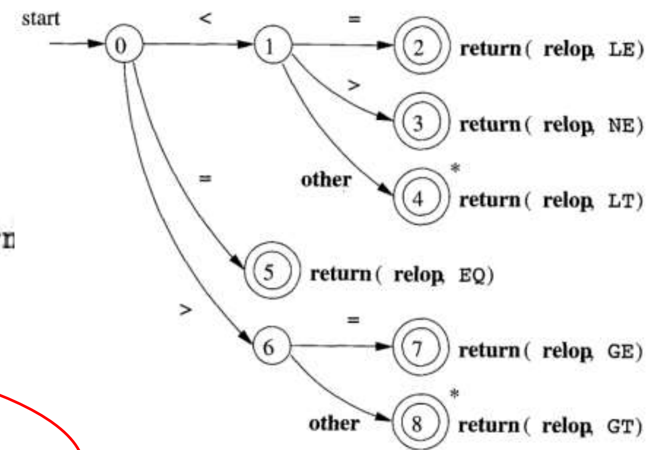



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) 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);
        }
    }
}
```

Sketch of implementation of relop transition diagram

A switch statement based on the value of state takes us to code for each of the possible states

Building a Lexical Analyzer from Transition Diagrams



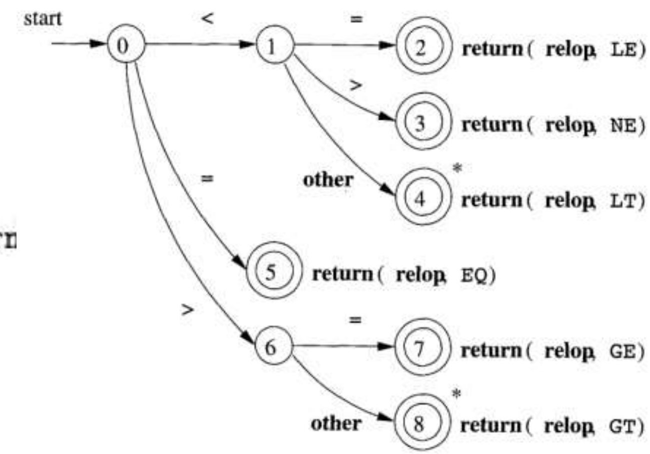

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) 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);
        }
    }
}
```

Sketch of implementation of relop transition diagram

The code of a normal state:

1. Read the next character
2. Determine the next state
3. If step 2 fails, do error recovery

Building a Lexical Analyzer from Transition Diagrams




```
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);
        }
    }
}
```

The code of an accepting state:

1. Perform retraction if the state has *
2. Set token attribute values
3. Return the token to parser

Sketch of implementation of relop transition diagram

Building the Entire Lexical Analyzer

- **Strategy 1**: Try the transition diagram for each token sequentially
  - fail() resets the pointer forward and starts the next diagram
- **Strategy 2**: Run transition diagrams in parallel
  - Need to resolve the case where one diagram finds a lexeme and others are still able to process input.
  - **Solution:** take the longest prefix of the input that matches any pattern
- **Strategy 3**: Combining all transition diagrams into one (preferred)
  - Allow the transition diagram to read input until there is no possible next state
  - Take the longest lexeme that matched any pattern

Outline

- The Role of Lexical Analyzer
- Specification of Tokens (Regular Expressions)
- Recognition of Tokens (Transition Diagrams)
- The Lexical-Analyzer Generator
- Finite Automata

The Lexical-Analyzer Generator Lex

- Lex, or a more recent tool Flex, allows one to specify a lexical analyzer by specifying regexps to describe patterns for tokens
- Often used with Yacc/Bison to create the frontend of compiler

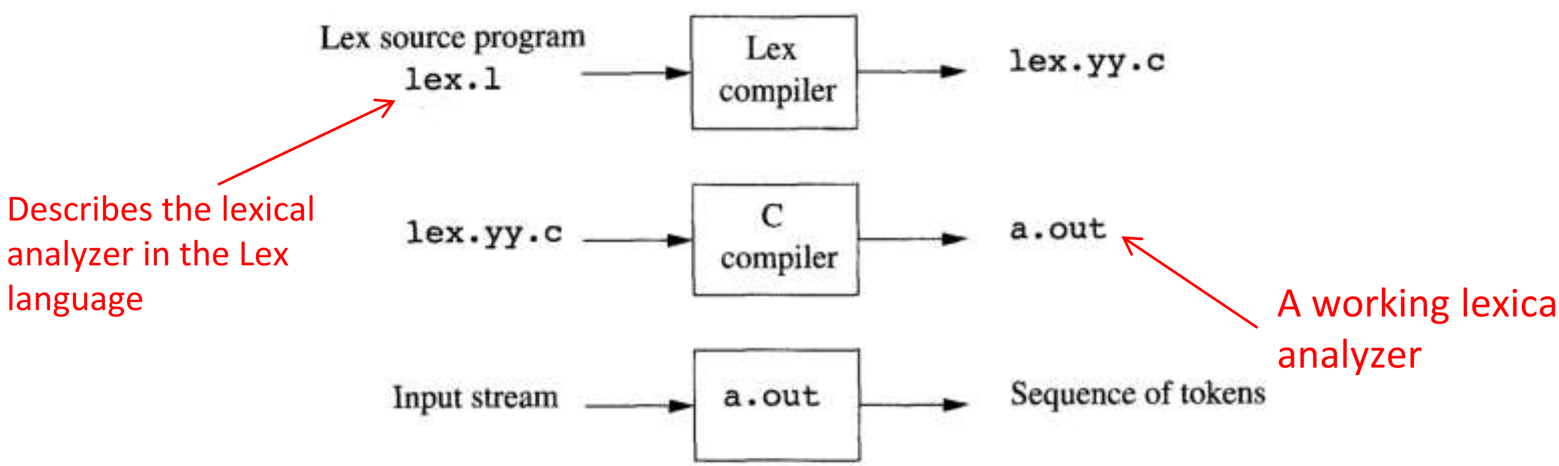


Figure 3.22: Creating a lexical analyzer with Lex

Structure of Lex Programs

- **A Lex program has three sections separated by %%**
  - Declaration (声明)
    - Variables, manifest constants (e.g., token names)
    - Regular definitions
  - Translation rules (转换规则) in the form "Pattern {Action}"
    - Each pattern (模式) is a regexp (may use the regular definitions of the declaration section)
    - Actions (动作) are fragments of code, typically in C, which are executed when the pattern is matched
  - Auxiliary functions section (辅助函数)
    - Additional functions that can be used in the actions

Lex Program Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Anything in between `%{` and `}%` is copied directly to `lex.yy.c`.

In the example, there is only a comment, not real C code to define manifest constants

Regular definitions that can be used in translation rules

Section separator

Lex Program Example Cont.

Continue to recognize other tokens

Return token name to the parser

Place the lexeme found in the symbol table

```
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); 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);}

%%
```

A global variable that stores a pointer to the symbol table entry for the lexeme. Can be used by the parser or a later component of the compiler.

Lex Program Example Cont.

- Everything in the auxiliary function section is copied directly to the file **`lex.yy.c`**
- Auxiliary functions may be used in actions in the translation rules

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Variables defined and set automatically by the lexical analyzer Lex generates

Conflict Resolution

- When the generated lexical analyzer runs, it analyzes the input looking for prefixes that match <u>any</u> of its patterns.*
- **Rule 1**: If it finds multiple such prefixes, it takes the longest one
  - The analyzer will treat <= as a single lexeme, rather than < as one lexeme and = as the next
- **Rule 2**: If it finds a prefix matching different patterns, the pattern listed first in the Lex program is chosen.
  - If the keyword patterns are listed before identifier pattern, the lexical analyzer will not recognize keywords as identifiers

*See Flex manual for details (Chapter 8: How the input is matched) at http://dinosaur.compilertools.net/flex/

Outline

- The Role of Lexical Analyzer
- Specification of Tokens (Regular Expressions)
- Recognition of Tokens (Transition Diagrams)
- The Lexical-Analyzer Generator
- Finite Automata →
  - NFA & DFA
  - NFA → DFA
  - Regexp → NFA
  - Combining NFA's (to be discussed in lab)
  - DFA Minimization (to be discussed in lab)

Finite Automata (有穷自动机)

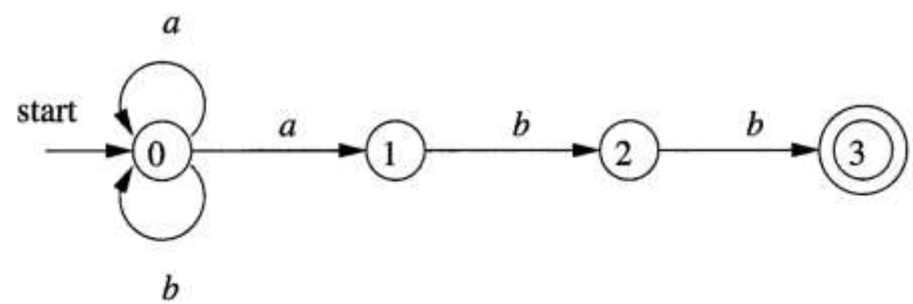
- Finite automata are the simplest machines to recognize patterns
- They are essentially graphs like transition diagrams. They simply say "yes" or "no" about each possible input string.
  - Nondeterministic finite automata (NFA, 非确定有穷自动机): A symbol can label several edges out of the same state (allowing multiple target states), and the empty string ϵ is a possible label.
  - Deterministic finite automata (DFA, 确定有穷自动机): For each state and for each symbol of its input alphabet, there is exactly one edge with that symbol leaving that state.
- NFA and DFA recognize the same languages, regular languages, that regexps can describe.

Nondeterministic Finite Automata

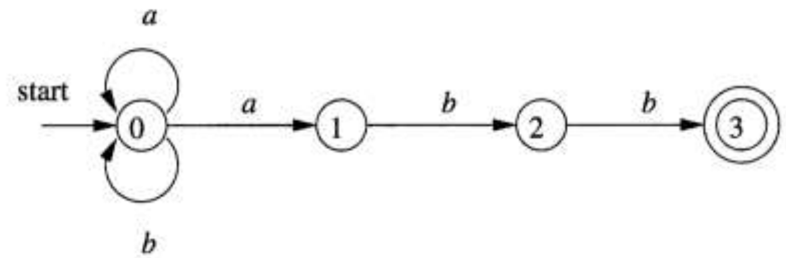
- A nondeterministic finite automaton (NFA) consists of:
  - A finite set of states S
  - A set of input symbols Σ, the input alphabet. We assume that the empty string ϵ is never a member of Σ.
  - A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.
  - A start state (or initial state) s_0 from S
  - A set of accepting states (or final states) F, a subset of S

NFA Example

- S = {0, 1, 2, 3}
- Start state: 0
- Accepting states: {3}
- Transition function
  - (0, a) → {0, 1} (0, b) → {0}
  - (1, b) → {2} (2, b) → {3}



Transition Table

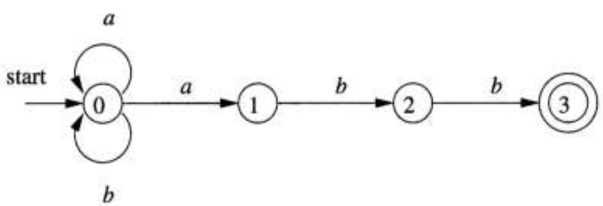


- Another representation of an NFA
  - Rows correspond to states
  - Columns correspond to the input symbols or ϵ
  - The table entry for a state-input pair lists the set of next states
  - $\emptyset$: the transition function has no info about the state-input pair

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Acceptance of Input Strings

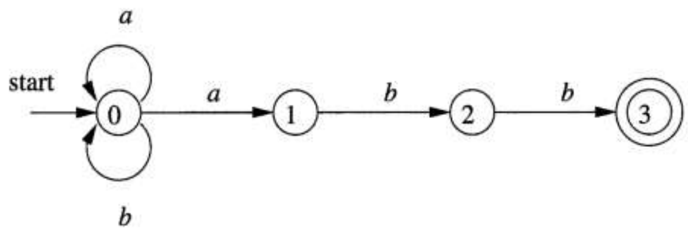
- An NFA accepts an input string x if and only if
  - There is a path in the transition graph from the start state to one accepting state, such that the symbols along the path form x (ϵ labels are ignored).



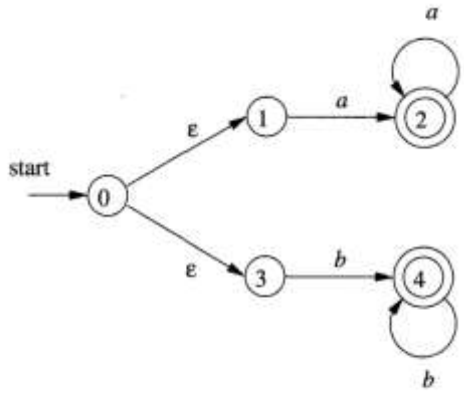
accepts the string aabb

- The language defined or accepted by an NFA
  - The set of strings labelling some path from the start state to an accepting state

NFA and Regular Languages



L((a|b)*abb)



L(aa*|bb*)

Deterministic Finite Automata (DFA)

- A deterministic finite automaton is a special NFA where:
  - There are no moves on input ϵ
  - For each state s and input symbol a, there is exactly one edge out of s labeled a
- DFA can efficiently accept/reject strings (recognize patterns)
- Every regexp and every NFA can be converted to a DFA accepting the same language

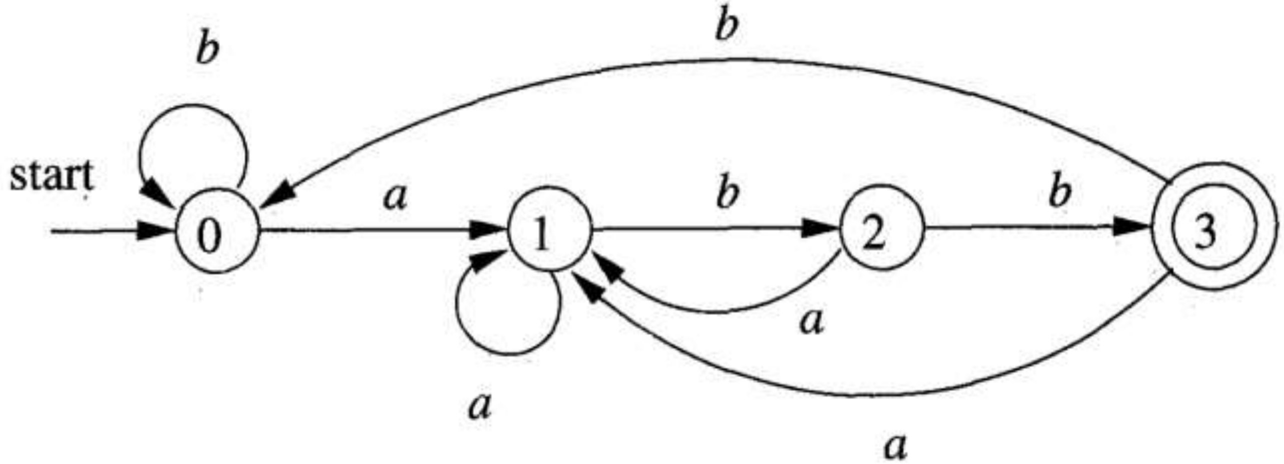
Simulating a DFA

- **Input:**
  - String x terminated by an end-of-file character eof.
  - DFA D with start state s_0, accepting states F, and transition function move
- **Output:** Answer "yes" if D accepts x; "no" otherwise

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

DFA Example

- Given the input string ababb, the DFA below enters the sequence of states 0, 1, 2, 1, 2, 3 and returns ”yes”



What’s the language defined by this DFA?

From Regular Expressions to Automata

- Regexps concisely & precisely describe the patterns of tokens
- DFA can efficiently recognize patterns (comparatively, the simulation of NFA is less straightforward*)
- When implementing lexical analyzers, regexps are often converted to DFA:
  - Regexp → NFA → DFA

* There may be multiple transitions at a state when seeing a symbol

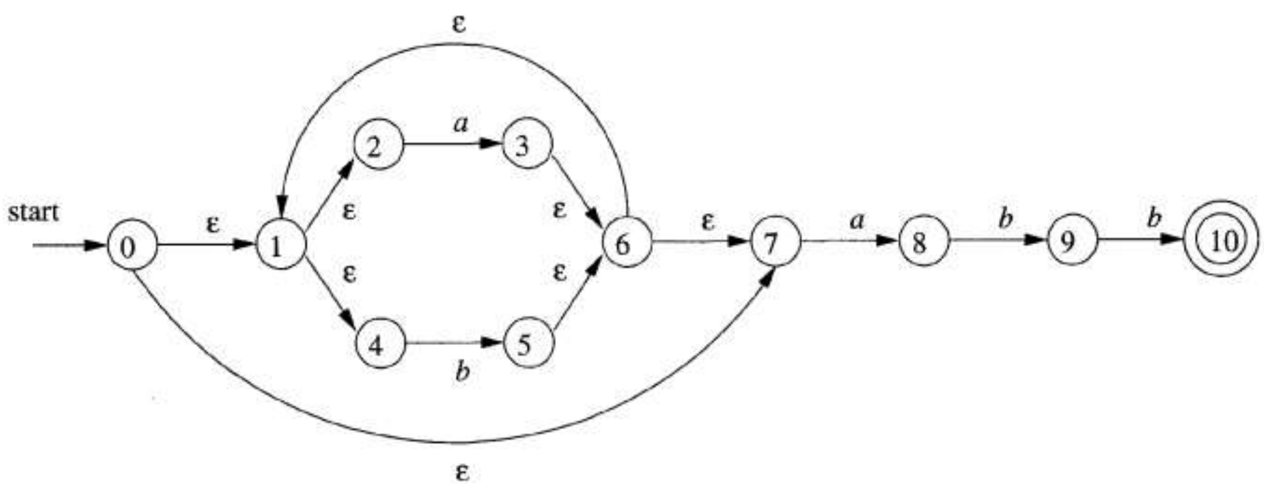
Conversion of an NFA to a DFA

- The subset construction algorithm (子集构造法)
  - Insight: Each state of the constructed DFA corresponds to a set of NFA states
    - After reading the input $a_1a_2\ldots a_n$, the DFA is in the state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1a_2\ldots a_n$
  - The algorithm simulates "in parallel" all possible moves an NFA can make on a given input string

Example for Algorithm Illustration

- The NFA below accepts the string babb
  - There exists a path from the start state 0 to the accepting state

10, on which the labels on the edges form the string babb



Subset Construction Technique

- It is possible that # DFA states is exponential in # NFA states (worst case)
  - Each DFA state corresponds to a subset of NFA states
- However, for real languages, the NFA and DFA have approximately the same number of states, and the exponential behavior is not seen

Subset Construction Technique Cont.

- Operations used in the algorithm:
  - ***ϵ*-closure(s)**: Set of NFA states reachable from NFA state s on ϵ-transitions alone
  - ***ϵ*-closure(T)**: Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone
    - That is, $\bigcup_{s \text{ in } T}$ ϵ-closure(s)
  - **move(T, a)**: Set of NFA states to which there is a transition on input symbol a from some state s in T

Subset Construction Technique Cont.

- **Computing ϵ-closure(T)**
  - It is a graph traversal process (only consider ϵ edges)
  - Computing ϵ-closure(s) is essentially the same

```
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;
                }
}
```

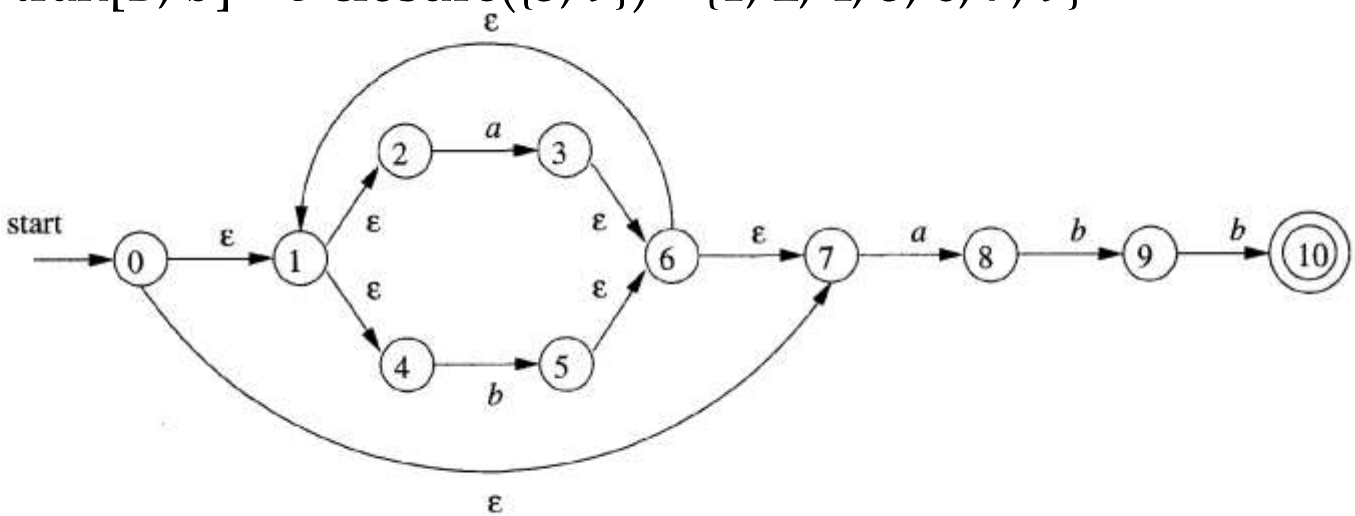
Subset Construction Technique Cont.

- The construction of the set of D's states, Dstates, and the transition function Dtran is also a search process
  - Initially, the only state in Dstates is ϵ-closure(s_0) and it is unmarked
    - Unmarked state means that its next states have not been explored

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

Example

- A: ϵ-closure(0) = {0, 1, 2, 4, 7}
- B: Dtran[A, a] = ϵ-closure({3, 8}) = {1, 2, 3, 4, 6, 7, 8}
- C: Dtran[A, b] = ϵ-closure({5}) = {1, 2, 4, 5, 6, 7}
- D: Dtran[B, b] = ϵ-closure({5, 9}) = {1, 2, 4, 5, 6, 7, 9}
- …



Transition Table of the DFA

- Start state: A; Accepting states: {E}

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

This DFA can be further minimized: A and C have the same moves on all symbols and can be merged.

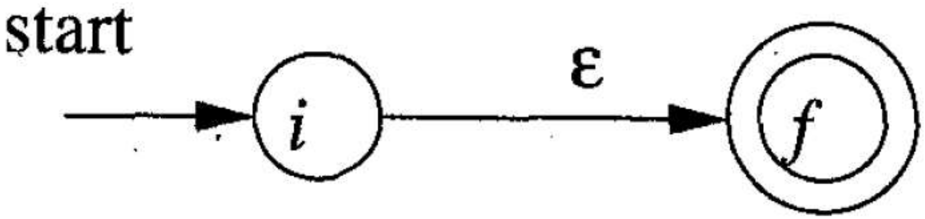
Regular Expression to NFA

- **Thompson's construction algorithm (Thompson构造法)**
- The algorithm works recursively by splitting an expression into its constituent subexpressions, from which the NFA will be constructed using a set of rules
- The rules for constructing an NFA:
  - **Two basis rules (基本规则)**: handle subexpressions with no operators
  - **Three inductive rules (归纳规则)**: construct larger NFA's from the NFA's for subexpressions

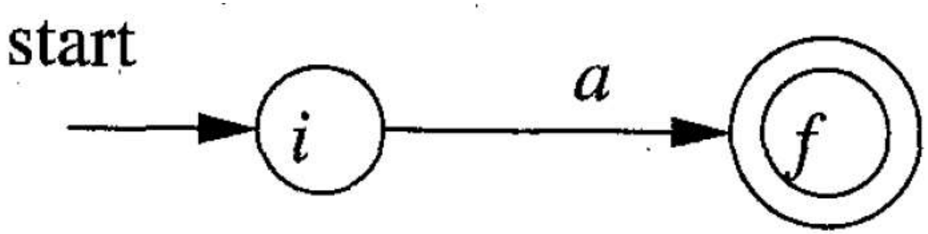
Thompson's Construction Algorithm

Two basis rules:

1. The empty expression ϵ is converted to



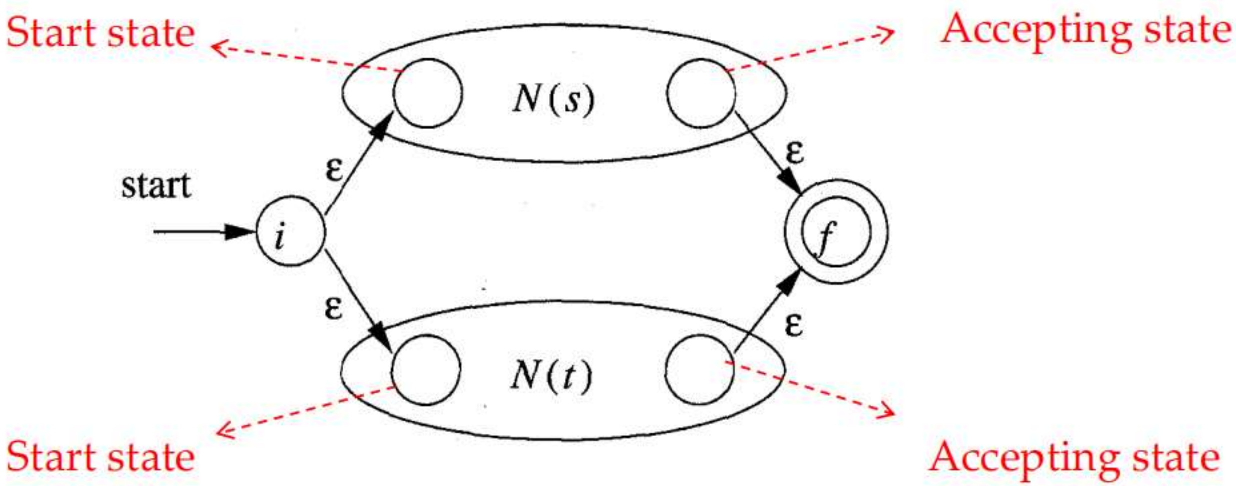
2. Any subexpression a (a single symbol in input alphabet) is converted to



Thompson's Construction Algorithm

The inductive rules: the union case

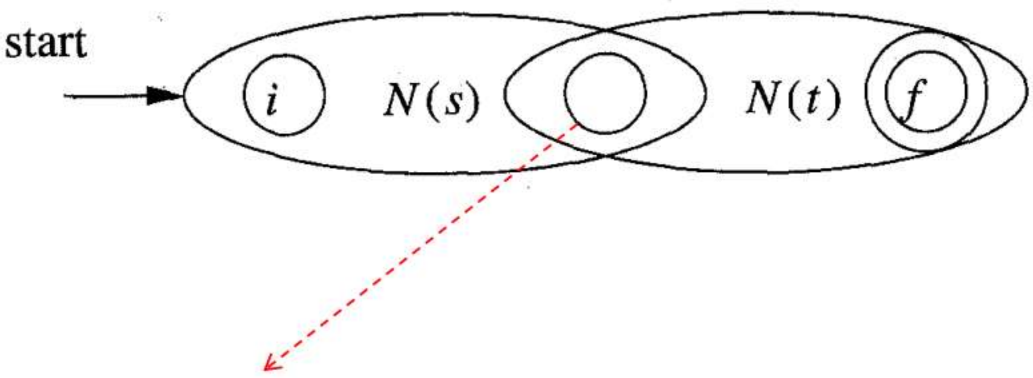
- s | t : N(s) and N(t) are NFA's for subexpressions s and t



Thompson's Construction Algorithm

The inductive rules: the concatenation case

- st :N(s) and N(t) are NFA's for subexpressions s and t

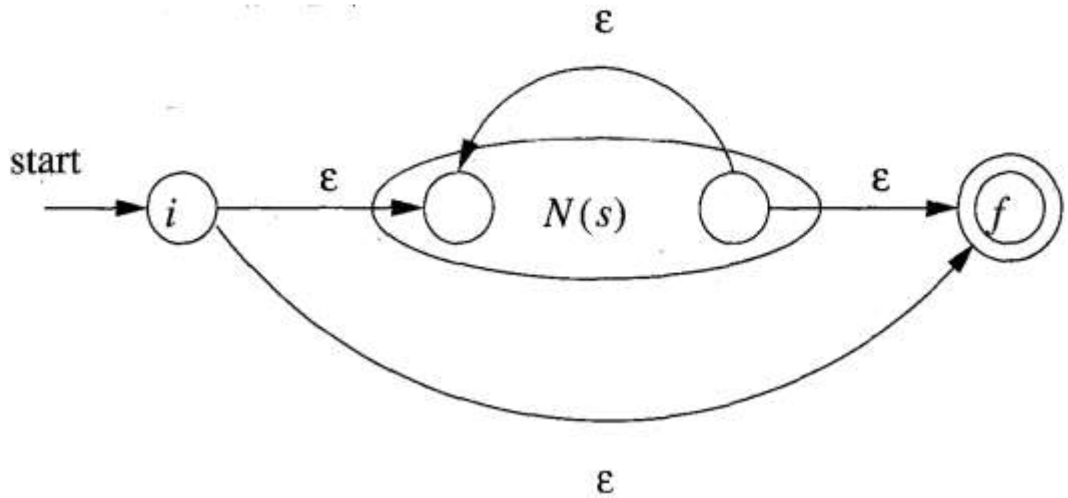


Merging the accepting state of $N(s)$ and the start state of $N(t)$

Thompson's Construction Algorithm

The inductive rules: the Kleene star case

- s^*: N(s) is the NFA for subexpression s

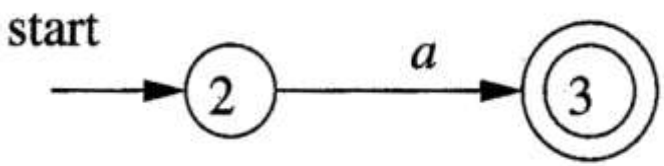


Example

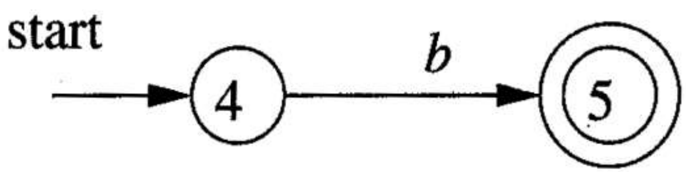
- Use Thompson's algorithm to construct an NFA for the regexp

  r = **$(a \mid b)^*abb$**

NFA for the first a (apply basis rule #1)

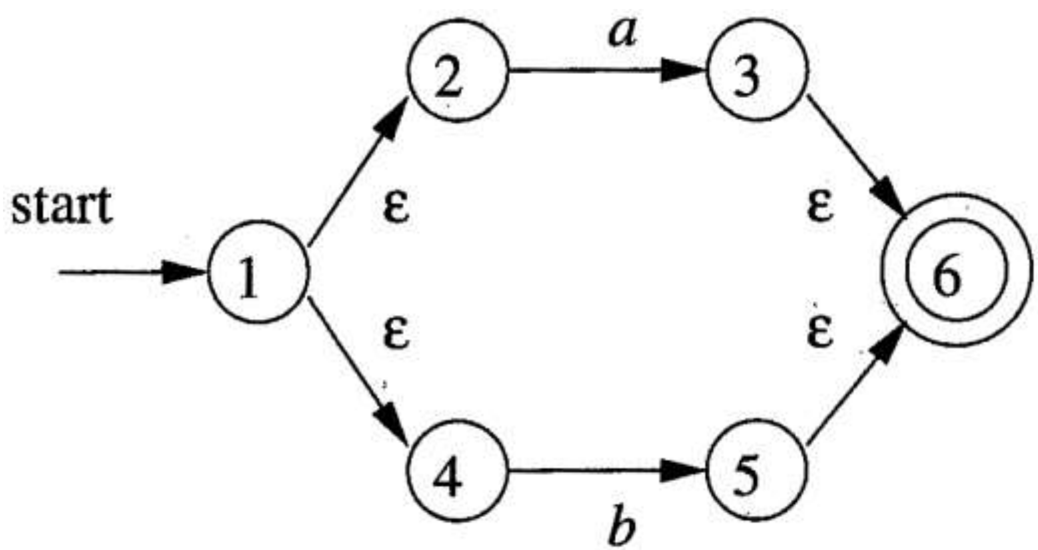


NFA for the first b (apply basis rule #1)



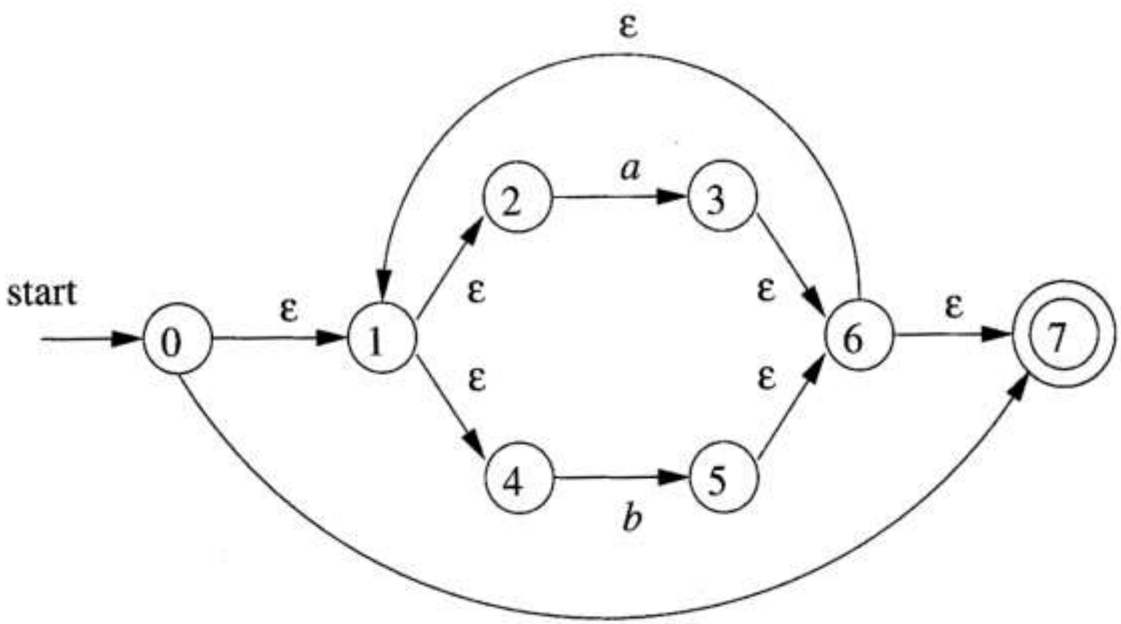
Example Cont.

- NFA for **(a|b)** (apply inductive rule #1)



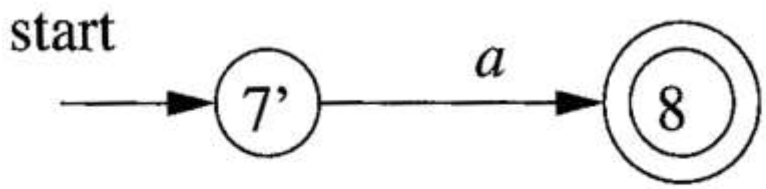
Example Cont.

- NFA for **(a|b)*** (apply inductive rule #3)



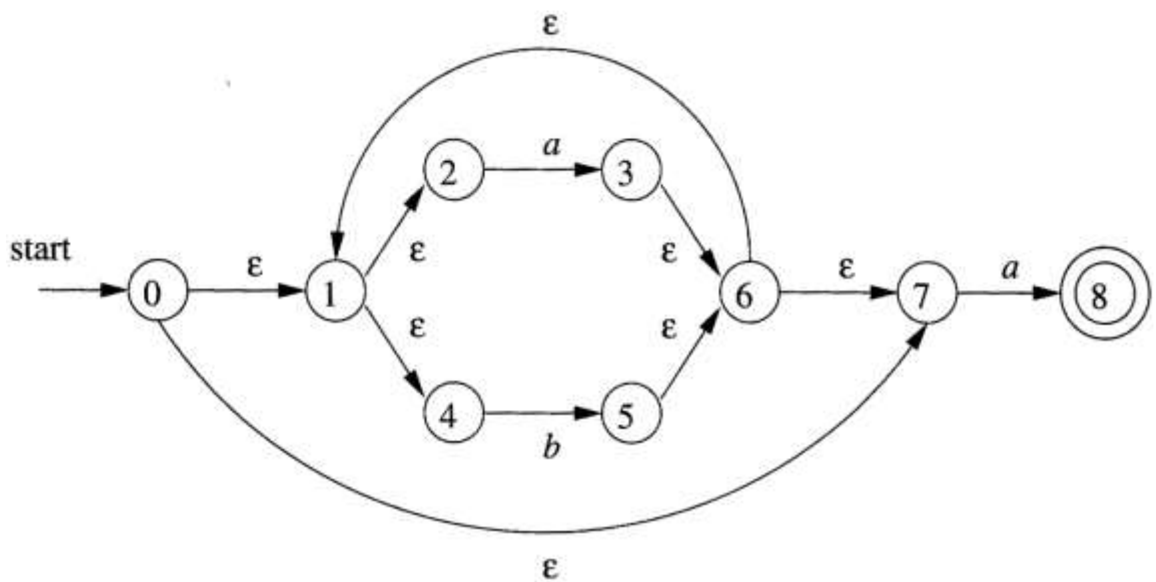
Example Cont.

- NFA for the second **a** (apply basic rule #1)



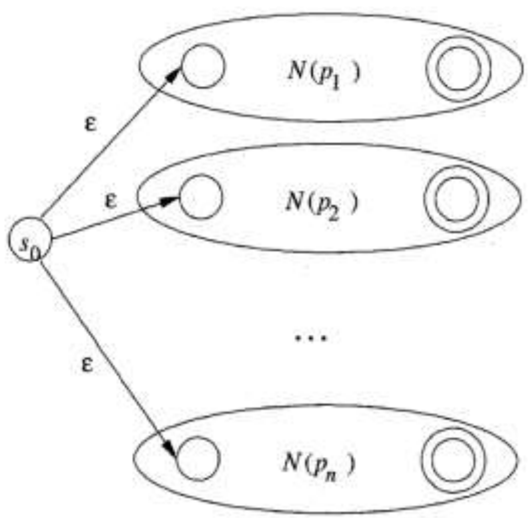
Example Cont.

- NFA for **(a|b)*a** (apply inductive rule #2) …



Combining NFA’s

- **Why?** In the lexical analyzer, we need a single automaton to recognize lexemes matching any pattern (in the lex program)
- **How?** Introduce a new start state with ϵ-transitions to each of the start states of the NFA’s N_i for pattern p_i



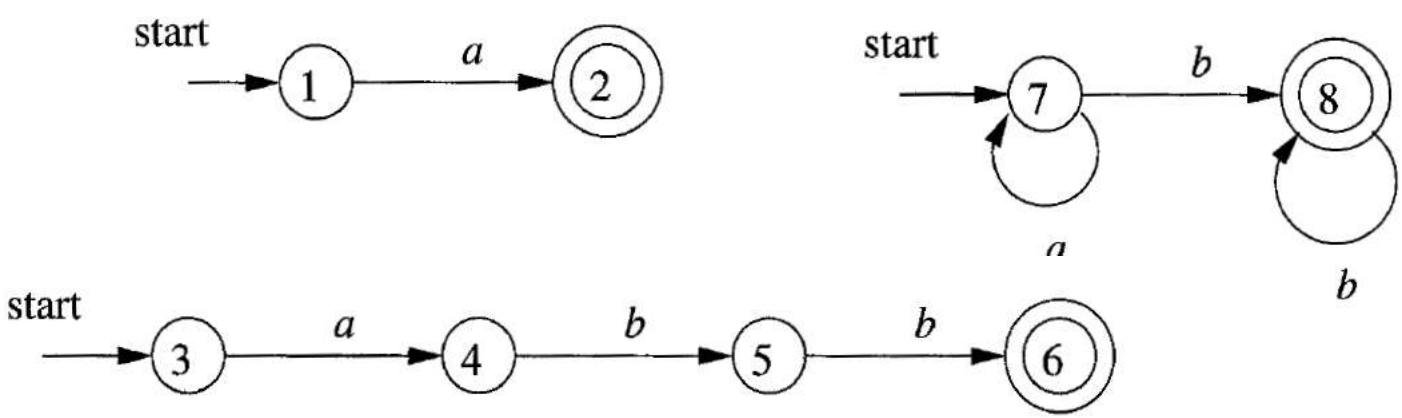
- The language that can be accepted by the big NFA is the union of the languages that can be accepted by the small NFA’s
- Different accepting states correspond to different patterns

DFA's for Lexical Analyzers

- Convert the NFA for all the patterns into an equivalent DFA, using the subset construction algorithm
- An accepting state of the DFA corresponds to a subset of the NFA states, in which at least one is an accepting NFA state
  - If there are more than one accepting NFA state, this means that conflicts arise (the prefix of the input string matches multiple patterns)
  - Upon conflicts, find the first pattern whose accepting state is in the set and make that pattern the output of the DFA state

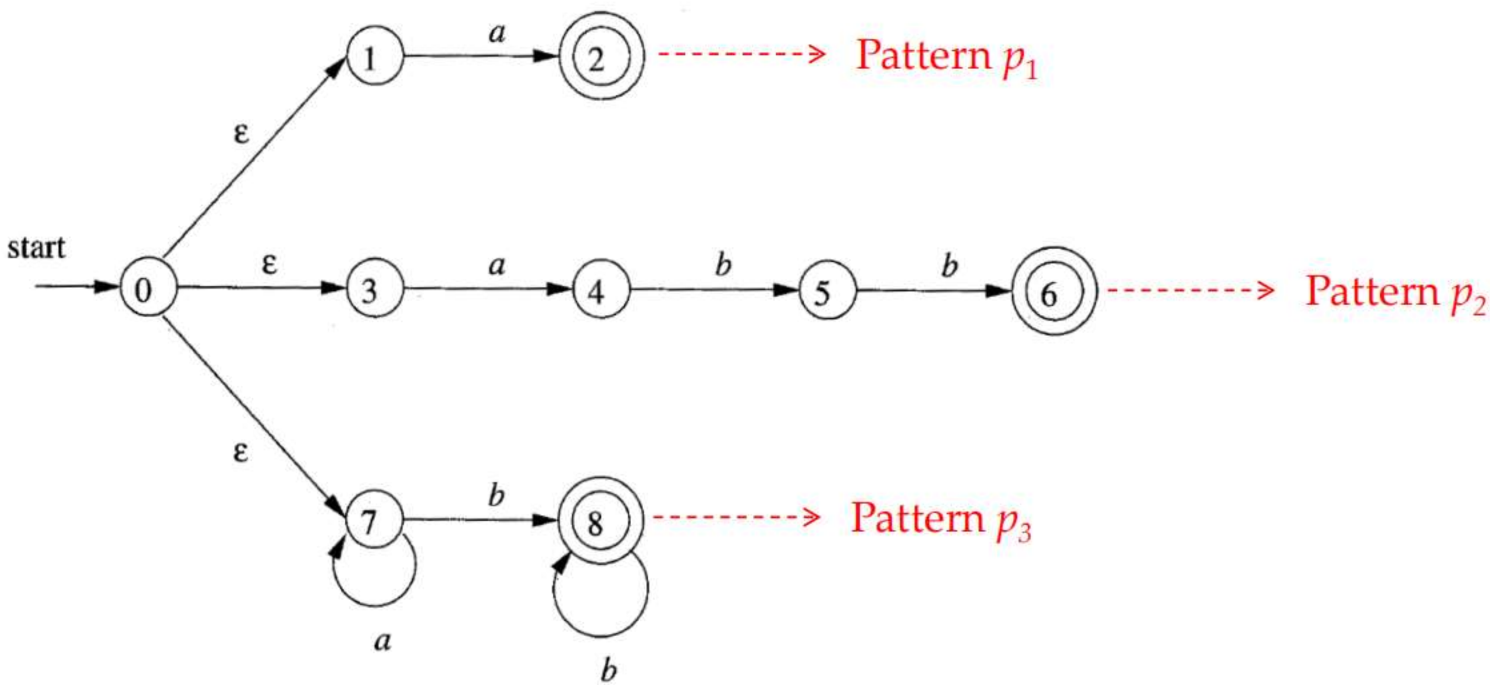
Example

- Suppose we have three patterns: p_1, p_2, and p_3
  - **a** {action A_1 for pattern p_1}
  - **abb** {action A_2 for pattern p_2}
  - $\mathbf{a^*b^+}$ {action A_3 for pattern p_3}
- We first construct an NFA for each pattern



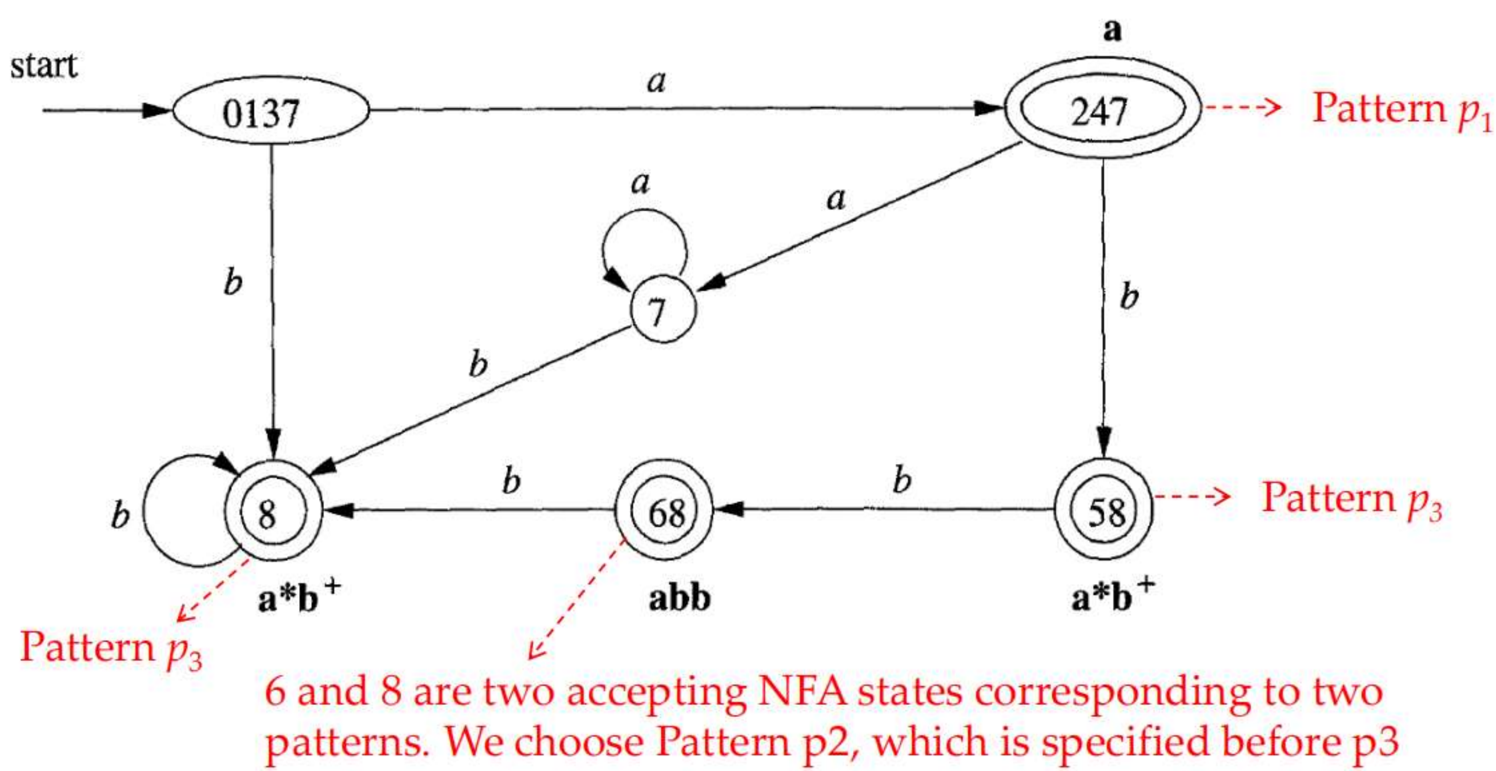
Example Cont.

- Combining the three NFA's



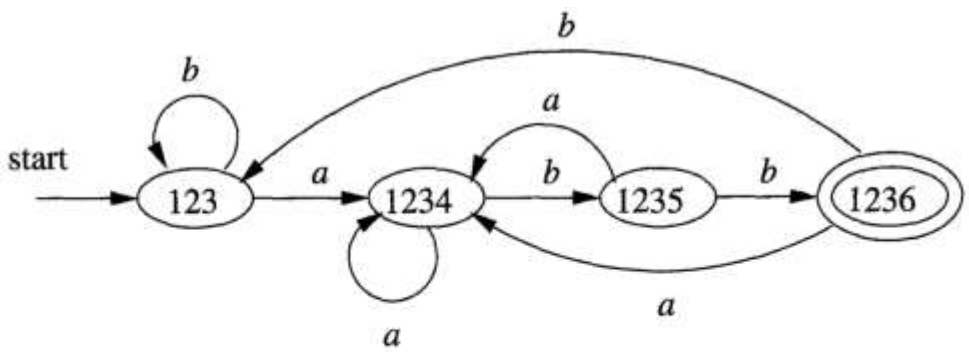
Example Cont.

- Converting the big NFA to a DFA

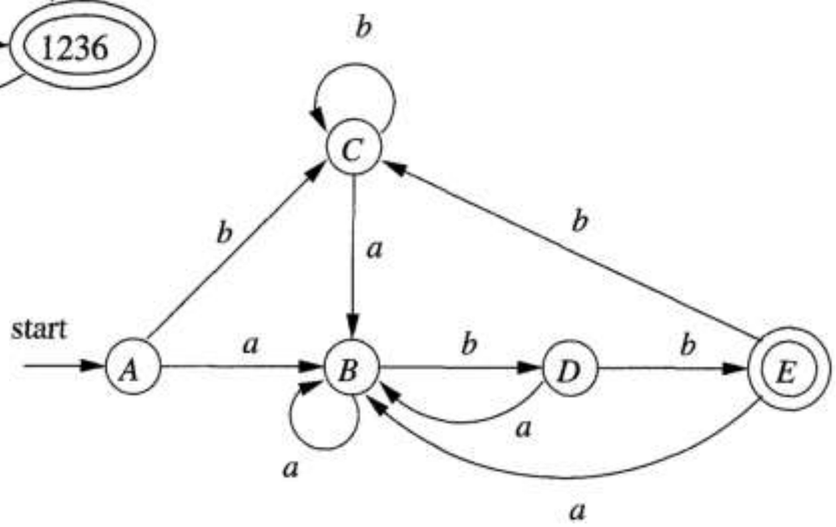


Minimizing # States of a DFA

- There can be many DFA's recognizing the same language

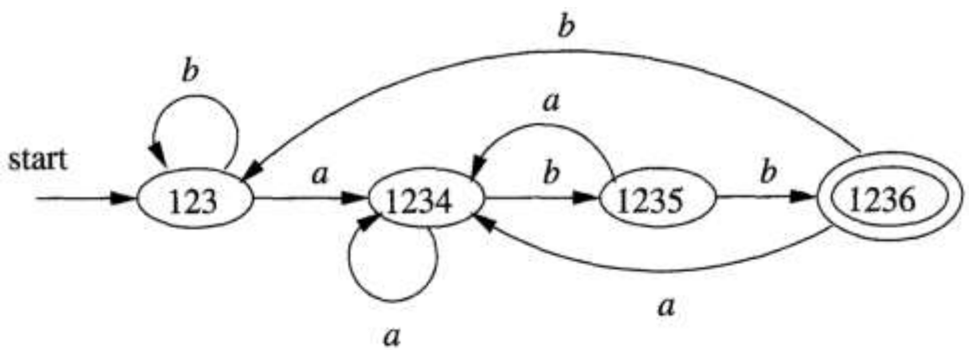


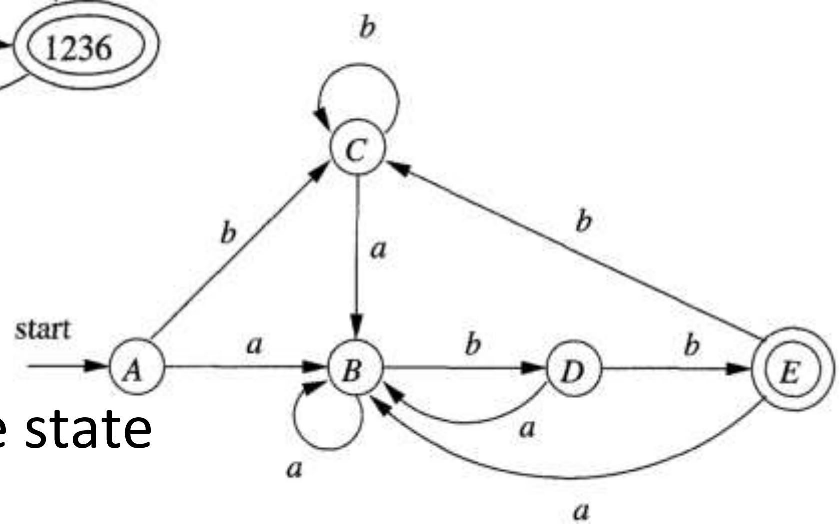
Two equivalent DFA's, both recognizing regular language L((a|b)*abb)



Minimizing # States of a DFA

- There can be many DFA's recognizing the same language





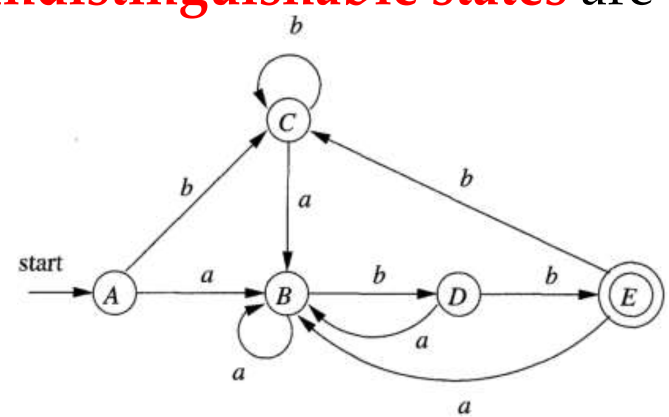
States A and C are equivalent:
Neither is accepting, and on any
symbol they transfer to the same state
A and C behave like 123

Minimizing # States of a DFA Cont.

- There is always a unique minimum-state DFA for any regular language (state name does not matter)
- The minimum-state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states

Distinguishing States

- **Distinguishable states**
  - We say that string x distinguishes state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state
  - States s and t are distinguishable if there exists some string that distinguishes them
- **Indistinguishable states** are equivalent and can be merged



- The empty string ϵ distinguishes any accepting state from any nonaccepting state

- The string bb distinguishes state A from B, since bb takes A to a nonaccepting state C, but takes B to accepting state E

DFA State-Minimization Algorithm

Works by partitioning the states of a DFA into groups of states that cannot be distinguished (an iterative process)

- The algorithm maintains a partition (划分), whose groups are sets of states that have not yet been distinguished
- Any two states from different groups are known to be distinguishable
- When the partition cannot be refined further by breaking any group into smaller groups, we have the minimum-state DFA

The Partitioning Part

1.Start with an initial partition $\prod$ with two groups, F and S-F, the accepting and nonaccepting states of D

2. Apply the procedure below to construct a new partition $\prod_{new}$

```
initially, let Πnew = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Πnew by the set of all subgroups formed;
}
```

3. If $\prod_{new} == \prod$, let $\prod_{final} = \prod$ and the partitioning finishes; Otherwise, $\prod = \prod_{new}$ and repeat step 2

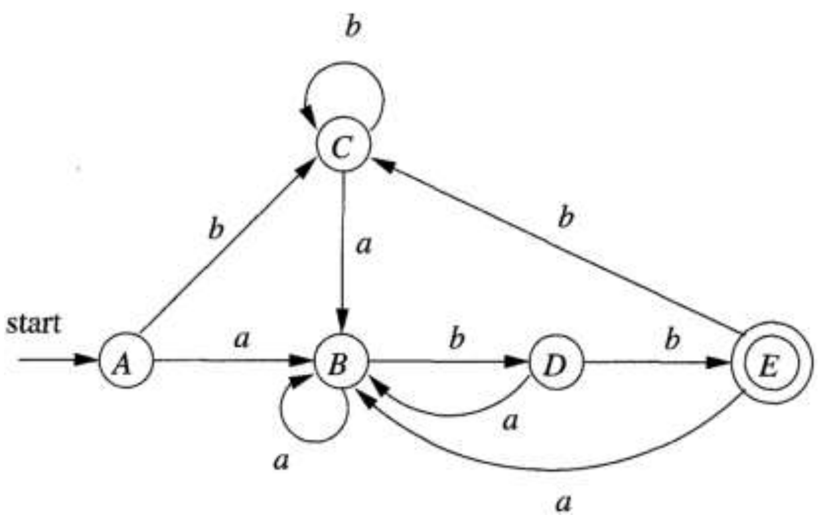
The Construction Part

Choose one state in each group of $\prod_{final}$ as the representative for that group. The representatives will be the states of the minimum state DFA D′

- The start state of D′ is the representative of the group containing the start state of D
- The accepting states of D' are the representatives of those groups that contain an accepting state of D
- Establish transitions:
  - Let s be the representative of group G in ∏final; Let the transition of D from s on input a be to state t; Let r be the representative of t′s group H
  - Then in D′, there is a transition from s to r on input a

Example

- Initial partition: {A, B, C, D}, {E}
- Handling group {A, B, C, D}: b splits it to two subgroups {A, B, C} and {D}
- Handling group {A, B, C}: b splits it to two subgroups {A, C} and {B}
- Picking A, B, D, E as representatives to construct the minimum-state DFA



STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A

State Minimization in Lexical Analyzers

The basic idea is the same as the state-minimization algorithm for DFA.

- Differences are:
  - Each accepting state in the lexical analyzer's DFA corresponds to a
    different pattern. These states are not equivalent.
  - So the initial partition should be: one group of non accepting states + groups of accepting states for different patterns

Example

Initial partition: {0137, 7}, {247}, {68}, {8, 58}, {∅}

- We add a dead state ∅: we suppose has transitions to itself on inputs a and b. It is also the target of missing transitions on a from states 8, 58, and 68.