Chapter 2: Lexical Analysis

Yepang Liu

liuyp1@sustech.edu.cn

The chapter numbering in lecture notes does not follow that in the textbook.

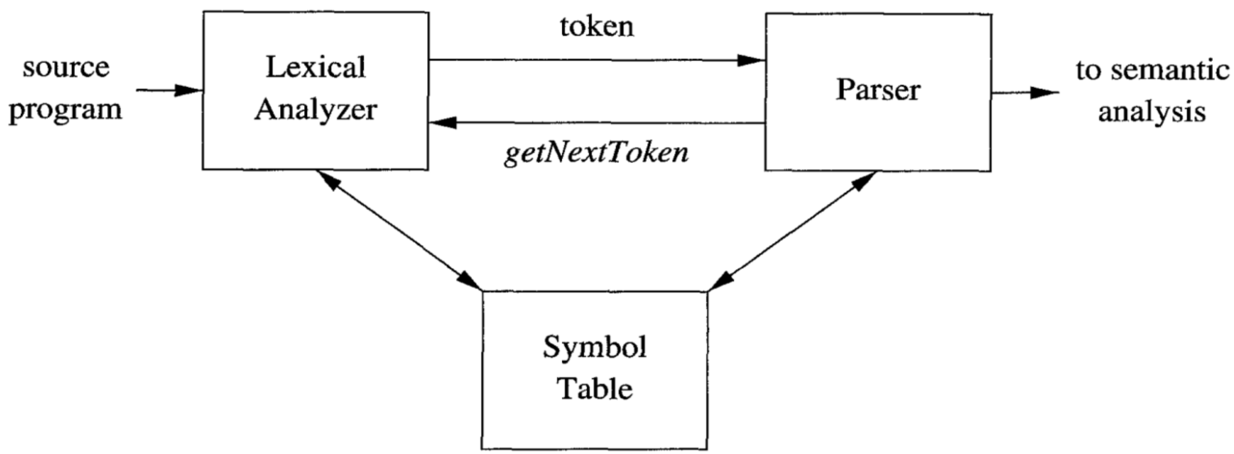
Outline

- 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



The Role of Lexical Analyzer

```
position = initial + rate * 60
```

Lexical Analyzer

```
<id, 1>
<=>
<id, 2>
<+>
<id, 3>
<*>
<60>
```

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...

Tokens, Patterns, and Lexemes

- A ***lexeme*** is a string of characters that is a lowest-level syntactic unit in programming languages

- A ***token*** is a syntactic category representing a class of lexemes. Formally, it is a pair <token name, attribute value>
  - Token name: an abstract symbol representing the kind of the token
  - Attribute value (optional) points to the symbol table

- Each token has a particular ***pattern***: a description of the form that the lexemes of the token may take

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.

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

Lexical Errors

- When none of the patterns for tokens match any prefix of the remaining input
- Example: `int 3a = a * 3;`

Outline

- The Role of the 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 symbols in *s* (i.e., cardinality)
  - Empty string (空串): the string of length 0, ϵ

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 *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)

Terms Cont.

- **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 (bnn)

How many substrings does banana have?

(Two substrings are different as long as they have different start/end index)

String 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

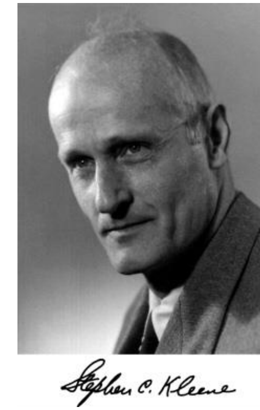
- **Exponentiation (幂/指数运算):** $s^0 = \epsilon$, $s^1 = s$, $s^i = s^{i-1}s$
  - x = dog, $x^0 = \epsilon$, x^1 = dog, x^3 = dogdogdog

Language (语言)

- A **language** 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.

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

The exponentiation of L can be defined using concatenation. L^n means concatenating L n times.

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

Examples

- $L = \{A, B, \ldots, Z, a, b, \ldots, z\}$
- $D = \{0, 1, \ldots, 9\}$

$L \cup D$	$\{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
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 strings
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: $(a) \mid ((b)^{*}(c)) = a \mid b^{*}c$

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 Language (正则语言)

- A **regular language** 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 $r = s$

Regular Language Cont.

- 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

Is (**a** | **b**)(**a** | **b**) = **aa** | **ab** | **ba** | **bb** true?

Regular Definitions (正则定义)

- 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:

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

where:

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

Each new symbol denotes a regular language. The second rule means that you may reuse previously-defined symbols.

Examples

- Regular definition for C identifiers

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

`_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
    - $[a_1 a_2 \ldots a_n] = a_1 \mid a_2 \mid \ldots \mid a_n$
    - $[a\text{-}e] = a \mid b \mid c \mid d \mid e$
- The extensions are only for notational convenience, they do not change the descriptive power of regexps

Outline

- The Role of the 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` $\rightarrow$ `(blank | tab | newline)`$^{+}$
  - When the lexical analyzer recognizes a whitespace token, it does not return it to the parser, but restart from the next character

Example: Patterns and Tokens

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

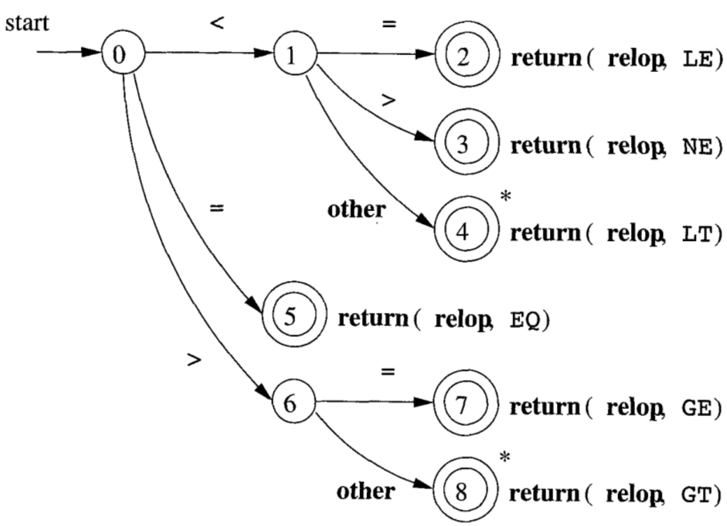
Patterns for tokens

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

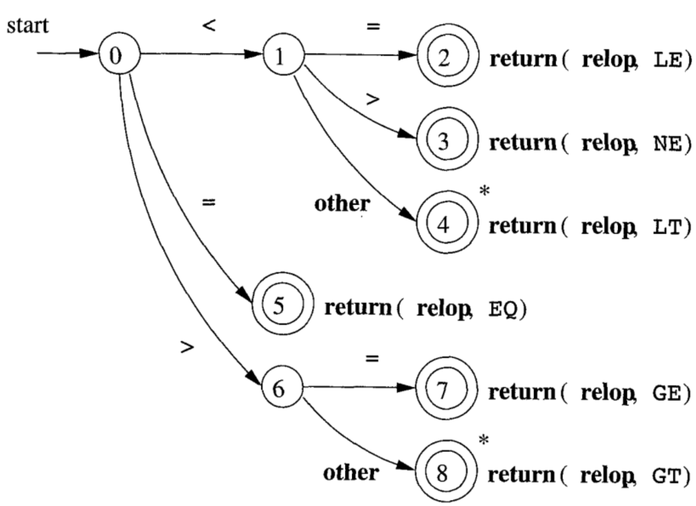


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

The transition diagram in the left recognizes **relop** tokens

States

- Represent conditions that could occur during the process of scanning (i.e., 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

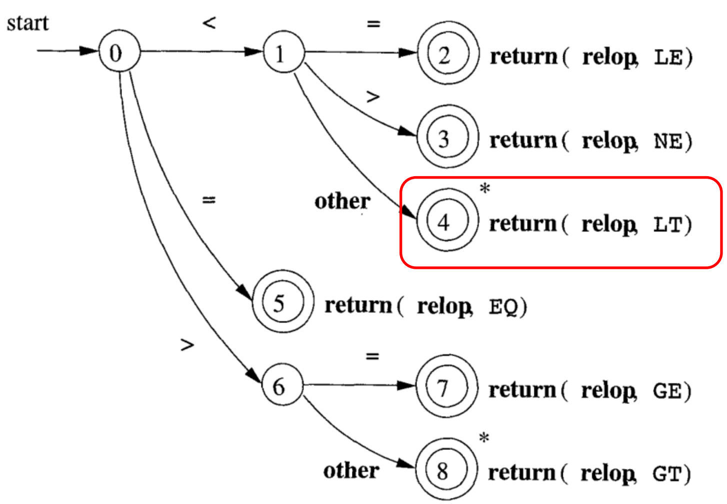


States 2-8 are accepting. They return a pair (token name, attribute value).

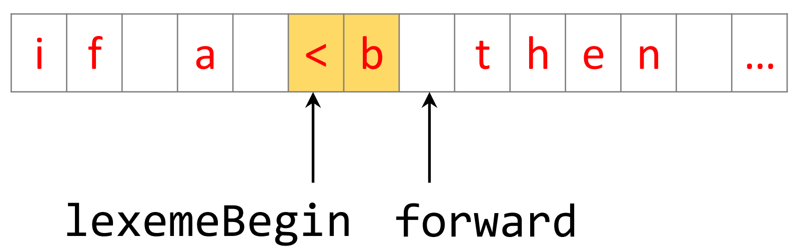
By convention, we indicate accepting states by double circles

The Retract Action

- 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



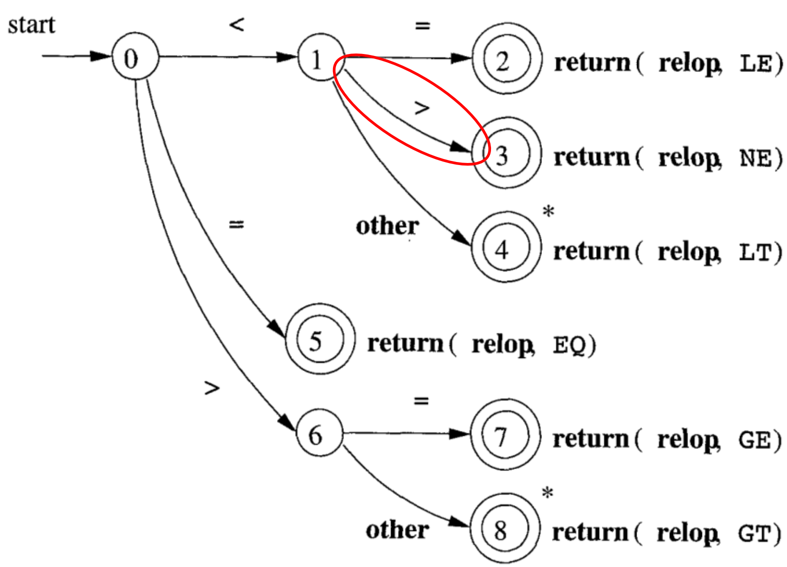
- The found lexeme: <
- The characters we've seen: <b

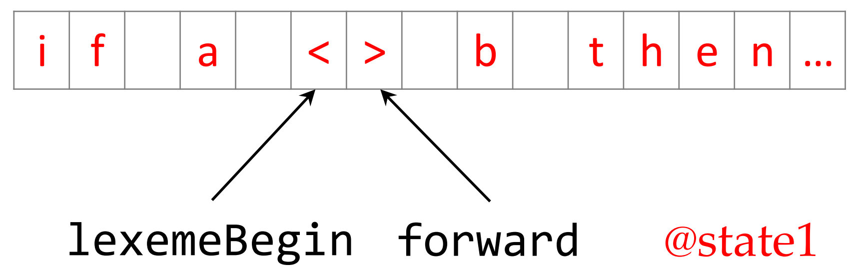


We should retract `forward` one step back

Edges

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

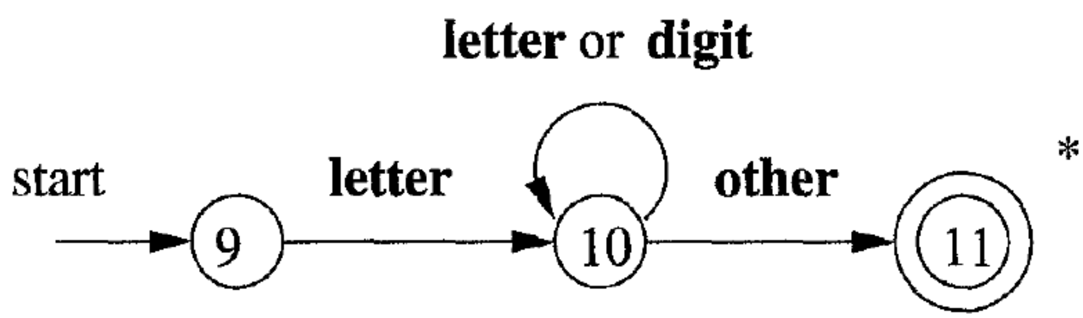




In the above case, we should follow the circled edge to enter state 3 and advance the `forward` pointer

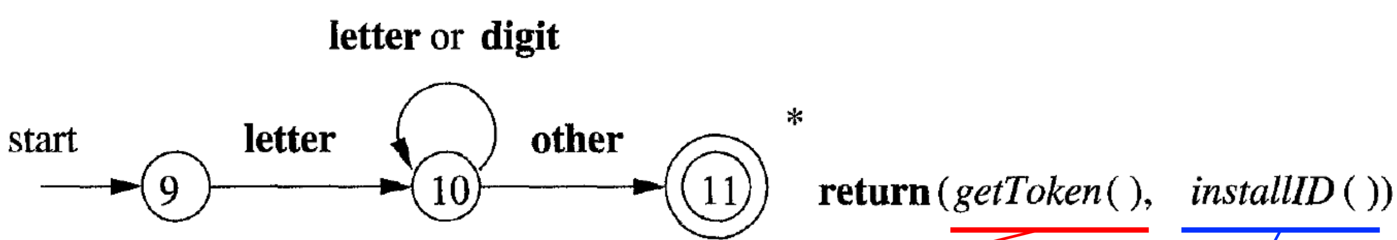
Recognition of Reserved Words and Identifiers (保留字和标识符的识别)

- 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

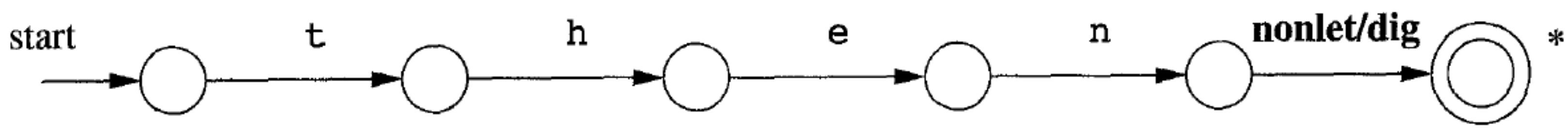


Examine the symbol table and return the token name (identifier or reserved word)

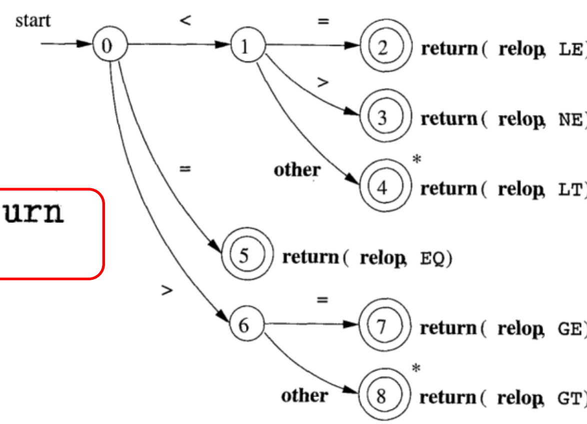
Place a recognized identifier/keyword in the symbol table if it is not already there and return a pointer to the entry

Handling Reserved Words

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



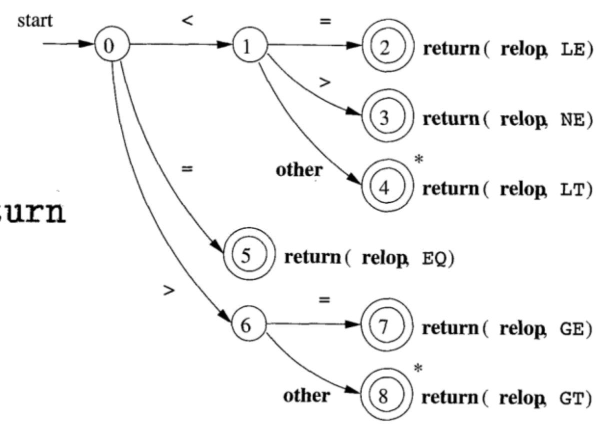
Building a Lexical Analyzer from Transition Diagrams




```
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);
        }
    }
}
```

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

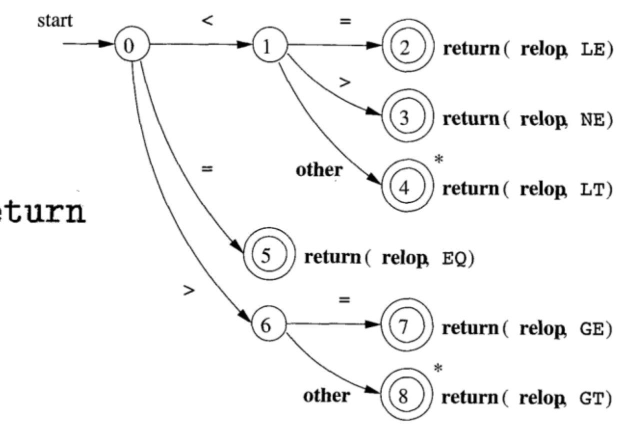



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

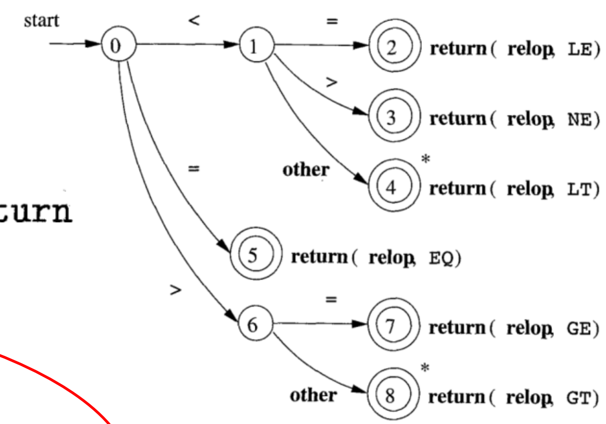



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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

A `switch` statement based on the value of `state` takes us to the processing code

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams



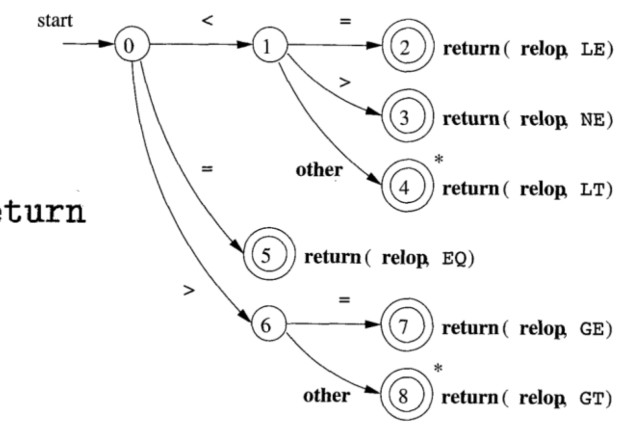

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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

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

Sketch implementation of `relop` transition diagram

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 implementation of relop transition diagram

Building the Entire Lexical Analyzer

- **Strategy 1:** Try the transition diagram for each type of token sequentially
  - `fail()` resets the pointer `forward` and tries the next diagram

- **Problem:** Not efficient
  - May need to try many irrelevant diagrams whose first edge does not match the first character in the input stream

Building the Entire Lexical Analyzer

- **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

- **Problem:** Requires special hardware for parallel simulation, may degenerate into the sequential strategy on certain machines

Building the Entire Lexical Analyzer

- **Strategy 3:** Combining all transition diagrams into one
  - Allow the transition diagram to read input until there is no possible next state
  - Take the longest lexeme that matched any pattern

- This is a commonly-adopted strategy in real-world compiler implementation (efficient & requires no special hardware)

How? Be patient ☺, we will talk about this later.

Outline

- The Role of the Lexical Analyzer
- Specification of Tokens (Regular Expressions)
- Recognition of Tokens (Transition Diagrams)
- The Lexical-Analyzer Generator (to be discussed in lab)
- 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

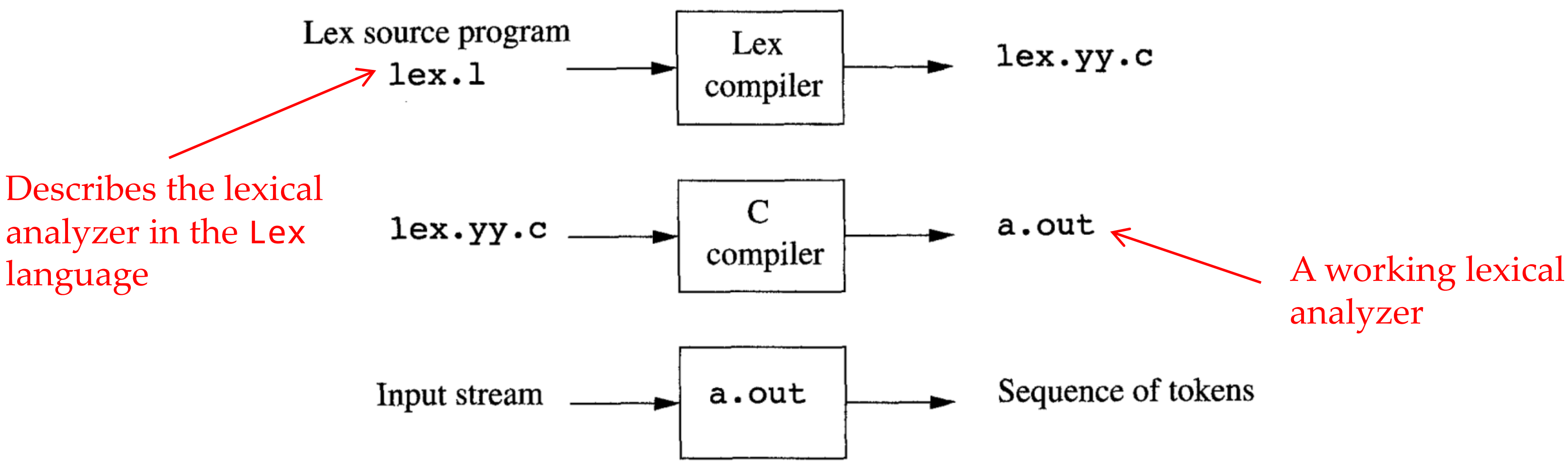


Figure 3.22: Creating a lexical analyzer with Lex

Structure of Lex Programs

- **A Lex program has three sections separated by %%**
  - Declaration (声明)
    - Variables, 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);}

%%
```

Literal strings*

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.

* The characters inside have no special meaning (even if it is a special one such as *).

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 the 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
  - DFA Minimization (Self-Study Materials)

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

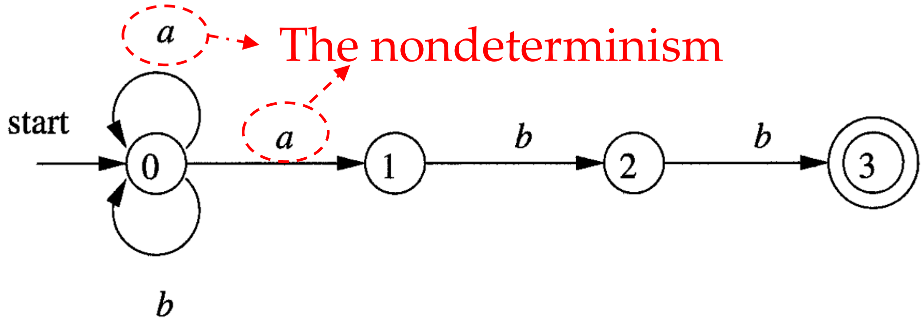
- An **NFA** is a 5-tuple, consisting of:

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

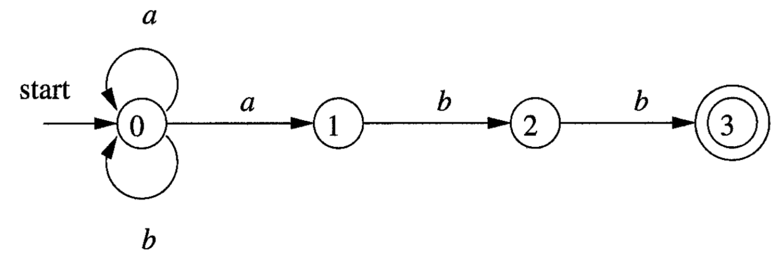
NFA Example

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

The NFA can be represented as a Transition Graph:



Transition Table

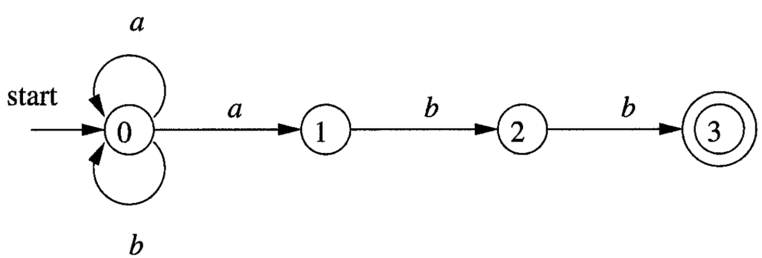


- Another representation of an NFA
  - Rows correspond to states
  - Columns correspond to the input symbols or ϵ
  - The table entry for a state-input pair lists the set of next states
  - $\emptyset$: the transition function has no info about the state-input pair

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

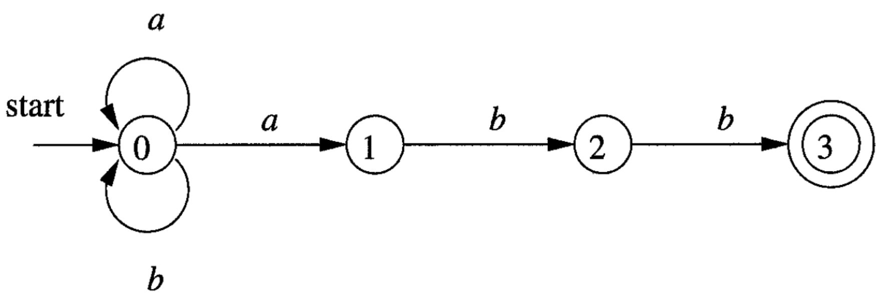
- An NFA accepts an input string x if and only if
  - There is a path in the transition graph from the start state to one accepting state, such that the symbols along the path form x (ϵ labels are ignored).



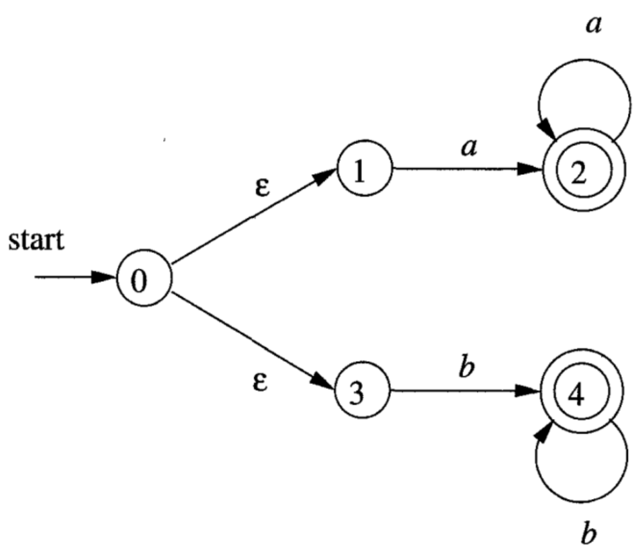
accepts the string *aabb*

- 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



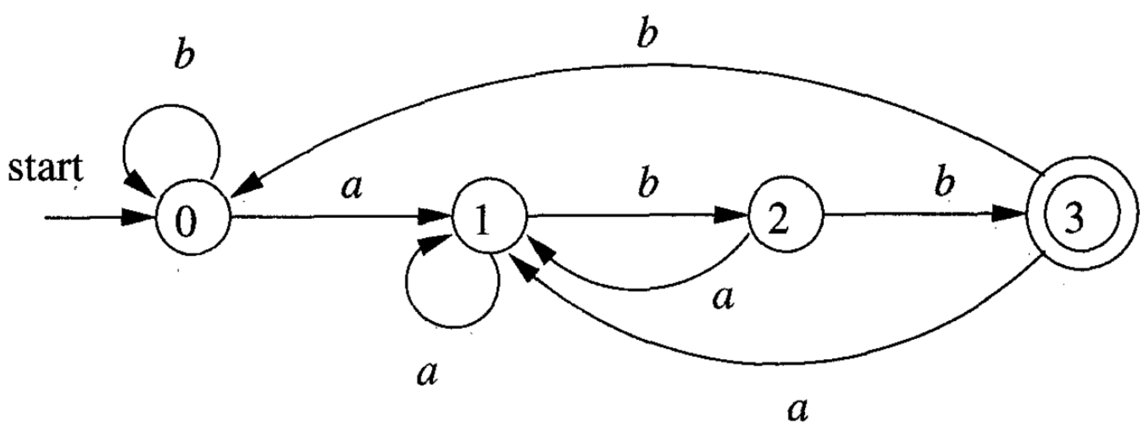
$L((\mathbf{a} \mid \mathbf{b})^*\mathbf{abb})$



$L(\mathbf{aa}^* \mid \mathbf{bb}^*)$

Deterministic Finite Automata (DFA)

- A **DFA** 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 (i.e., exactly one target state)



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

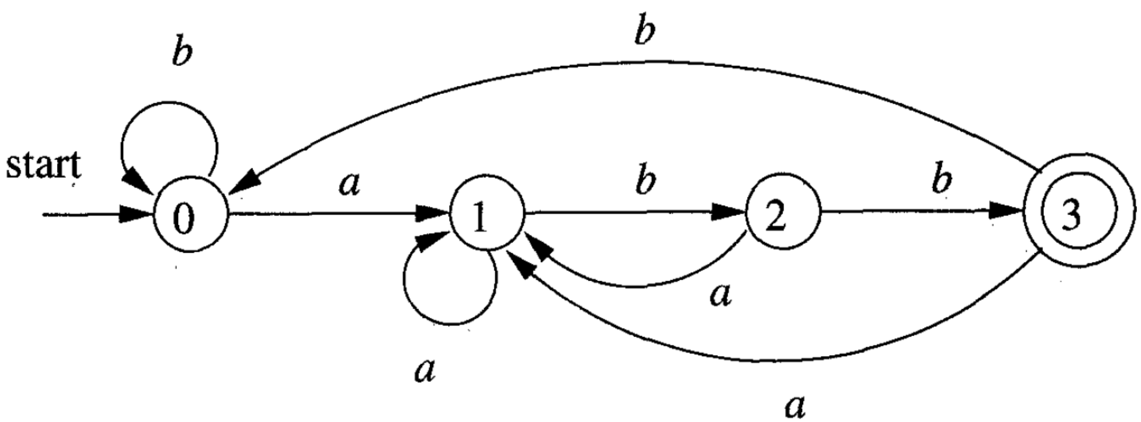
```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

We can see from the algorithm:

- DFA can efficiently accept/reject strings (i.e., recognize patterns)

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?

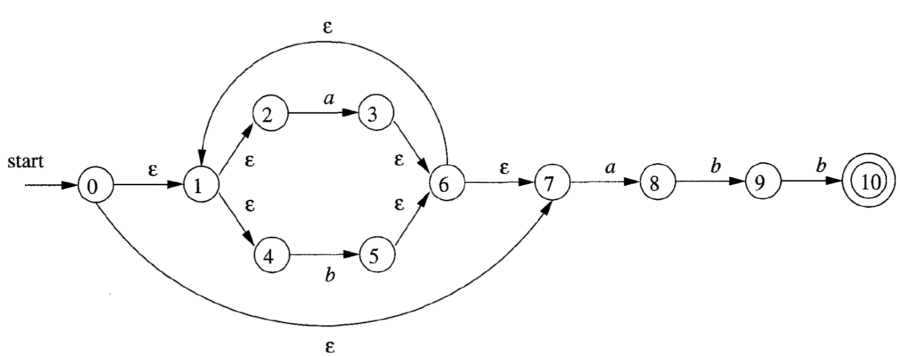
From Regular Expressions to Automata

- 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
  - Algorithms: Thompson's construction + subset construction

* 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
    - Why? Because after reading the input $a_1a_2\ldots a_n$, the DFA reaches one state while the NFA may reach multiple states
  - **Basic idea:** The algorithm simulates "in parallel" all possible moves an NFA can make on a given input string

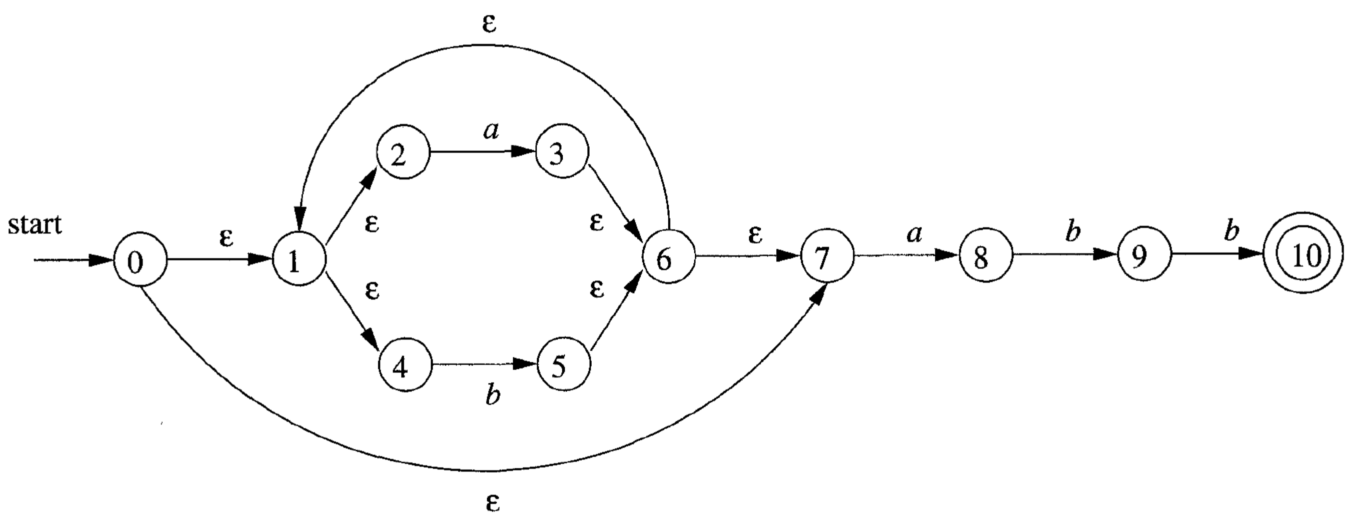


After reading "a", the NFA may reach any of these states:

3, 6, 1, 7, 2, 4, 8

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

- 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, $\bigcup_{s\ in\ T} \epsilon\text{–}closure(s)$
  - ***move*(*T, a*):** Set of NFA states to which there is a transition on input symbol a from some state s in T

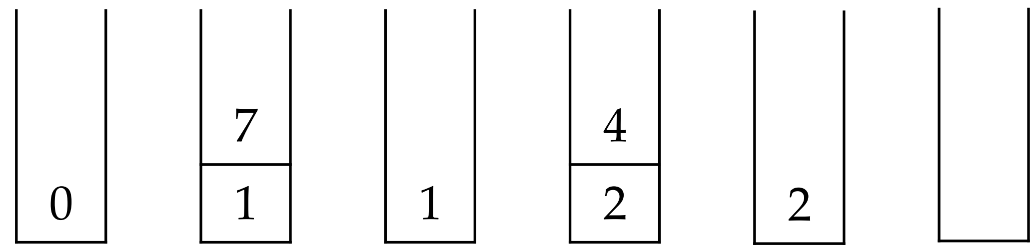
Subset Construction Technique

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

```
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;
                }
}
```

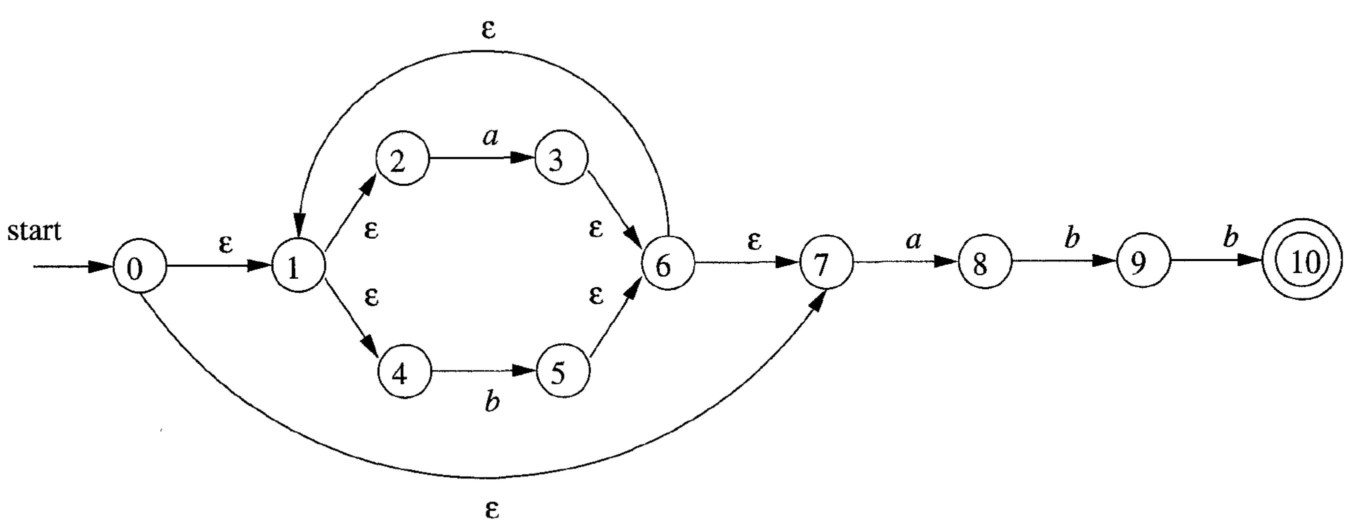
Illustrative Example

- ϵ-*closure*(0) = ?



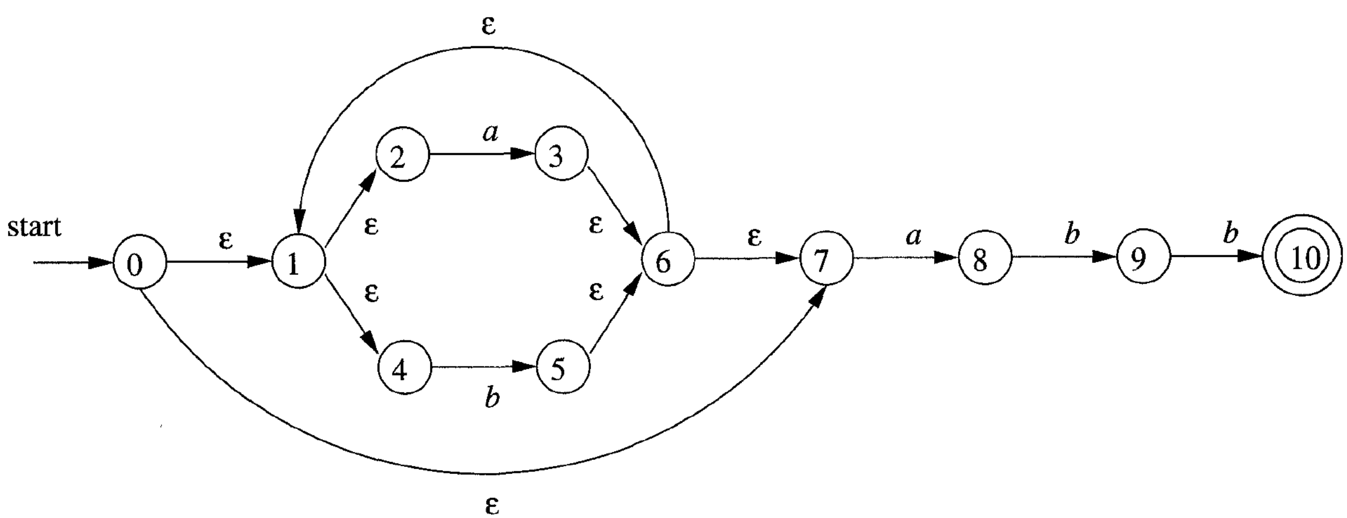
Push 0 | Pop 0, Push 1, 7 | Pop 7 | Pop 1, Push 2, 4 | Pop 4 | Pop 2

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



Exercise

- ϵ-*closure*({3, 8}) = ?



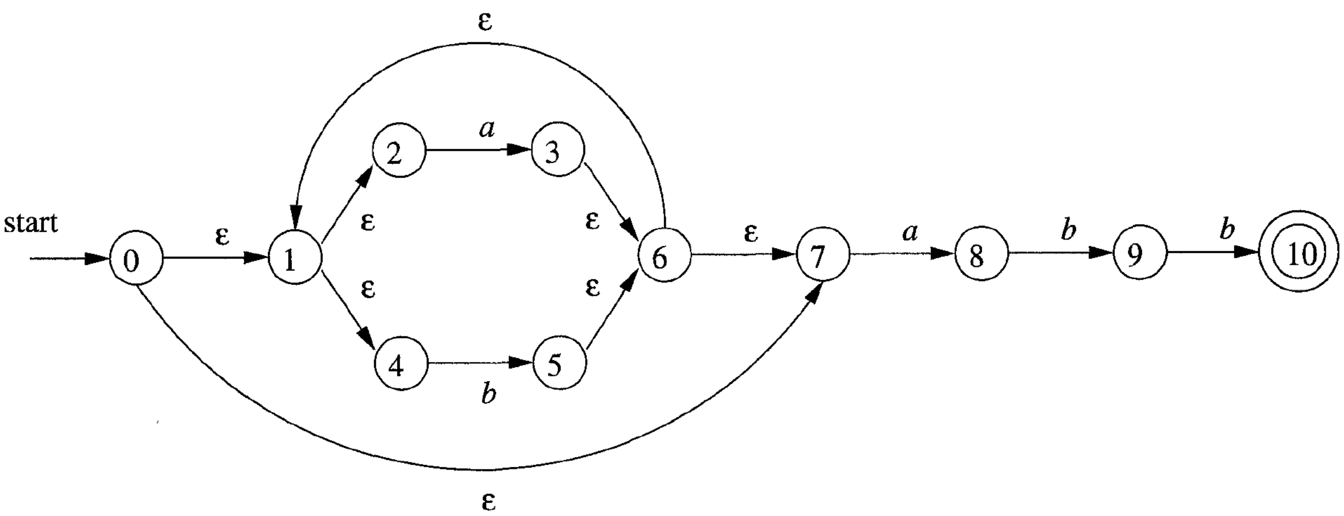
Subset Construction Technique Cont.

- The construction of the DFA *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;
        }
}
```

Illustrative Example

- Initially, Dstates only has one unmarked state:
  - ϵ-*closure*(0) = {0, 1, 2, 4, 7} -- A
- Dtran is empty



Illustrative Example

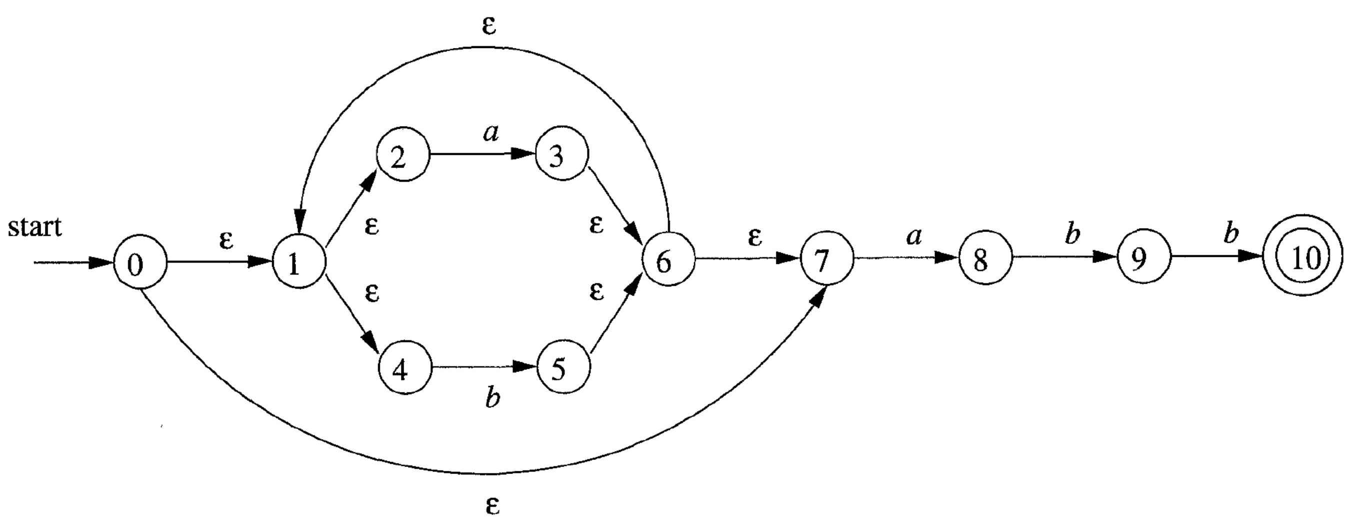
{0, 1, 2, 4, 7} -- A

ϵ-closure(*move*[A, a])

= *ϵ-closure*({3, 8})

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

- We get an unseen state {1, 2, 3, 4, 6, 7, 8} -- B
- Update Dstates: {A, B}
- Update Dtran: {[A, a] ➔ B}



Illustrative Example

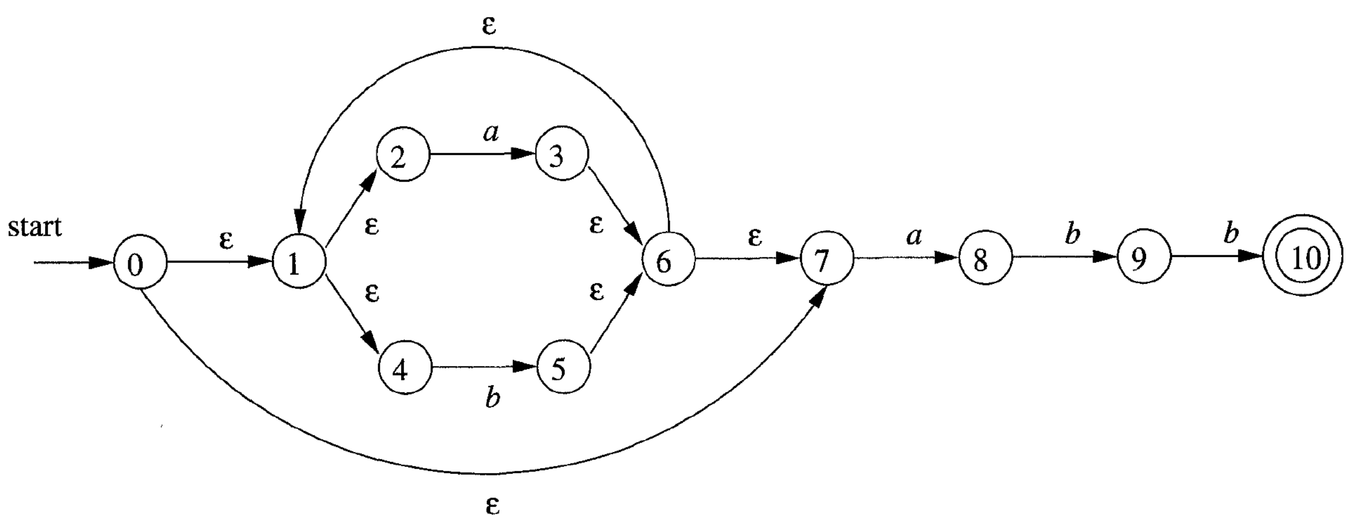
{0, 1, 2, 4, 7} -- A

ϵ-closure(*move*[A, b])

= *ϵ-closure*({5})

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

- We get an unseen state {1, 2, 4, 5, 6, 7} -- C
- Update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C}



Illustrative Example

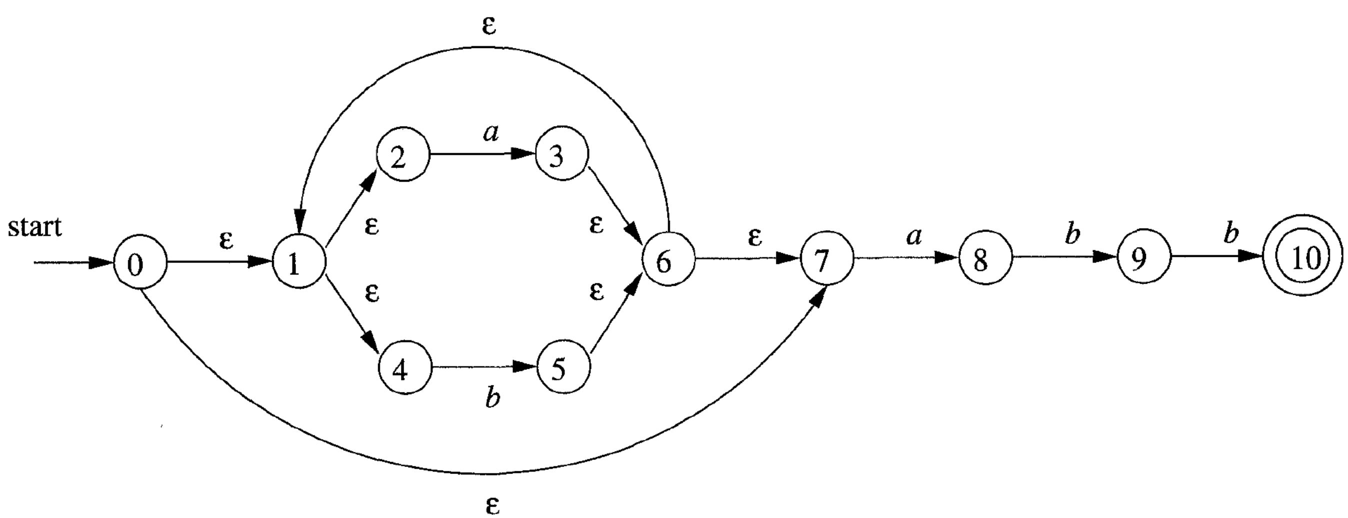
{1, 2, 3, 4, 6, 7, 8} -- B

ϵ-closure(*move*[B, a])

= *ϵ-closure*({3, 8})

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

- The state {1, 2, 3, 4, 6, 7, 8} already exists (B)
- No need to update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C, [B, a] ➔ B}



Illustrative Example

- Eventually, we will get the following 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
$\{1, 2, 4, 5, 6, 7, 10\}$	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

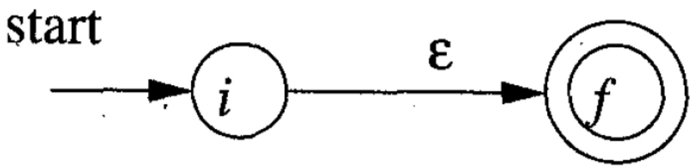
Thompson's construction algorithm (Thompson构造法)

- The algorithm works recursively by splitting a regular expression into subexpressions, from which the NFA will be constructed using the following rules:
  - **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

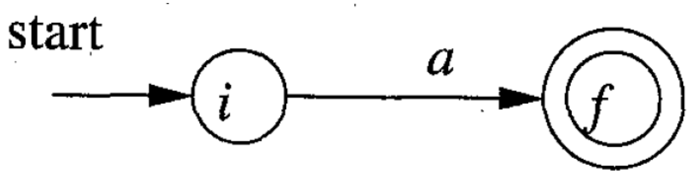
Thompson's Construction Algorithm

Two basis rules:

1. The empty expression ϵ is converted to



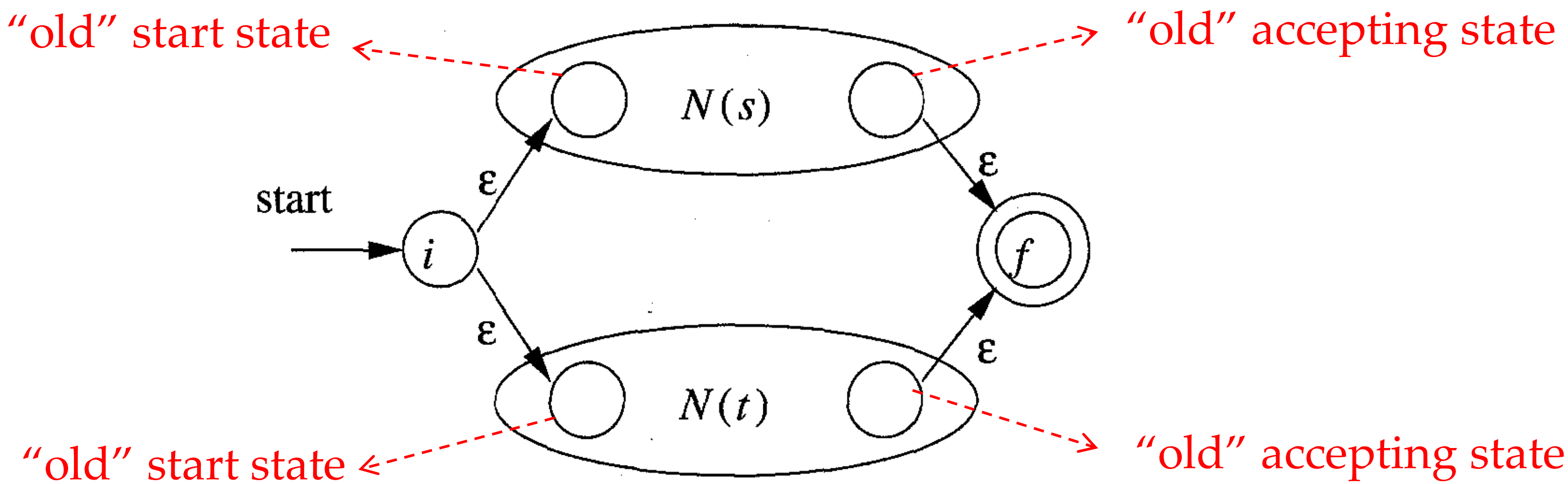
2. Any subexpression a (a single symbol in input alphabet) is converted to



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

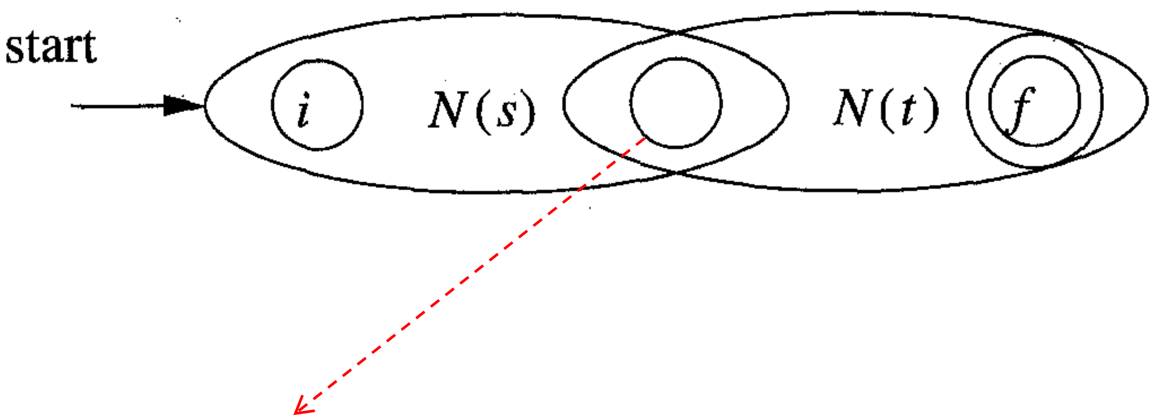


By construction, the NFA's have only one start state and one accepting state

Thompson's Construction Algorithm

The inductive rules: the concatenation case

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

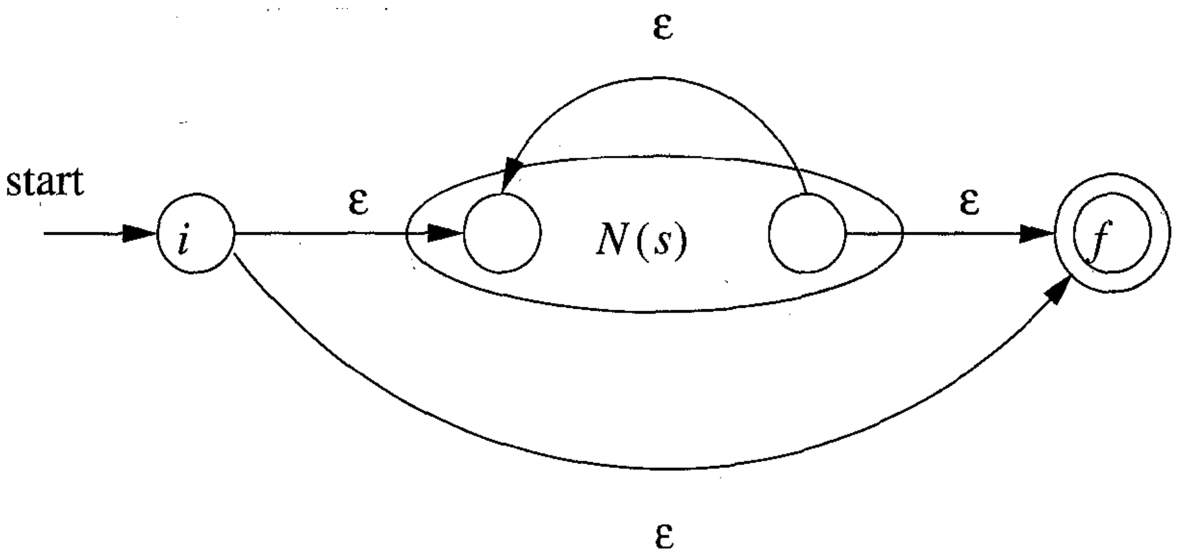


Merging the accepting state of $N(s)$ and the start state of $N(t)$

Thompson's Construction Algorithm

The inductive rules: the Kleene star case

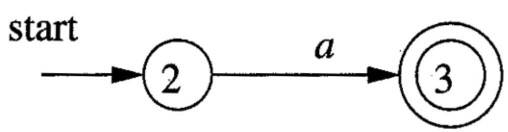
- s^* : $N(s)$ is the NFA for subexpression s



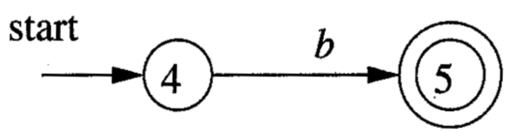
Example

Use Thompson's algorithm to construct an NFA for the regexp r = **(a|b)*a**

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

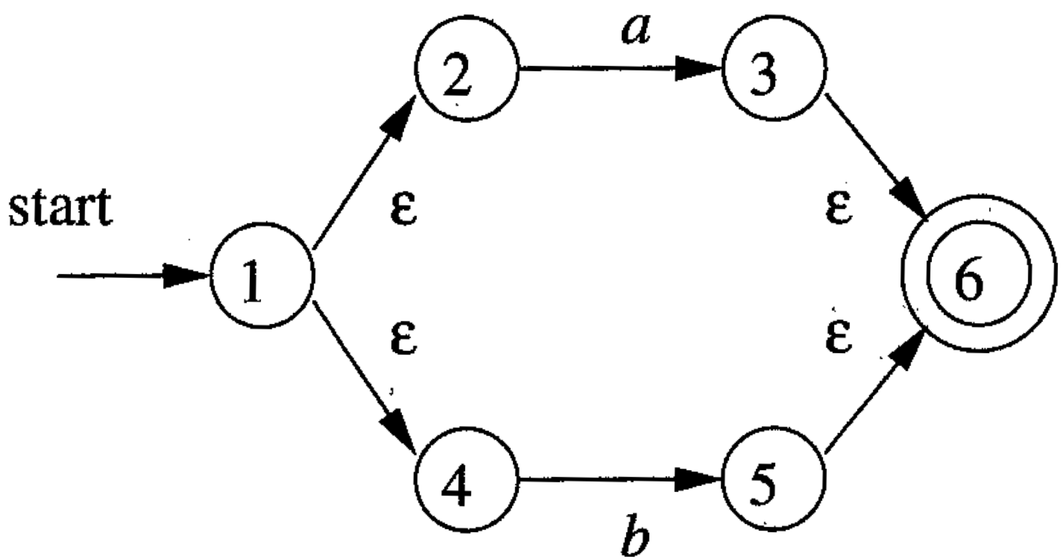


2. NFA for the first **b** (apply basis rule #1)



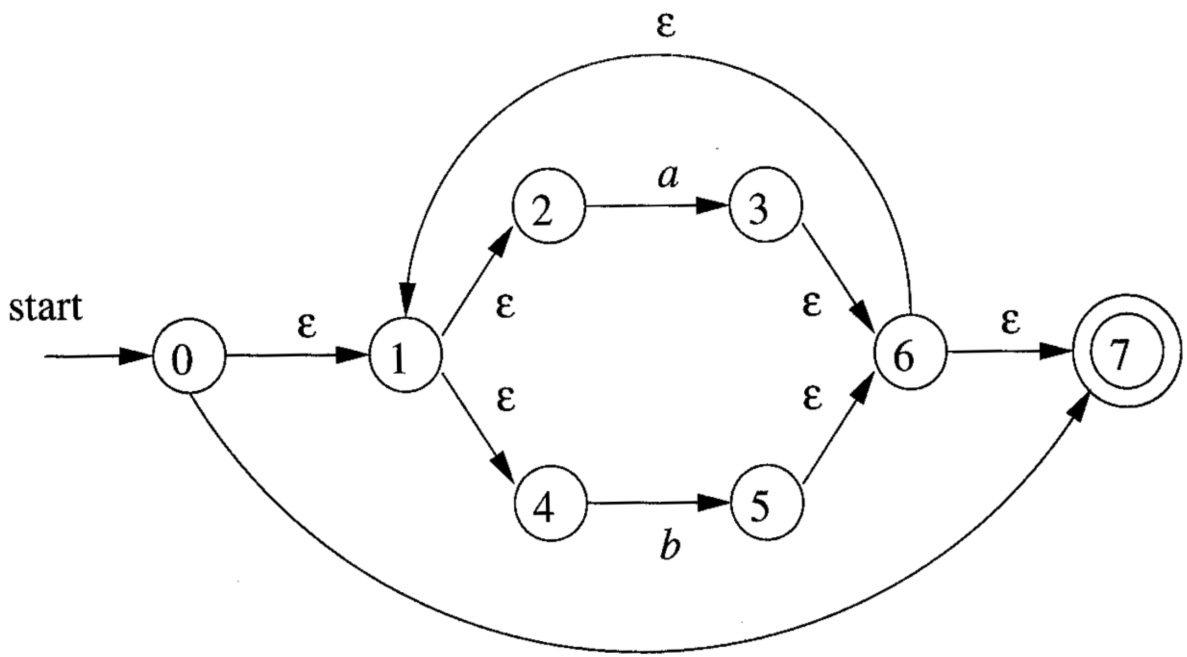
Example $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$

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



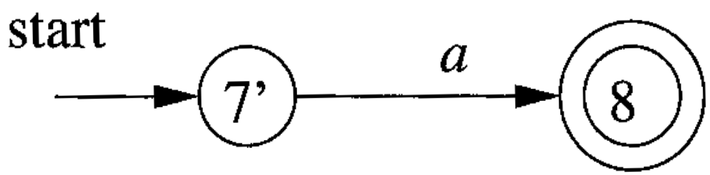
Example $r = (\mathbf{a}|\mathbf{b})^*\mathbf{a}$

4. NFA for $(\mathbf{a}|\mathbf{b})^*$ (apply inductive rule #3)



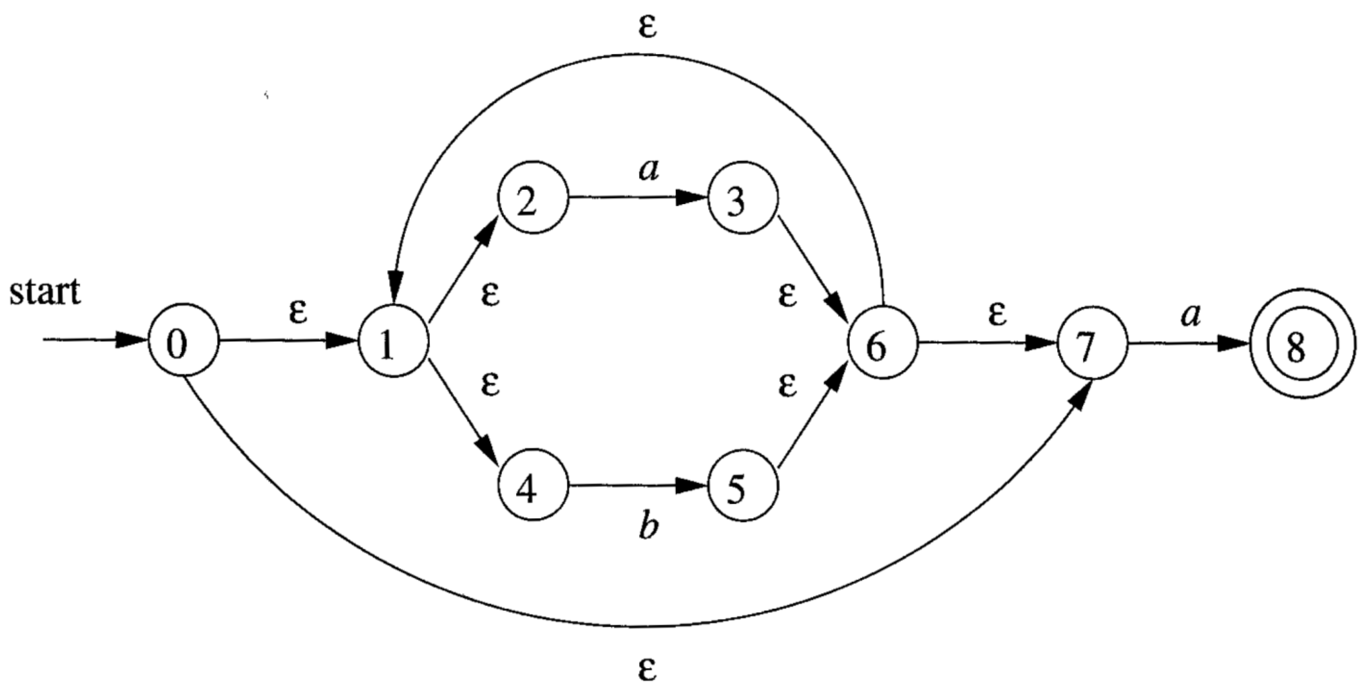
Example $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$

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



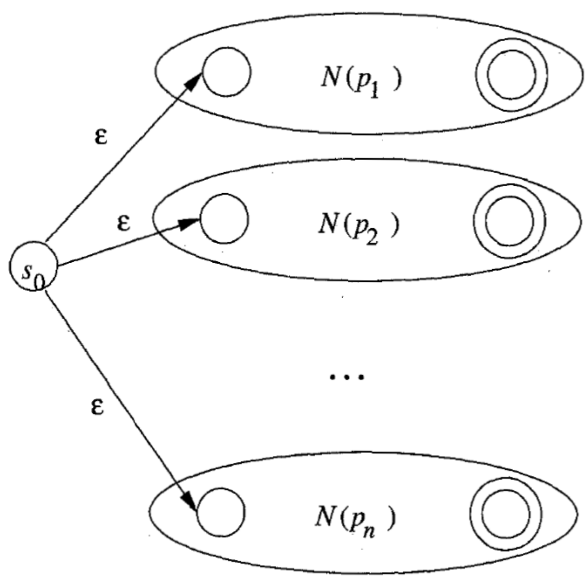
Example $r = (\mathbf{a}|\mathbf{b})^{*}\mathbf{a}$

6. NFA for $(\mathbf{a}|\mathbf{b})^{*}\mathbf{a}$ (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 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

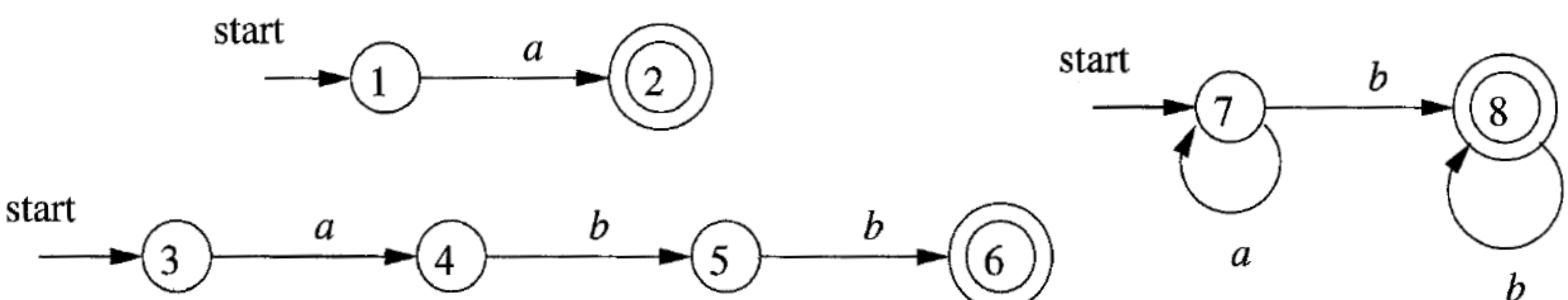
DFA's for Lexical Analyzers

- 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

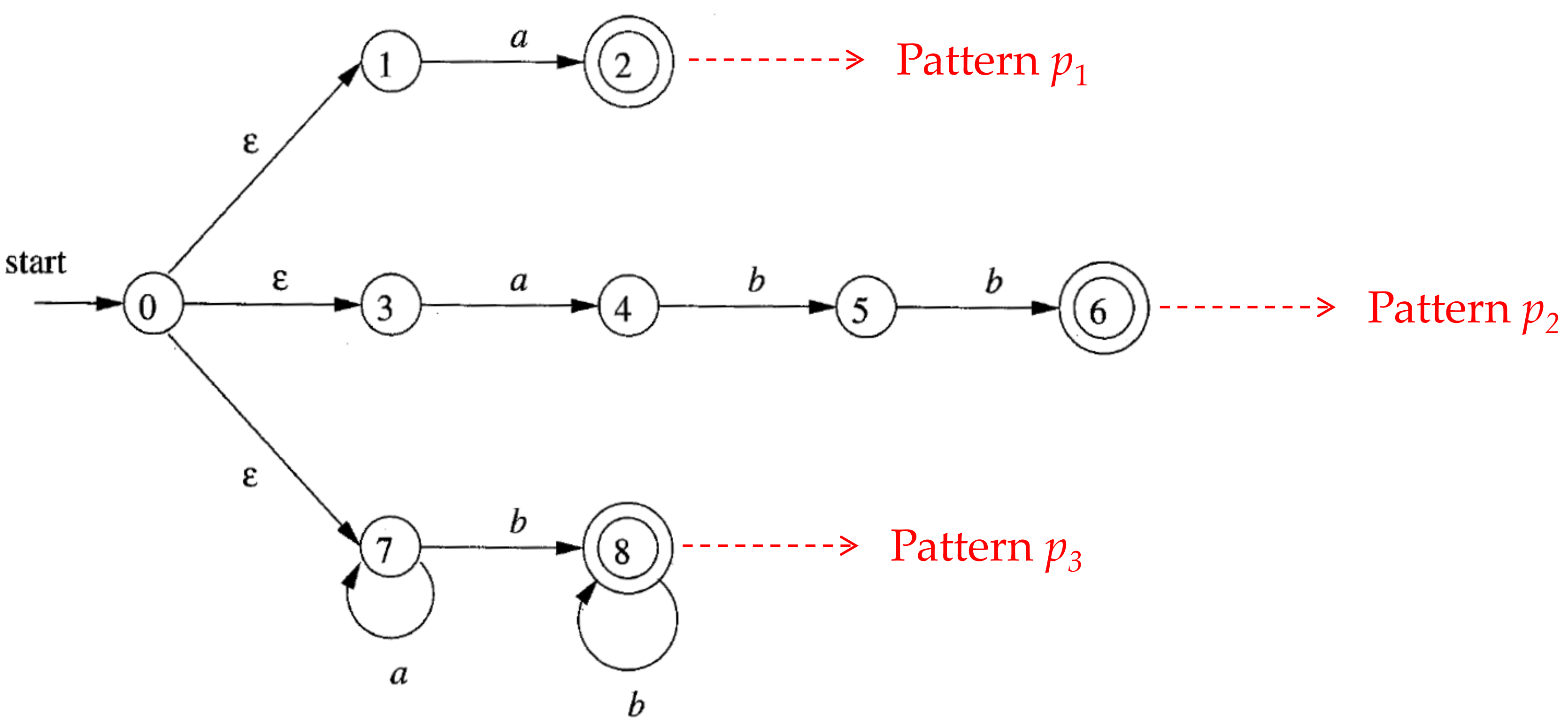
Example

- 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}
  - **a*b$^+$** {action A_3 for pattern p_3}
- We first construct an NFA for each pattern



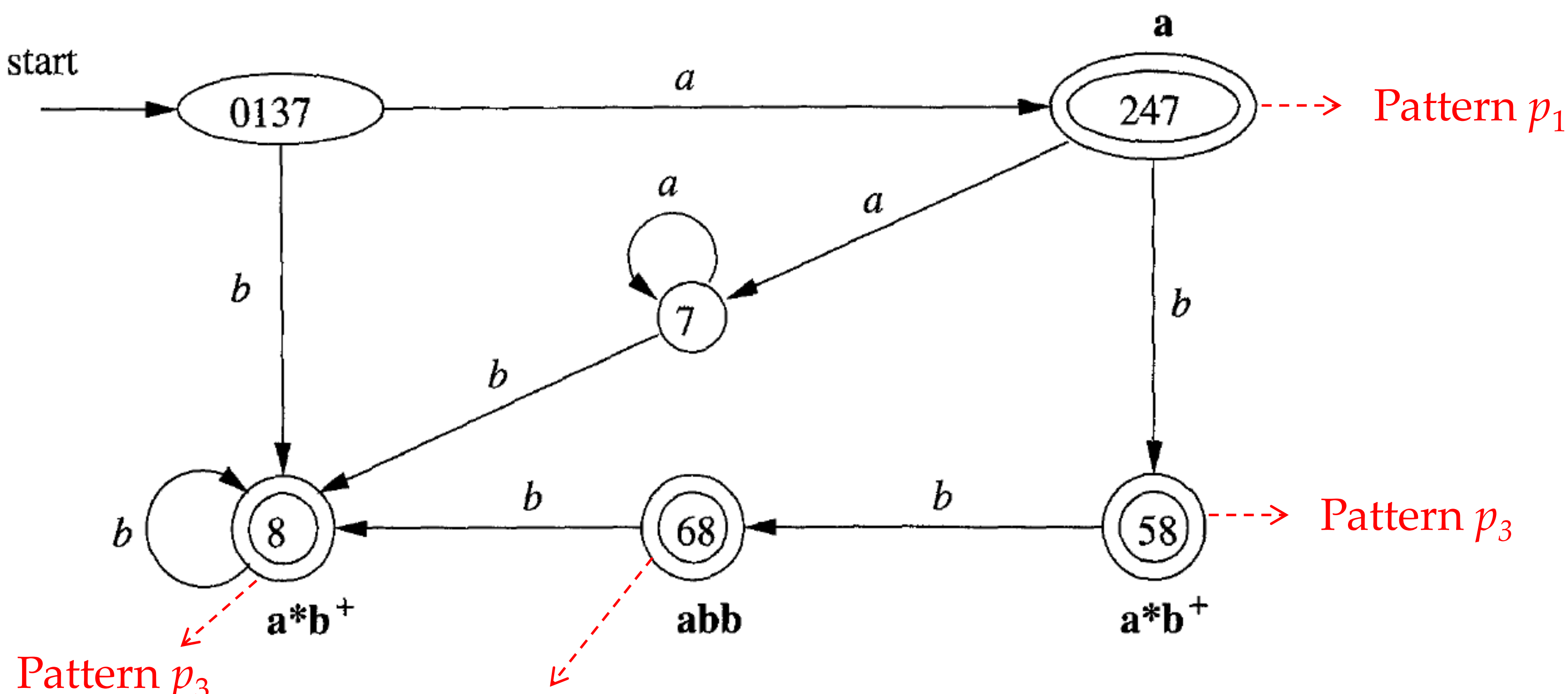
Example

- Combining the three NFA's



Example

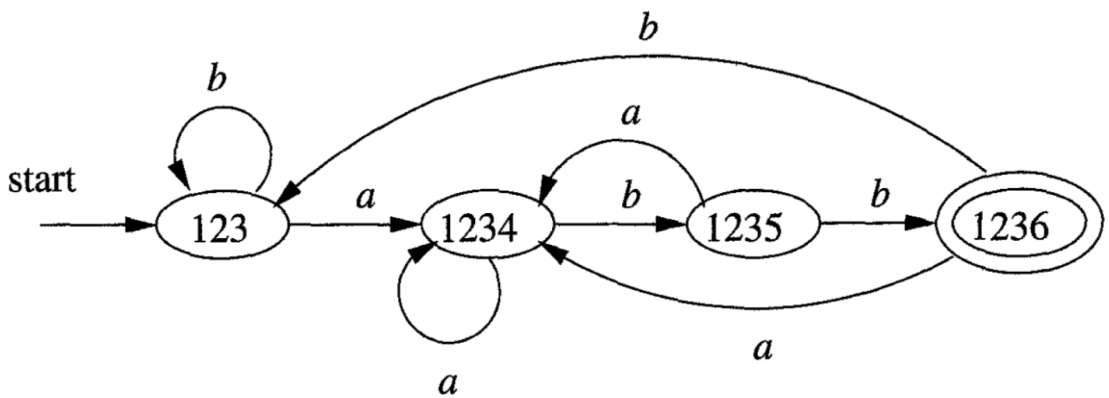
- Converting the big NFA to a DFA

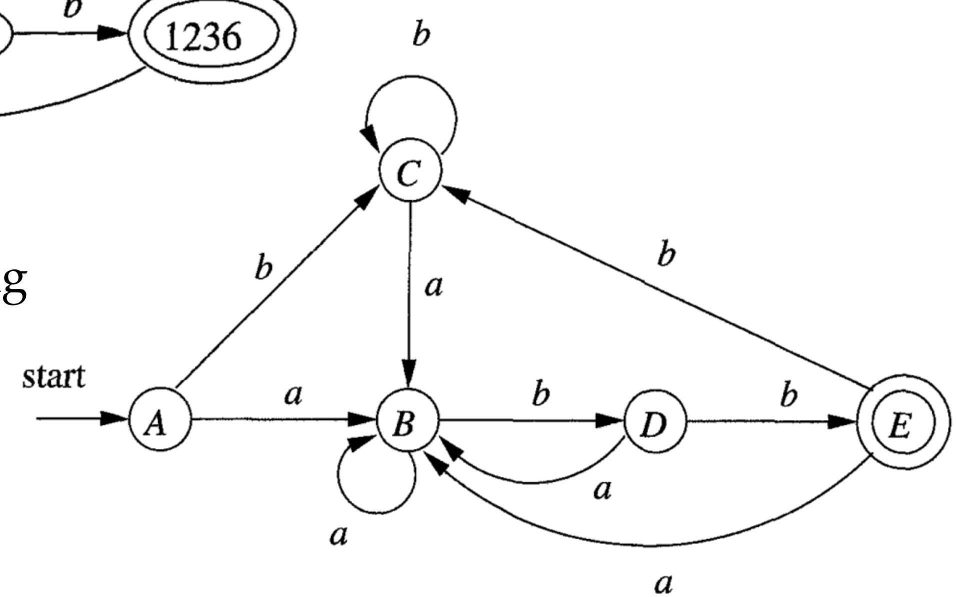


6 and 8 are two accepting NFA states corresponding to two patterns. We choose Pattern p2, which is specified before p3

Minimizing # States of a DFA*

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



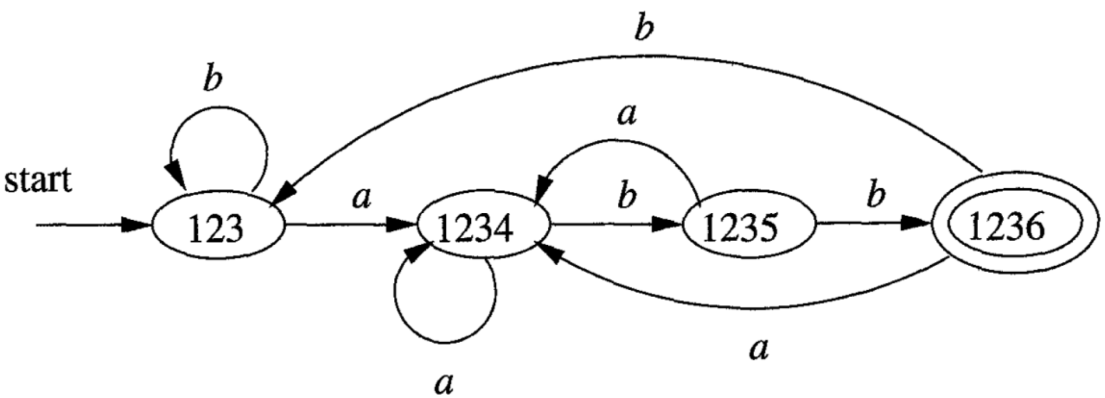


Two equivalent DFA's, both recognizing regular language $L((\mathbf{a}|\mathbf{b})^*\mathbf{abb})$

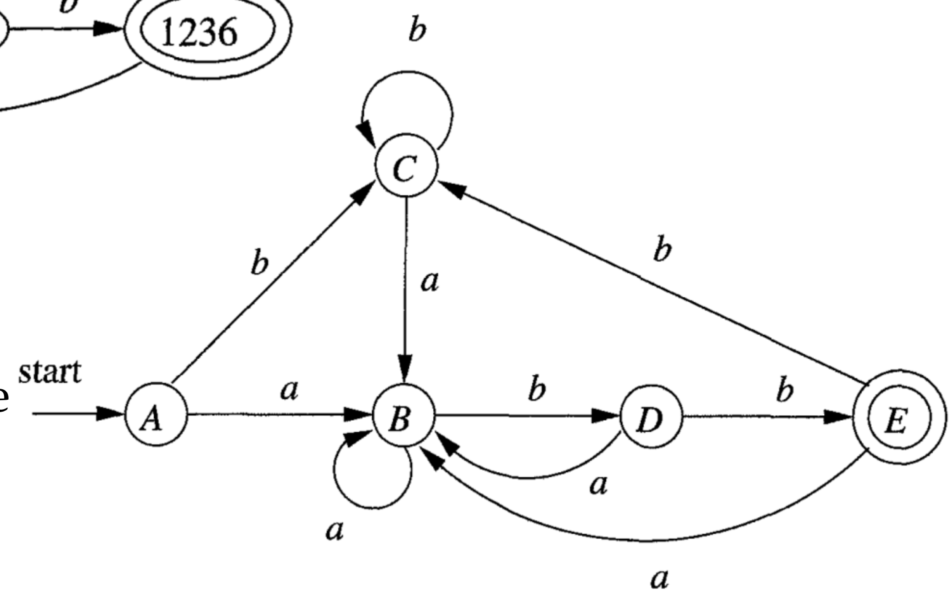
*Self-study materials

Minimizing # States of a DFA Cont.

- 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

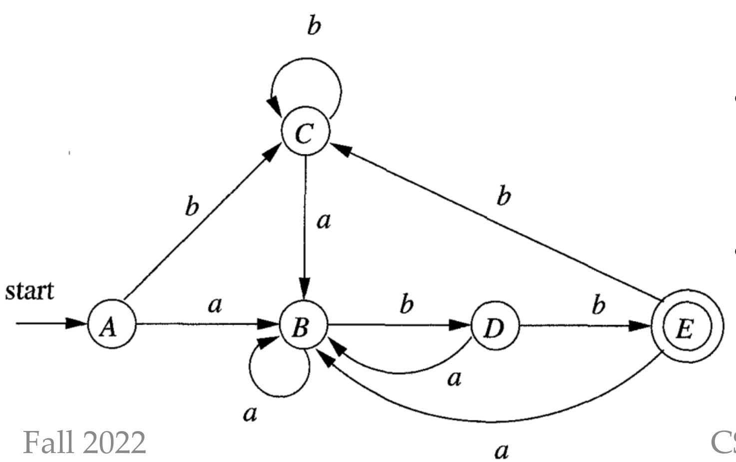
Minimizing # States of a DFA Cont.

- 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
- **For two indistinguishable states,** scanning any string will lead to the same state. Such states are equivalent and should 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 an 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 Π_new = 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 Π_new by the set of all subgroups formed;
}
```

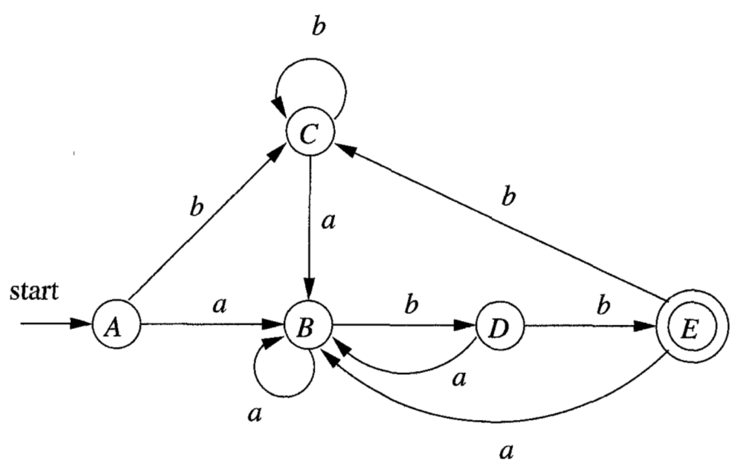
3. If $\prod_{new}$ == $\prod$, let $\prod_{final}$ = $\prod$ and the partitioning finishes; Otherwise, $\prod$ = $\prod_{new}$ and repeat step 2

The Construction Part

- Choose one state in each group of $\prod_{\text{final}}$ 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 $\prod_{\text{final}}$; 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



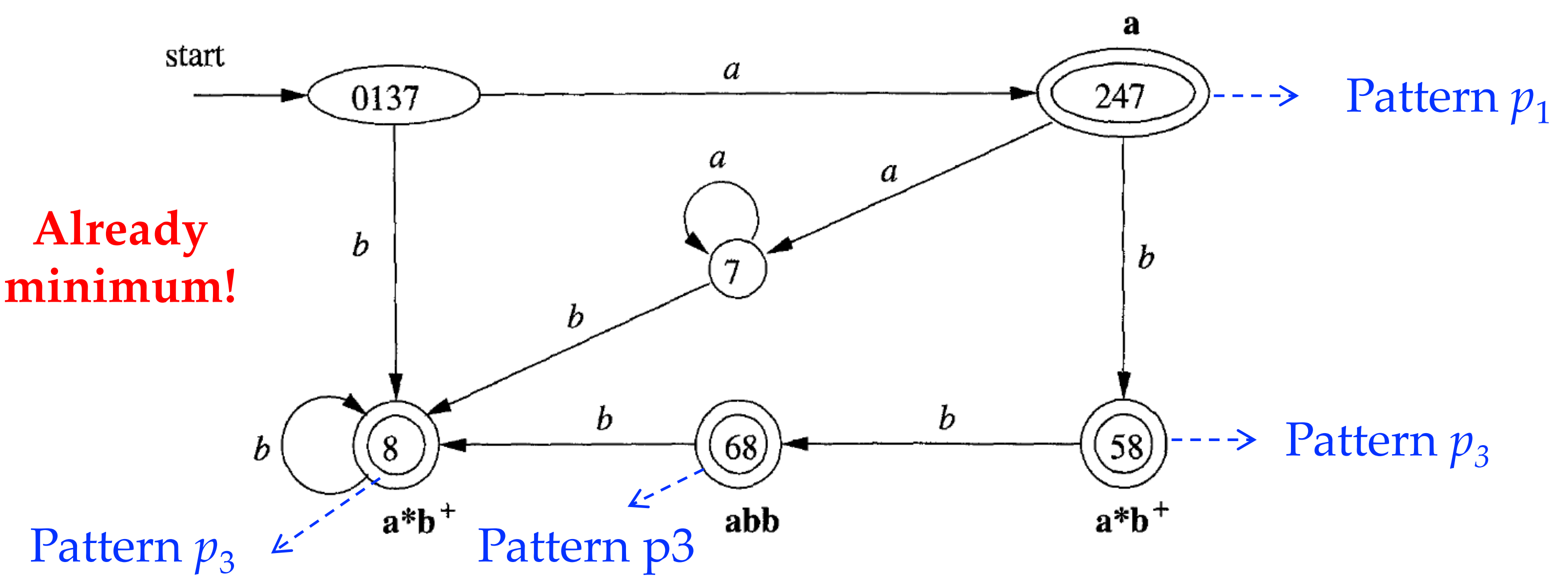
STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A

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.



Reading Tasks

- Chapter 3 of the dragon book
  - 3.1 The role of the lexical analyzer
  - 3.3 Specification of tokens
  - 3.4 Recognition of tokens
  - 3.5 The lexical-analyzer generator Lex
  - 3.6 Finite automata
  - 3.7 From regular expressions to automata
  - 3.8 Design of a lexical analyzer generator
    - 3.8.1 – 3.8.3, the remaining can be skipped