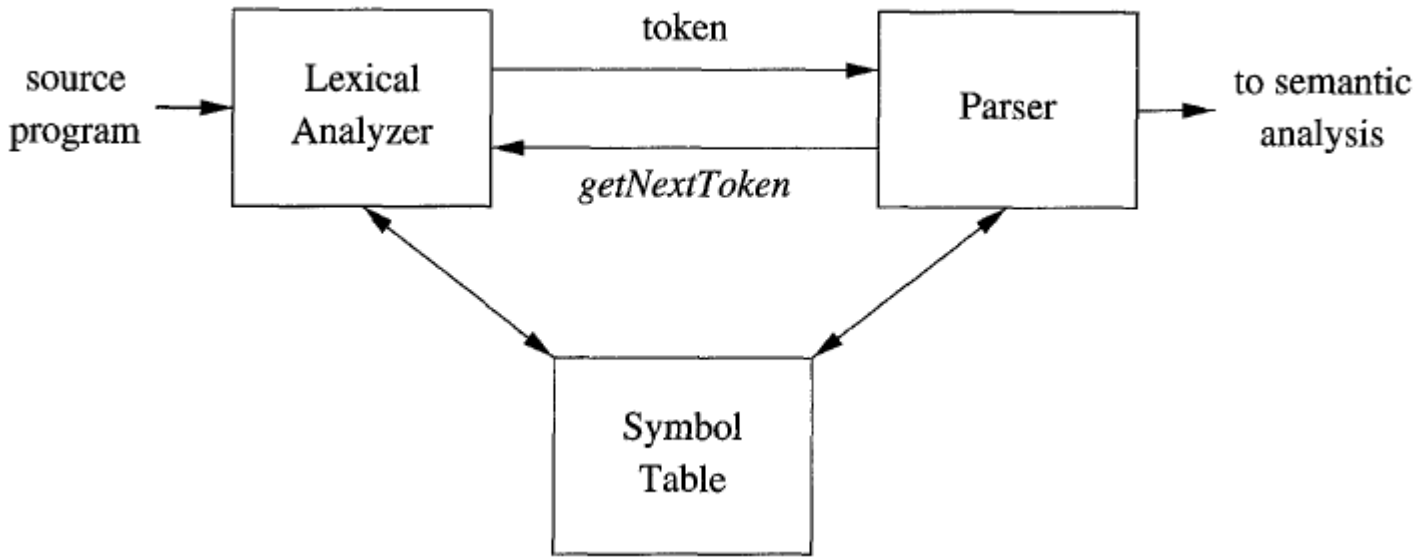
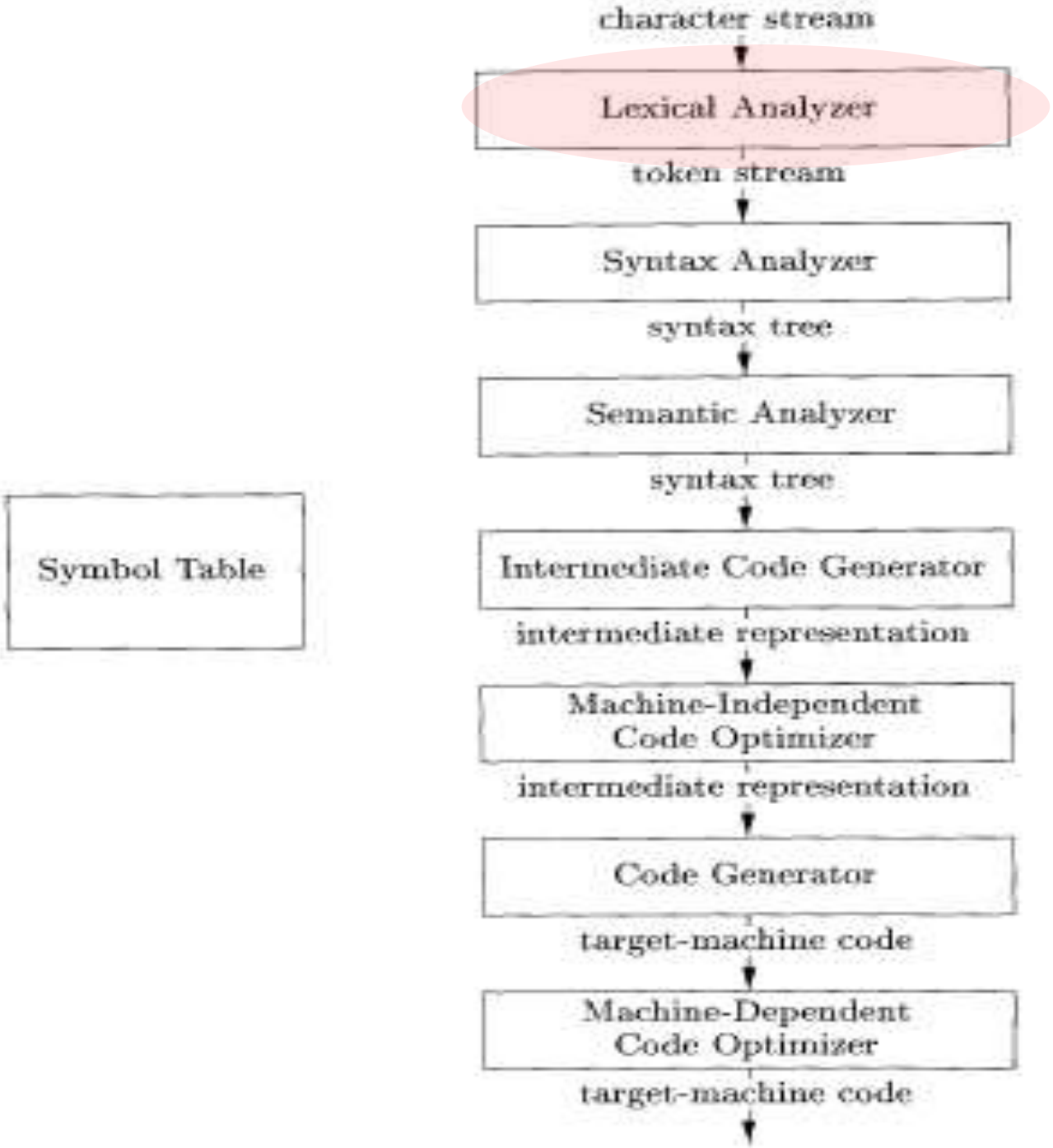
東南大學
SOUTHEAST UNIVERSITY

Compiler Principle
——Lexical Analysis

Zhizheng Zhang
Southeast University

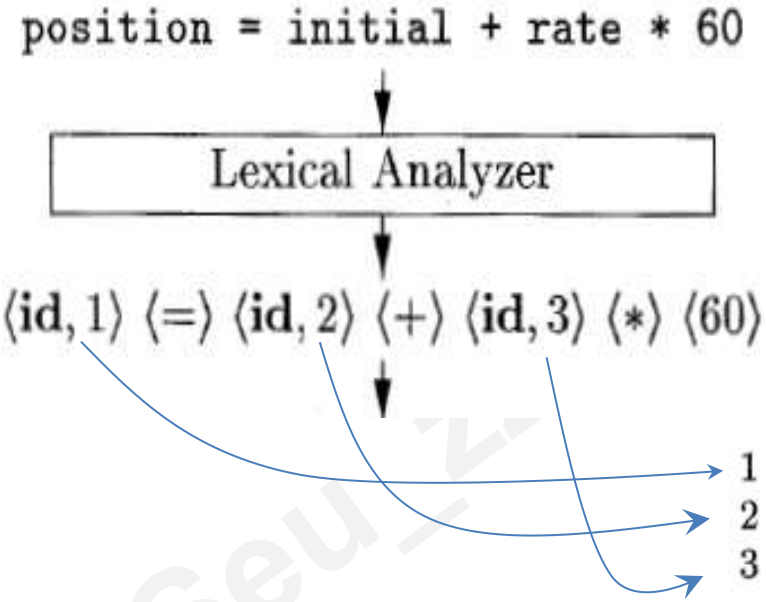
1. The Role of Lexical Analyzer




character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
target-machine code
Symbol Table

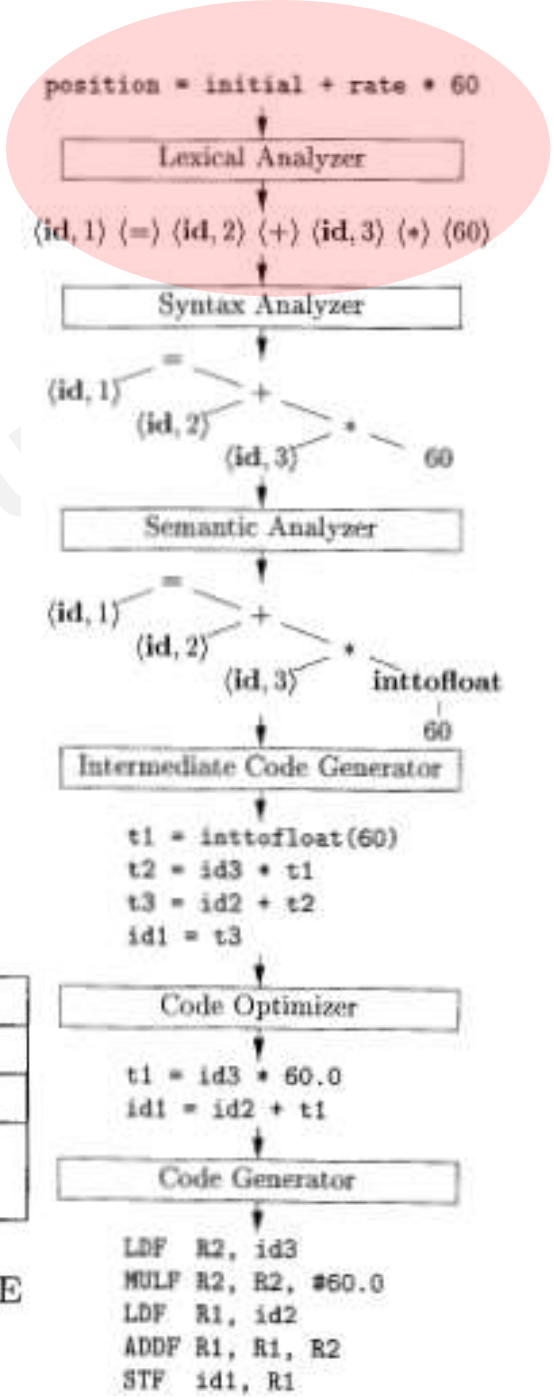
Example.

position = initial + rate * 60



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Sometimes, lexical analyzers are divided into a cascade of two processes:

1. *Scanning consists of the simple processes that do not require tokenization* of the input, such as deletion of comments, compaction of consecutive whitespace characters into one, generation of error messages, and the expansion of macros.

2. *Lexical analysis, where the scanner* produces the sequence of tokens as output.

1. 1 Scanning vs. Parsing

Why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration.
2. Compiler efficiency is improved.
3. Compiler portability is enhanced.

1. 2 Tokens, Patterns, and Lexemes

I. ***A token*** *is a pair consisting of a token name and an optional attribute* value.

II. ***A pattern*** *is a description of the form that the lexemes of a token may take.*

III. ***A lexeme*** *is a sequence of characters in the source program that matches t*he pattern for a token and is identified by the lexical analyzer as an instance of that token.

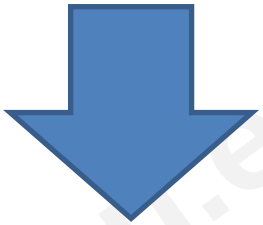
TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything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

1. 3 Attributes for Tokens

The most important example is the token id, where we need to associate with the token a great deal of information, e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.

Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

E = M * C ** 2



<**id**, pointer to symbol-table entry for E>
< assign-op >
<**id**, pointer to symbol-table entry for M>
<**mult -op**>
<**id**, pointer to symbol-table entry for C>
<**exp-op**>
<**number** , integer value 2 >

1. 4 Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components.

E.g., fi (b == f(x)) …

The lexical Analyzer cannot tell fi is wrong without the help of parser.

If the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. Possible error-recovery actions are:

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

3. Replace a character by another character.

4. Transpose two adjacent characters.

Exercise. 3.1.2

Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

Here is a photo of <B>my house</B>:
<P><IMG SRC = "house. gif I1>
See <A HREF = "morePix. Html">More Pictures</A> if you liked that one. <P>

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

2. Input Buffering

Problem:
How to read the source program fast and correctly.

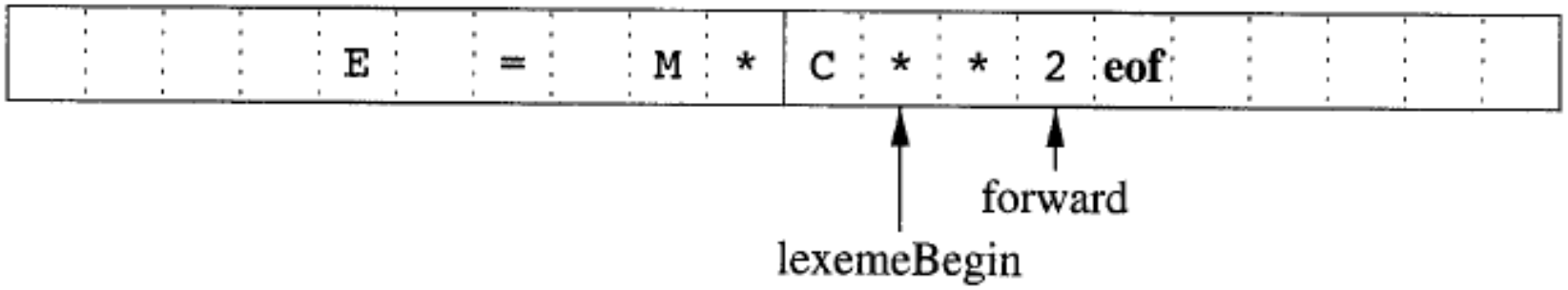
Difficult:
Have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

For instance,

I. We cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit.

II. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.

2.1 Buffer Pairs



Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes

Two pointers to the input are maintained:

I. Pointer *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine.

II. Pointer *forward* scans ahead until a pattern match is found.

Works in this way…

I. Once the next lexeme is determined, *forward* is set to the character at its right end.

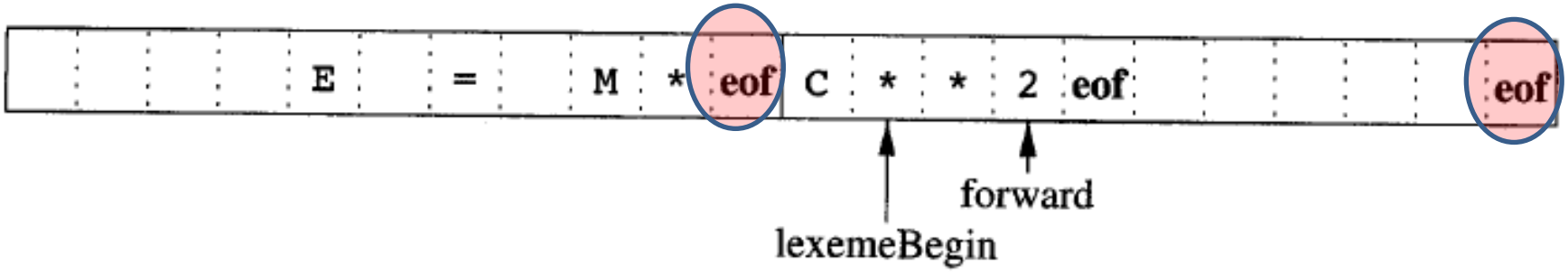
II. After the lexeme is recorded as an attribute value of a token returned to the parser, *lexemeBegin* is set to the character immediately after the lexeme just found.

III. Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer

NOTE: As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

2.2 Sentinels

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.



Sentinels at the end of each buffer

```
switch ( *forward++ ) {
        case eof:
                if (forward is at end of first buffer ) {
                        reload second buffer;
                        forward = beginning of second buffer;
                }
                else if (forward is at end of second buffer ) {
                        reload first buffer;
                        forward = beginning of first buffer;
                }
                else /* eof within a buffer marks the end of input */
                        terminate lexical analysis;
                break;
        Cases for the other characters
}
```

Lookahead code with sentinels

3. Specification of Tokens

- Regular Expressions
- Regular Grammar/Regular Definition

3.1 Regular Expression

- **Example.** Identifier Token

$letter\_(letter\_|digit)^*$

Regular Expression and its language

BASIS: Given an alphabet table Σ, there are two rules that form the basis:

1. ε is a regular expression, and $L(\varepsilon)$ is $\{\varepsilon\}$.
2. If a is a symbol in Σ, then a is a regular expression, and $L(a) = \{a\}$.

INDUCTION: Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
2. (r)(s) is a regular expression denoting $L(r)\,L(s)$.
3. $(r)^*$ is a regular expression denoting $(L\,(r))^*$.
4. (r) is a regular expression denoting $L(r)$. That means, the pairs of parenthesis cannot change the language they denote.

PRECEDENCE

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) | has lowest precedence and is left associative.

Example : Let Σ = {a, b}.

1. **The regular expression** $a|b$ **denotes the language {**a, b**}.**

2. $(a|b)(a|b)$ **denotes {**aa, ab, ba, bb**}. Another regular expression for the same language is** $aa|ab|ba|bb$**.**

3. a^* **denotes the language consisting of all strings of zero or more a's, that is,** $\{\varepsilon, a, aa, aaa, \ldots\}$.

4. $(a|b)^*$ **denotes** $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. **Another regular expression for the same language is** $(a^*b^*)^*$.

5. $a|a^*b$ **denotes the language {**$a, b, ab, aab, aaab, \ldots$**}.**

A language that can be defined by a regular expression is called a *regular set*.

If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$. For instance, $(a|b) = (b|a)$.

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

3.2 Regular Definition

If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

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

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Example

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

Example

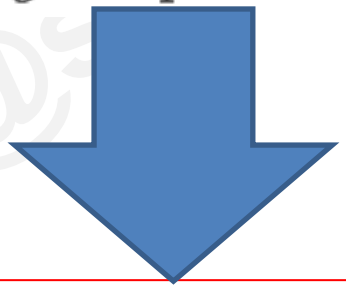
$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{optionalFraction} & \rightarrow & \mathtt{.}\ \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow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

NOTE: for convenient description, we also use

- r^+ **to denote** $r*r$**;**
- $[abc]$ **to denote** $a|b|c$**.**
- $L(r?)$ **to denote** $L(r)\cup\{\varepsilon\}$.

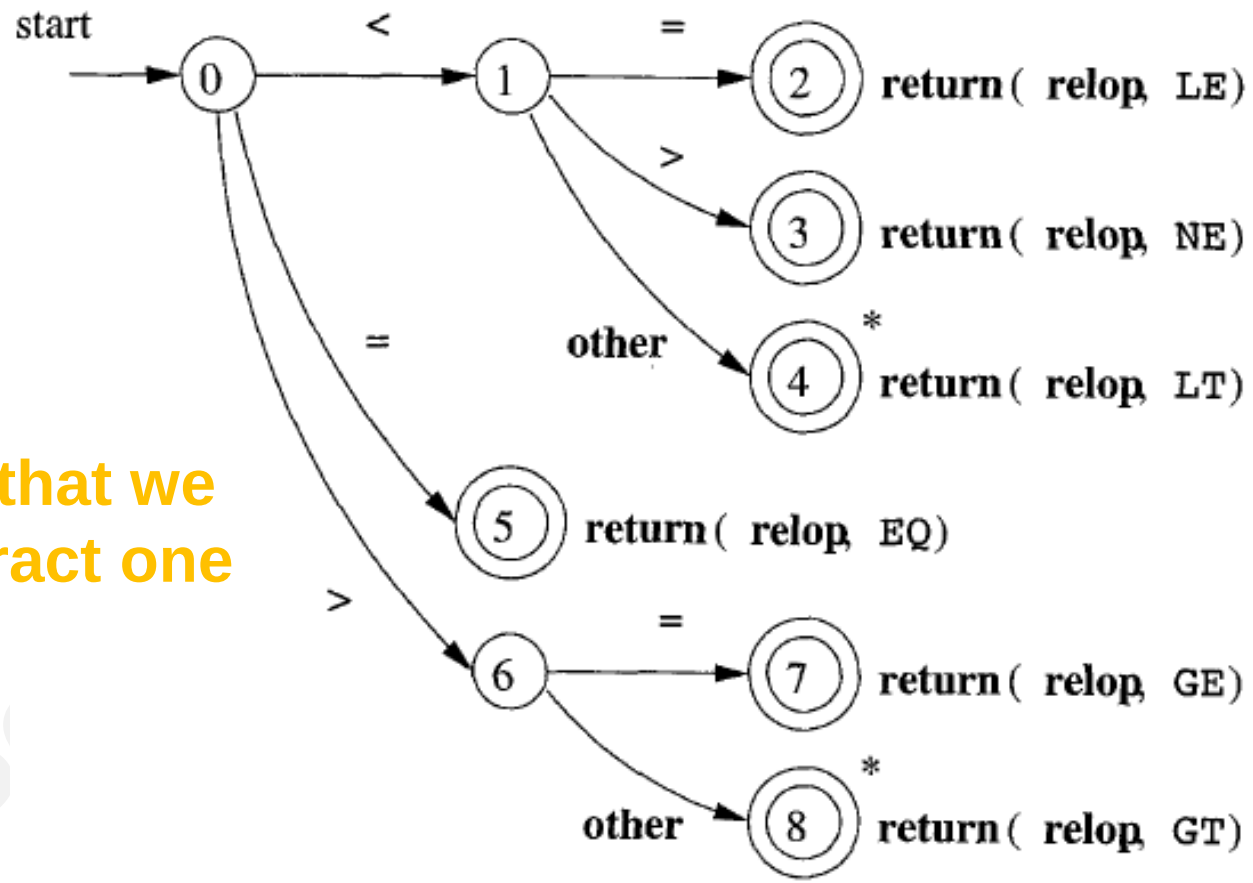
Example

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

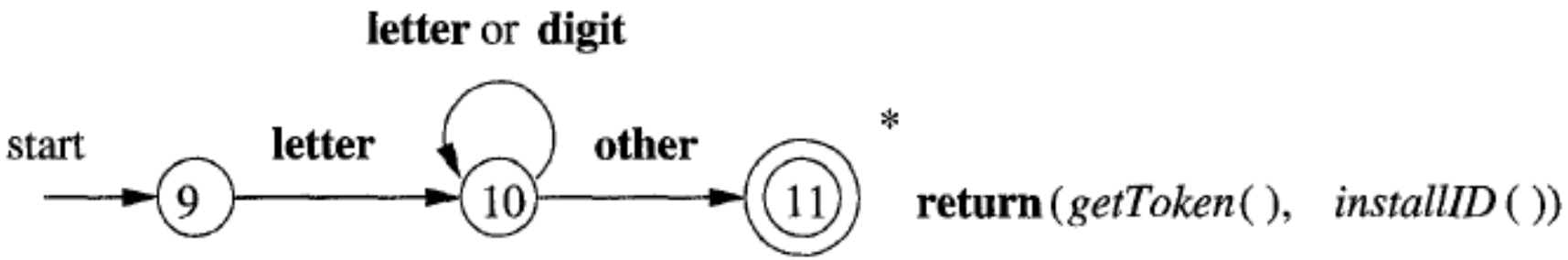


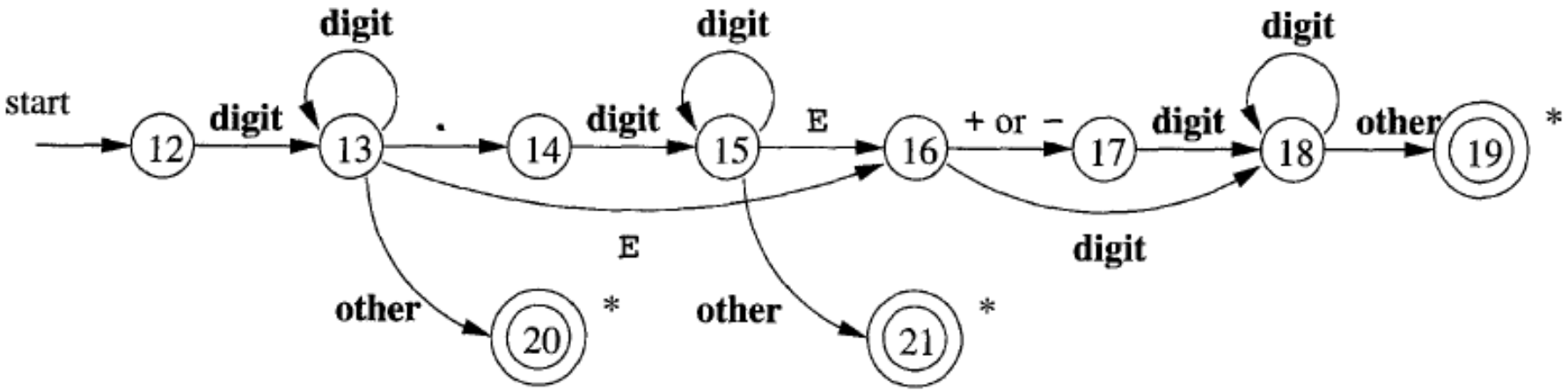
$$
\begin{aligned}
\mathit{digit} &\rightarrow [\mathtt{0\text{-}9}] \\
\mathit{digits} &\rightarrow \mathit{digit}^{+} \\
\mathit{number} &\rightarrow \mathit{digits}\ (\mathtt{.}\ \mathit{digits})?\ (\ \mathtt{E}\ [\mathtt{+\text{-}}]?\ \mathit{digits}\ )?
\end{aligned}
$$

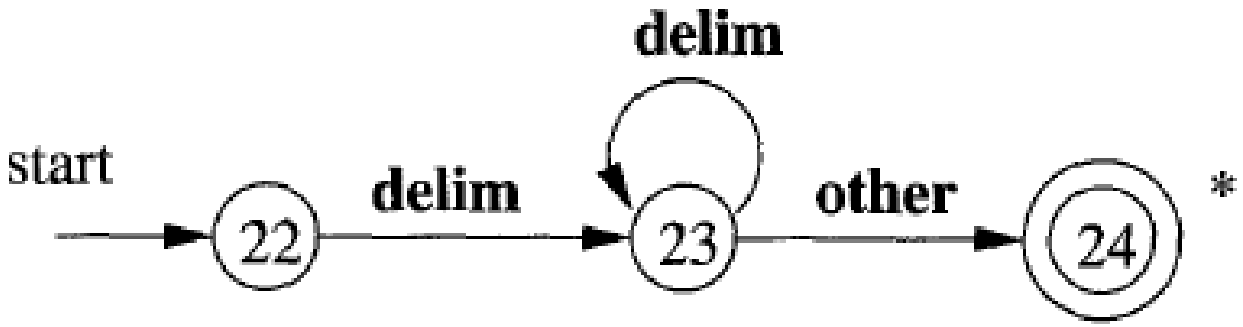
3.3 Transition Diagram



*** means that we must retract one position.**


letter or digit
start
letter
other
9
10
11
*
return (getToken( ), installID( ))


start
12
digit
13
digit
.
14
digit
15
digit
E
16
+ or −
17
digit
18
digit
other
19
*
E
digit
other
20
*
other
21
*


delim
start
delim
other
*
22
23
24

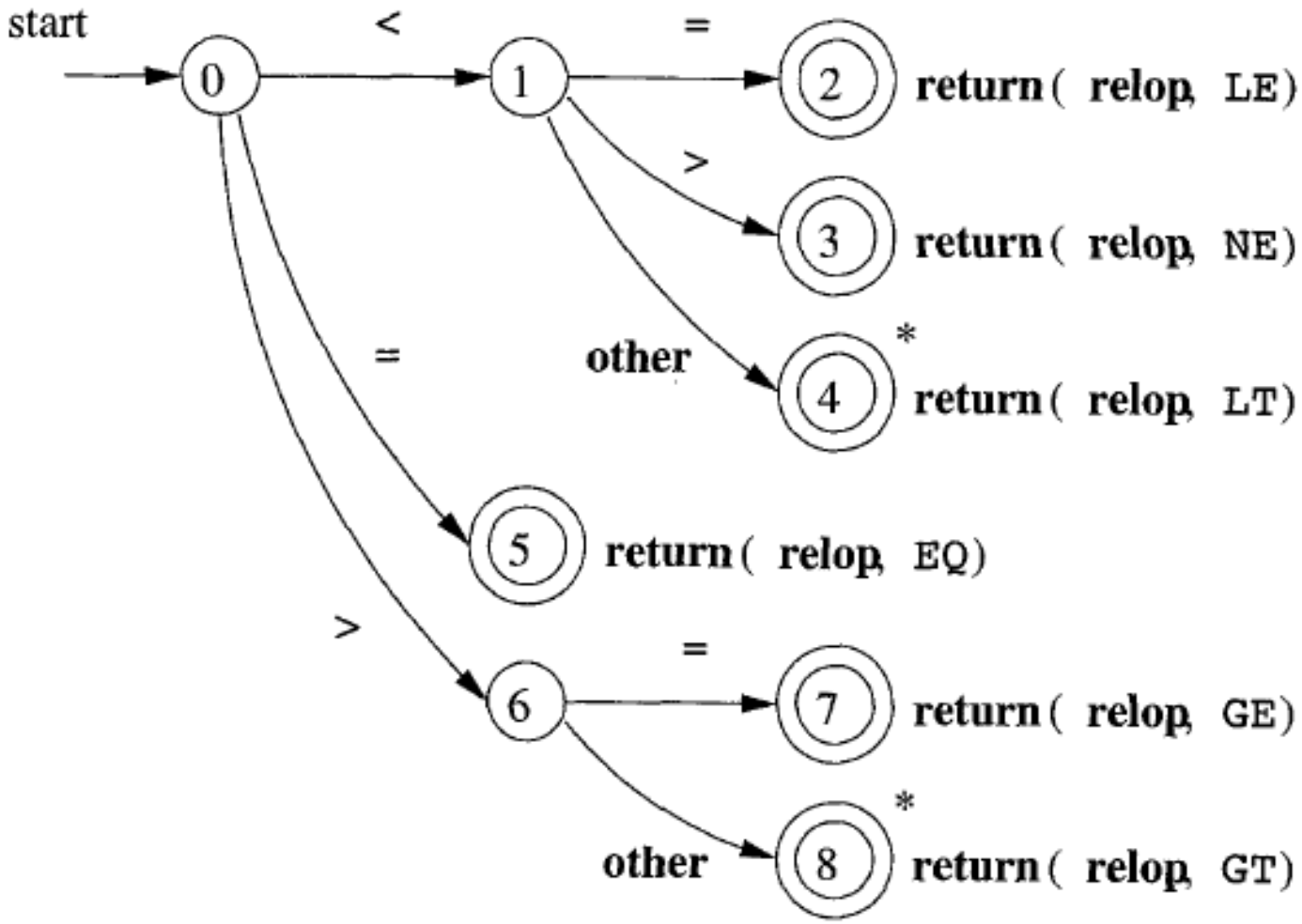
4 Tokens Recognization

TASK

Developing programs that can recognize the lexemes by the token patterns.

- ***Derivation*** *for regular grammars.*
- ***Reduction*** *for regular grammars.*
- ……

Example. RELOP Recognization



Example. RELOP Recognization

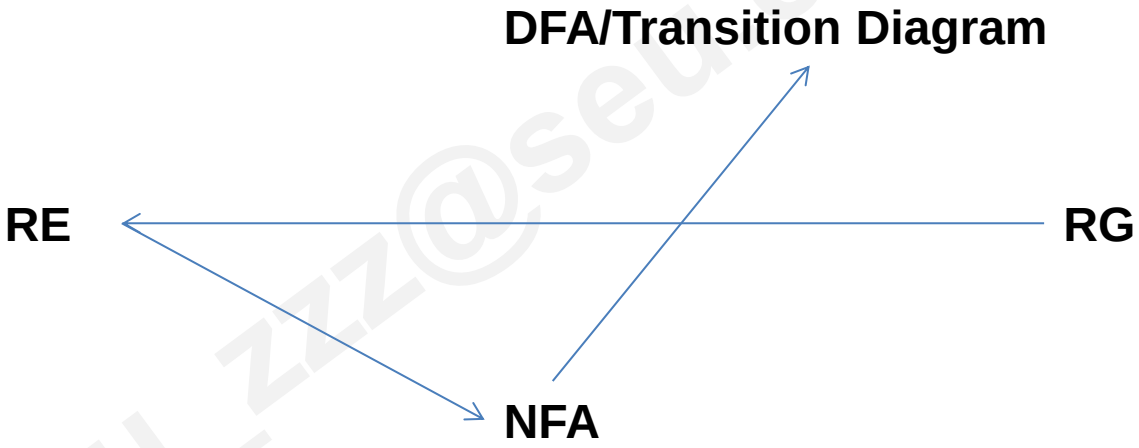
```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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

Questions

- **Can we find the corresponding transition diagram for a regular expressions?**
- **Can we construct the transition diagram for a regular grammar?**

Answer

- YES.



4.1 Finite State Automata

- Nondeterministic Finite Automata
- Deterministic Finite Automata

4.1.1 NFA

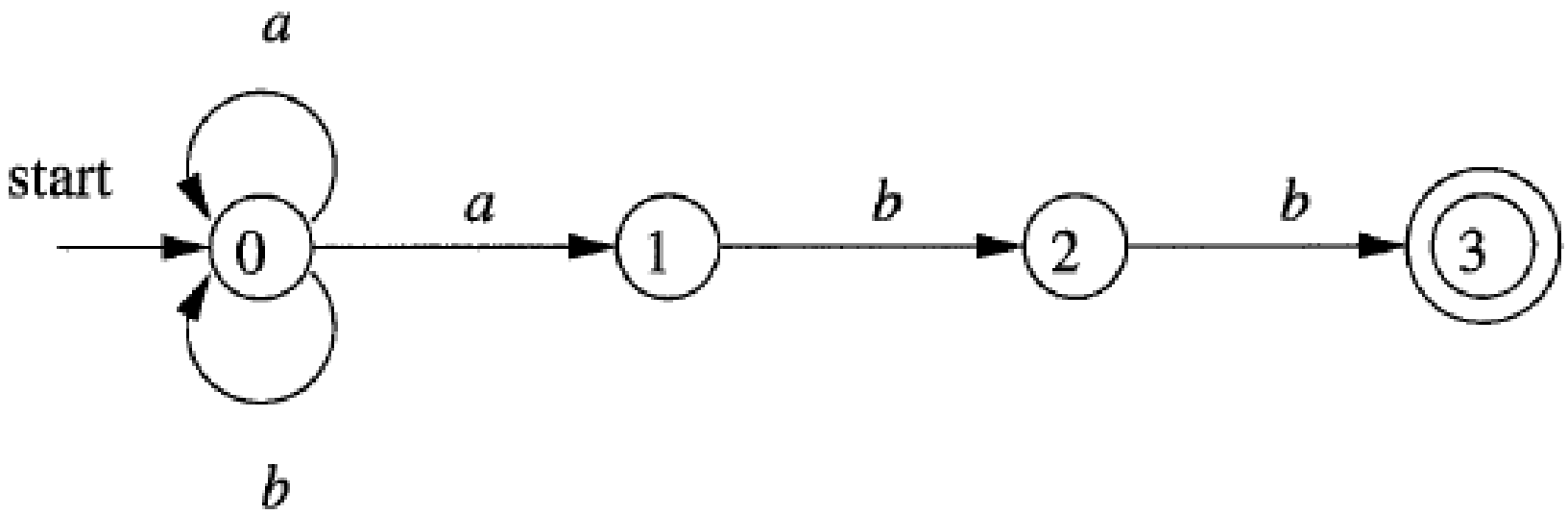
A *nondeterministic finite automaton consists of:*

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

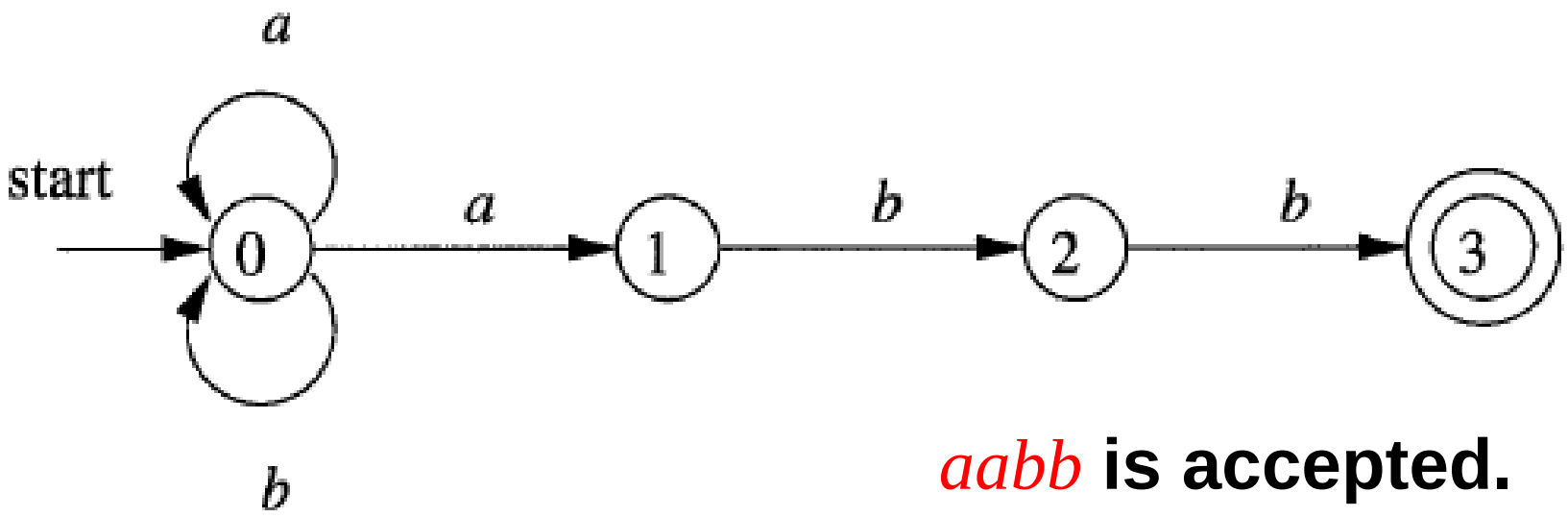
Example.

The transition graph for an NFA recognizing the language of regular expression $(a|b)^*abb$ **is shown**

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


a
start
0
a
1
b
2
b
3
b

An NFA *accepts* *input string* x *if and only if there is some path in the transition* graph from the start state to one of the accepting states, such that the symbols along the path spell out x.



aabb **is accepted.**

The *language defined (or accepted) by an NFA is the set of strings labeling* some path from the start to an accepting state.

4.1.2 DFA

A *deterministic finite automaton consists of:*

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

Algorithm : Simulating a DFA.

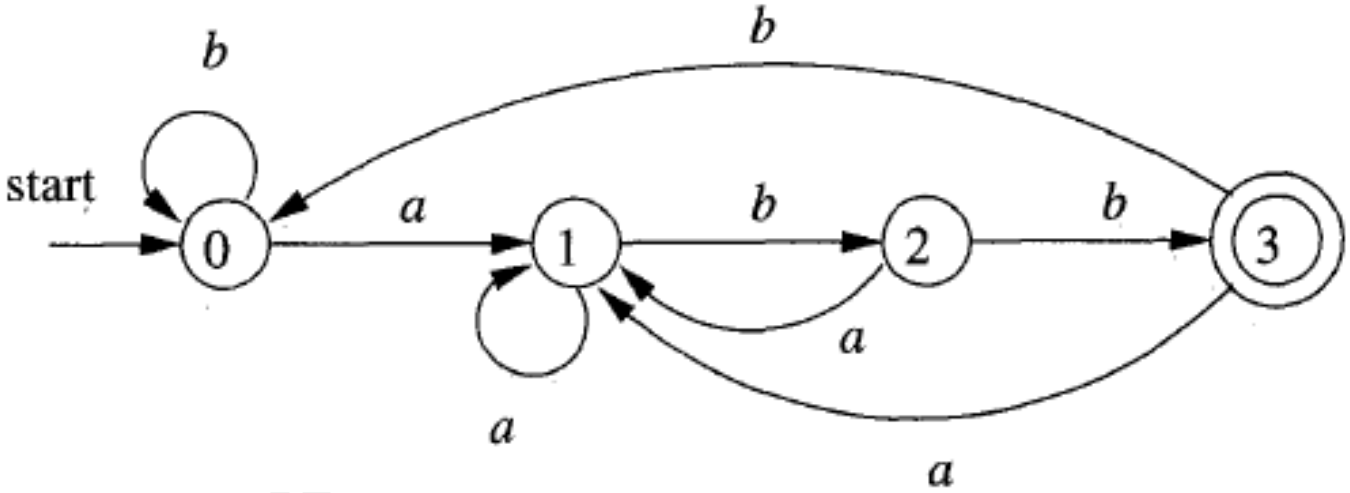
INPUT: An input string x terminated by an end-of-file character eof. A DFA D with start state so, accepting states F, and transition function $move$.

OUTPUT: Answer "yes" if D accepts x; "no" otherwise.

METHOD: Apply the algorithm to the input string x. The function $move(s, c)$ gives the state to which there is an edge from state s *on input* c. The function $nextChar$ returns the next character of the input string x.

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

Example.



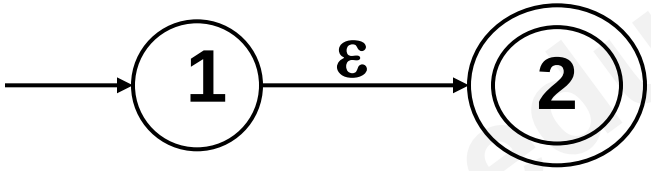
$(a|b)^*abb$ **is accepted.**

4.2 RE⇒NFA

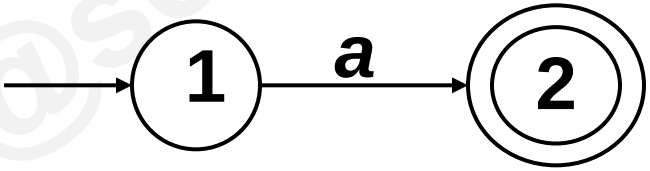
Algorithm

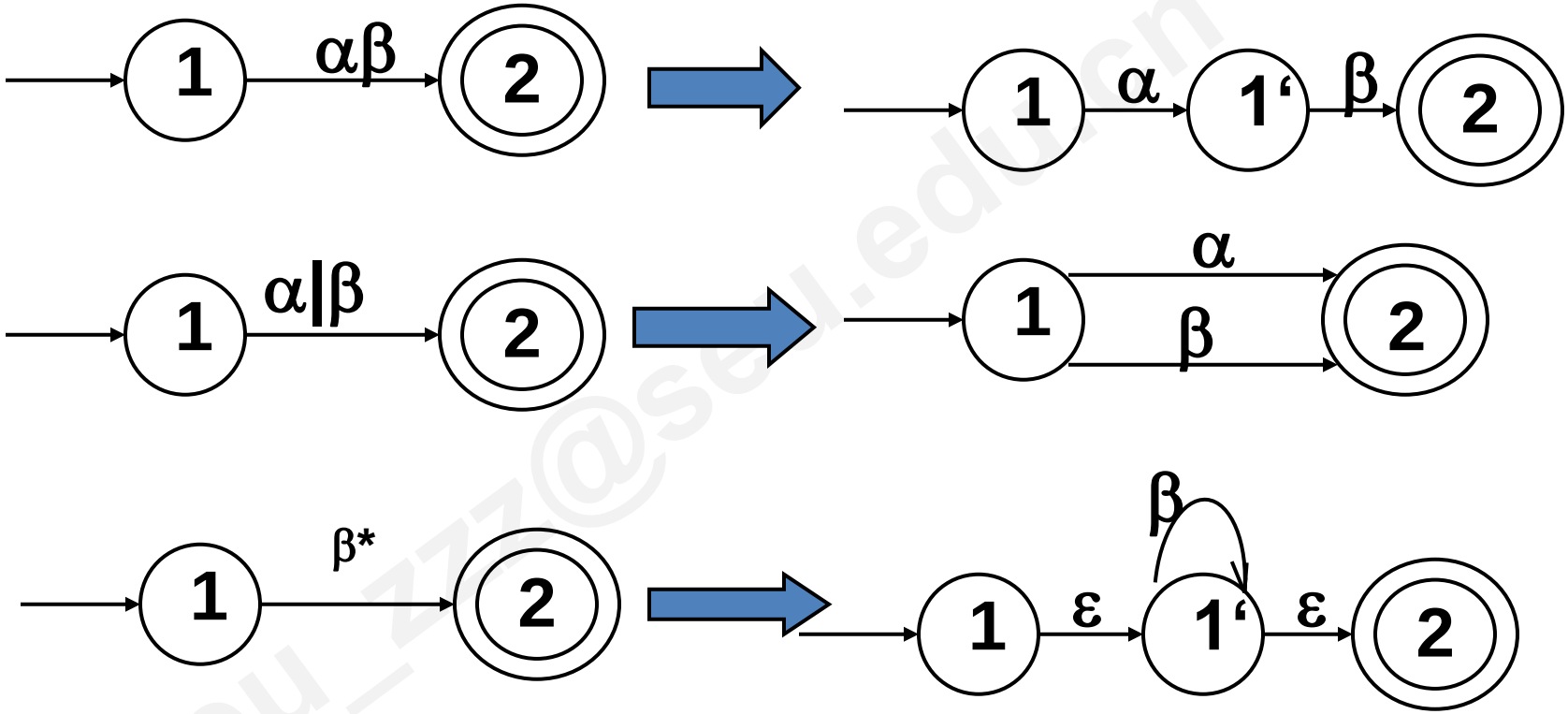
- Input. A regular expression *r* over an alphabet Σ
- Output. An NFA *N* accepting *L(r)*

1. For ε,

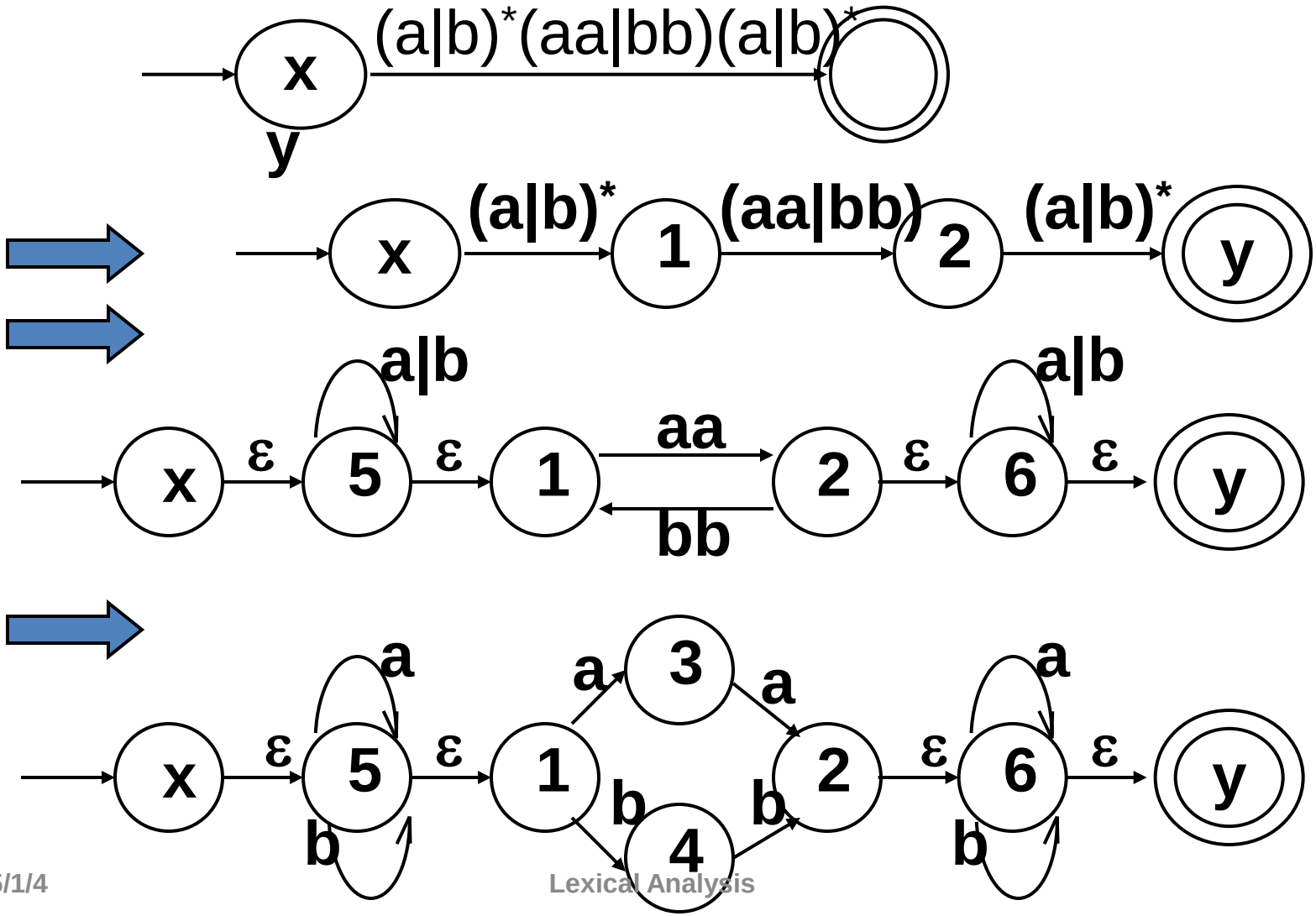


2. For *a* in Σ,





Example., Let us construct *N(r)* for the regular expression *r=(a|b)*(aa|bb)(a|b)**



4.3 NFA⇒DFA

Algorithm : The subset construction of a DFA from an NFA.

INPUT: A NFA N

OUTPUT: A DFA D accepting the same language as N.

METHOD: Our algorithm constructs a transition table $Dtran$ for D.

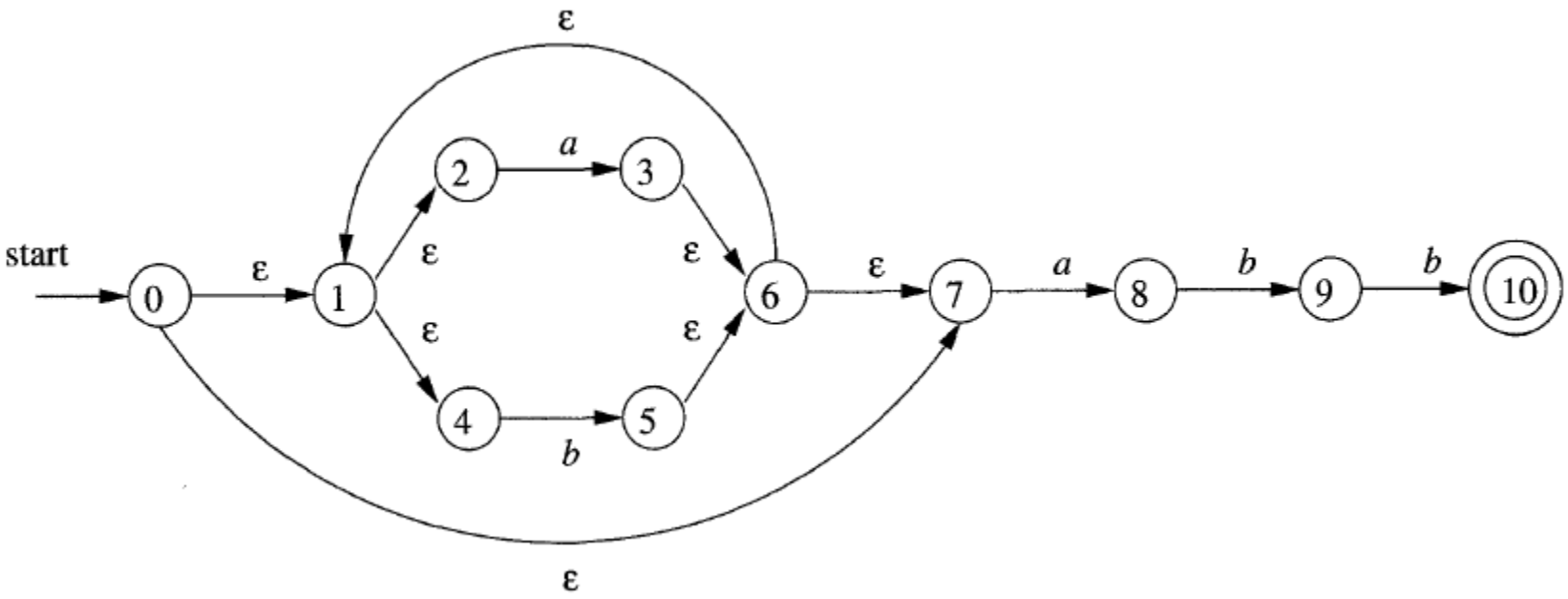
OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

initially, ϵ-*closure*(s_0) is the only state in *Dstates*, and it is unmarked;
while (there is an unmarked state T in *Dstates*) {
    mark T;
    for (each input symbol a) {
        $U = \epsilon$-*closure*$(move(T, a))$;
        if (U is not in *Dstates*)
            add U as an unmarked state to *Dstates*;
        $Dtran[T, a] = U$;
    }
}

```
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;
                }
}
```

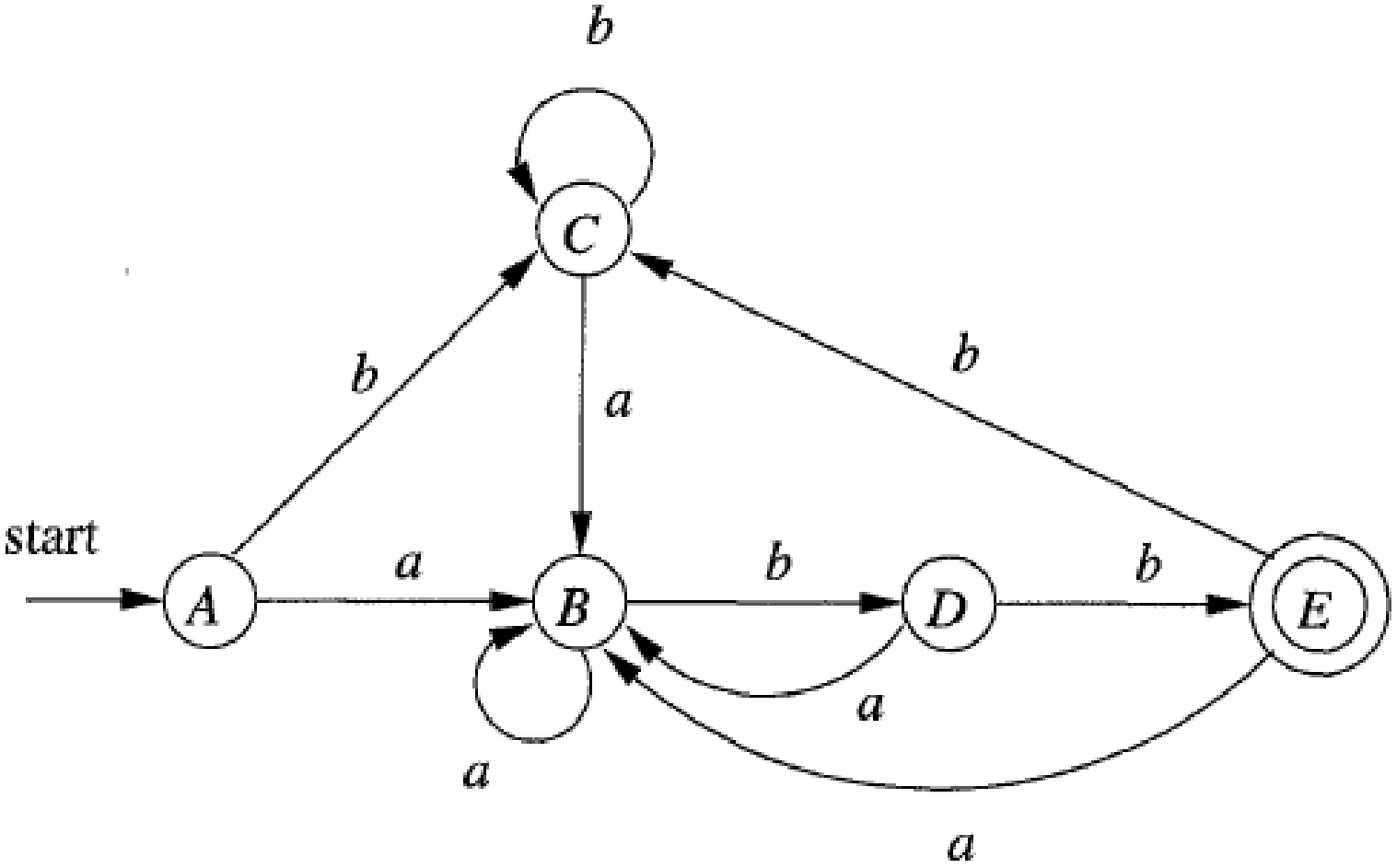
Computing the ε-closure

Example.



ε-closure(0) = {0, 1, 2, 4, 7}

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


b
C
b
a
b
start
A
a
B
b
D
b
E
a
a
a

4.3 Minimal DFA

Algorithm : Minimizing the number of states of a DFA.

INPUT: A DFA D with set of states S, input alphabet Σ, state state s_o, and set of accepting states F.

OUTPUT: A DFA D' accepting the same language as D and having as few states as possible.

METHOD:

Step1. Start with an initial partition Π with two groups, F and S - F, the accepting and non-accepting states of D.

Step2. Apply the procedure to construct a new partition Π_{new}

initially, let $\Pi_{new} = \Pi$;
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;
}

Step3. If $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π.

Step4. Choose one state in each group of Π_{final} as the *representative for that* group. The representatives will be the states of the minimum-state DFA D'. The other components of D' *are constructed as follows:*

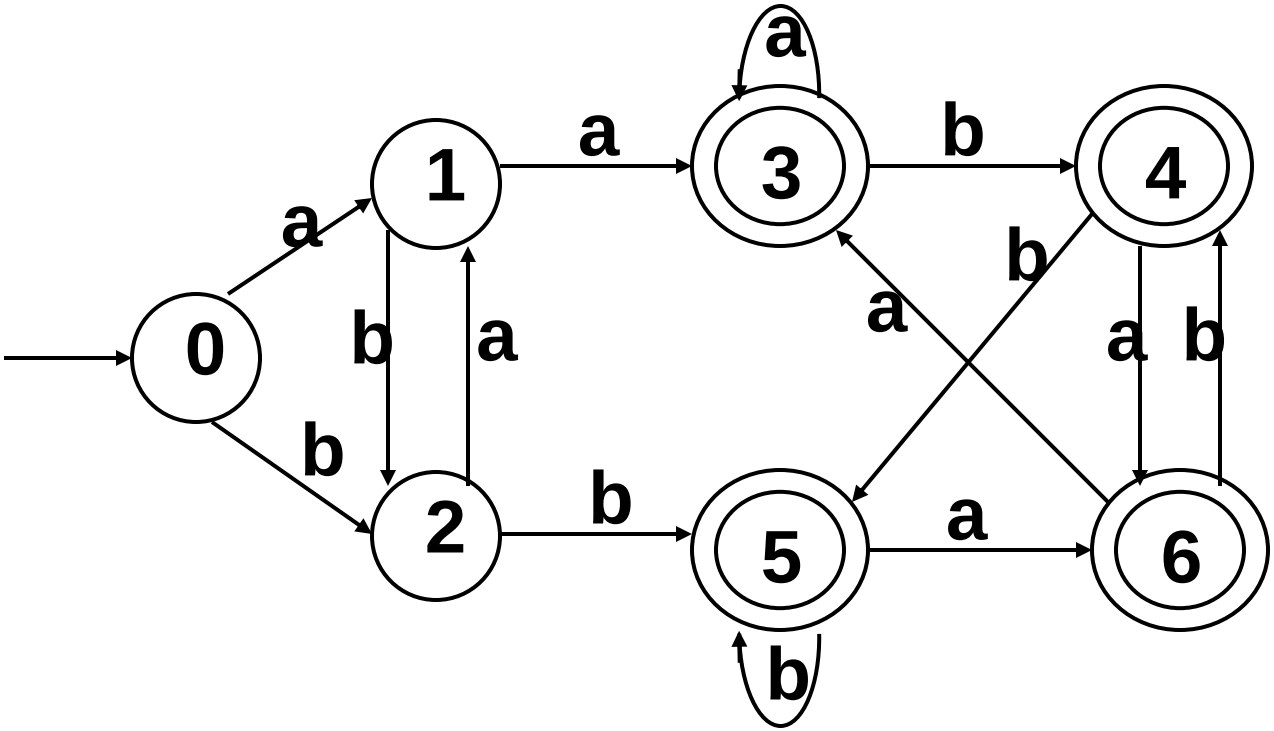
(a) The state state of D' is the representative of the group containing the start state of D.

(b) The accepting states of D' are the representatives of those groups that contain an accepting state of D.

(c) Let *s* and *t* be representative states for *s*'s and *t*'s group respectively, and suppose on input *a* there is a transition of *M* from *s* to *t*. Then *M'* has a transition from *s* to *t* on *a*.

Step5. If D' has a dead state(a state that is not accepting and that has transitions to itself on all input symbols),then remove it. Also remove any states not reachable from the start state.

Example. Minimize the following DFA.



- 1. Initialization: $\prod_0$＝{{0,1,2},{3,4,5,6}}
- 2.1 For Non-accepting states in $\prod_0$:
  - a: move({0,2},a)={1} ; move({1},a)={3} . 1,3 do not in the same subgroup of $\prod_0$.
  - So ,$\prod_1$`＝{{1}，{0,2}，{3,4,5,6}}
  - b: move({0},b)={2}; move({2},b)={5}. 2,5 do not in the same subgroup of $\prod_1$‘.
  - So, $\prod_1$``＝{{1}，{0}，{2}，{3,4,5,6}}

2.2 For accepting states in $\prod_0$:

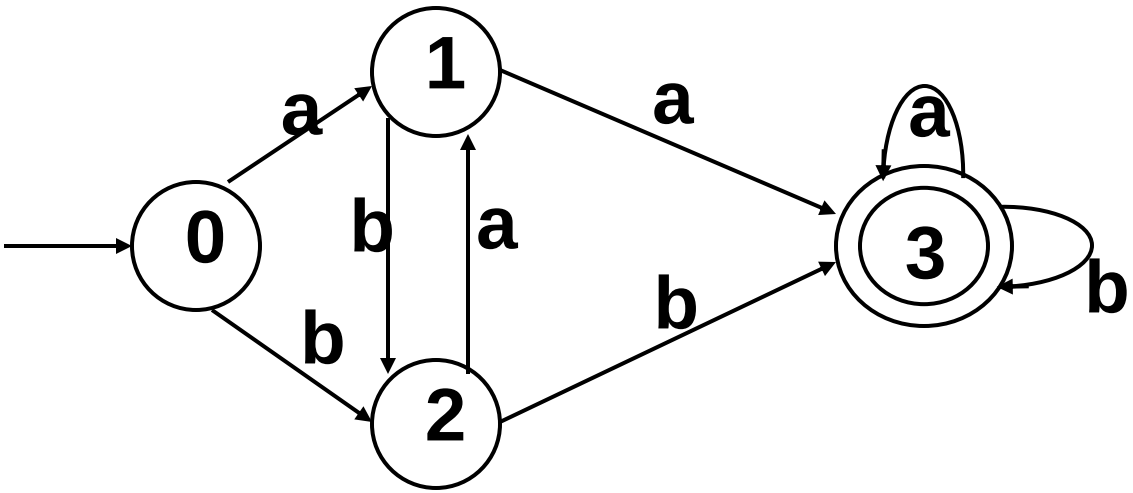
- a: move({3,4,5,6},a)={3,6}, which is the subset of {3,4,5,6} in $\prod_1$“
- b: move({3,4,5,6},b)={4,5}, which is the subset of {3,4,5,6} in $\prod_1$“
- So, $\prod_1$＝{{1}，{0}，{2}，{3,4,5,6}}.

3.Apply the step (2) again to $\prod_1$,and get $\prod_2$.

- $\prod_2$＝{{1},{0},{2},{3,4,5,6}}= $\prod_1$,
- So, $\prod_{final}$ = $\prod_1$

4. Let state 3 represent the state group {3,4,5,6}

So, the minimized DFA is :



Why the State-Minimization Algorithm Works?

P_{182}

5. LEX

SELF-Study