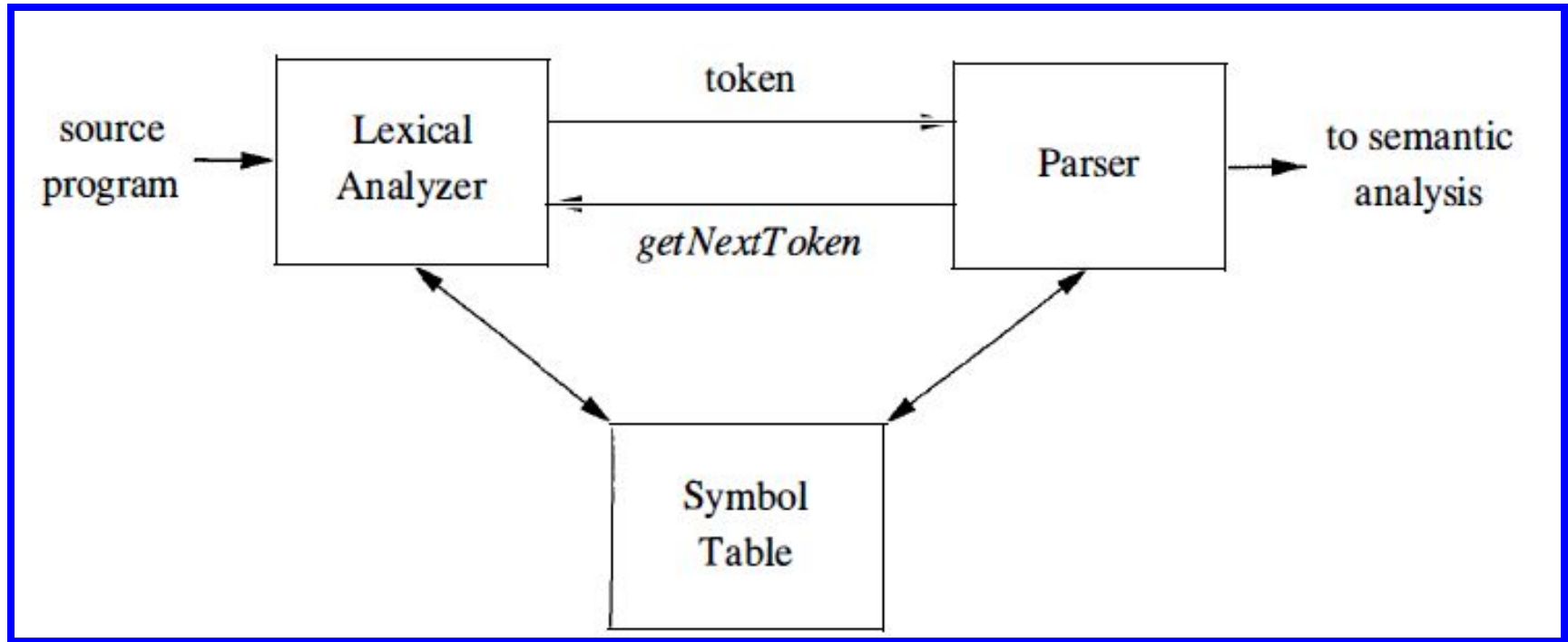


Lexical Analysis

The Role of the Lexical Analyzer

- The lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

Interaction between the lexical analyzer & the parser



- **Other Tasks:**

- Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Correlating error messages generated by the compiler with the source program.

Divided into two processes

1. **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
2. **Lexical analysis** proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis Versus 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.
 - The separation of lexical & syntactic analysis often allows us to simplify at least one of these tasks.
 - If we are designing a new language, separating lexical & syntactic concerns can lead to a cleaner overall language design.

2. Compiler efficiency is improved.

- A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
- In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced.

Input-device-specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns & Lexemes

- **Token**

- ❖ a token name + an optional attribute value.
- ❖ The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.
- ❖ The token names are the input symbols that the parser processes.

- **Pattern**

- A pattern is a description of the form that the lexemes of a token may take.
- In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- For identifiers & some other tokens, the pattern is a more complex structure that is matched by many strings.

- **Lexemes**

- A lexeme is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyzer as an instance of that token.

Example: Patterns & Lexemes

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"

- C statement

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

- both **printf** and **score** are lexemes matching the pattern for token **id**, and " **Total = %d\n**" is a lexeme matching **literal**.

Covering most or all of the tokens

1. **One token for each keyword.** The pattern for a keyword is the same as the keyword itself.
2. **Tokens for the operators,** either individually or in classes.
3. **One token representing all identifiers.**
4. **One or more tokens representing constants,** such as numbers & literal strings.
5. **Tokens for each punctuation symbol,** such as left & right parentheses, comma, & semicolon.

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases **additional information about the particular lexeme that matched.**
- For example, the pattern for token **number** matches both **0** and **1**, but it is extremely important for the code generator to know which lexeme was found in the source program.
- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token ;
- **Token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.**

Attributes for Tokens

- Tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information.
- Normally, information about an identifier-e.g., its lexeme, its type, and the location at which it is first found 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.

Example 3.2 : The token names & associated attribute values for the Fortran statement

$$E = M * C ** 2$$

- **Sequence of pairs:**

- **<id**, pointer to symbol-table entry for **E**>
- **<assign_op>** [no need to assign]
- **<id**, pointer to symbol-table entry for **M**>
- **<mult_op>** [no need to assign]
- **<id**, pointer to symbol-table entry for **C**>
- **<exp_op>** [no need to assign]
- **<number**, integer value **2**>

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string **fi** is encountered for the first time in a C program in the context :

fi (a == f (x)) . . .

- A lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function identifier.
- Since **fi** is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser

Lexical Errors

- Let the parser - handle an error due to transposition of the letters.
- However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- **Error Recovery**
- ✓ The simplest recovery strategy is "panic mode" recovery.
- ✓ We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.
- ✓ This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

- **Other Recovery**

- 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.

Terms for Parts of Strings

1. A **prefix** of string **s** is any string obtained by removing zero or more symbols from the end of **s**.
 - **ban, banana, and ϵ are prefixes of banana.**
2. A **suffix** of string **s** is any string obtained by removing zero or more symbols from the beginning of **s**.

nana, anana, and ϵ are suffixes of banana.
3. A **substring** of **s** is obtained by deleting any prefix and any suffix from **s**.

banana, nan, and ϵ are substrings of banana.
4. The **proper** prefixes, suffixes, and substrings of a string **s** are those, prefixes, suffixes, and substrings, respectively, of **s** that are not ϵ or not equal to **s** itself.
5. A **subsequence** of **s** is any string formed by deleting zero or more not necessarily consecutive positions of **s**.

baan is a subsequence of **banana**.

Regular Expressions

- **Regular expressions:** underscore is included among the letters.
- if *letter_* is established to stand for any letter or the underscore, and *digit_* established to stand for any digit, then we could describe the language of C identifiers by :

*Letter_ (letter_ | digit)**

- **Vertical bar:** union
- **Parentheses:** group sub expressions,
- **Star:** zero or more occurrences of
- **Juxtaposition** of *letter_* with the remainder of the expression signifies concatenation

- Each regular expression **r** denotes a language **L(r)** , which is also defined recursively from the languages denoted by r's **sub expressions**.

- Rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.

BASIS: There are two rules

- 1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$
 - the language whose sole member is the empty string.
- 2. If a is a symbol in Σ , then a is a regular expression, and $L(a) = \{a\}$,
 - the language with one string, of length one, with a in its one position.

INDUCTION: There are 04 parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$.

1. $(r)/(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
3. $(r)^*$ is a regular expression denoting $(L(r))^*$.
4. (r) is a regular expression denoting $L(r)$.
 - This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

- We may drop certain pairs of parentheses
- **Conventions**
 - a) The unary operator $*$ has highest precedence & is left associative.
 - b) Concatenation has second highest precedence and is left associative.
 - c) $|$ has lowest precedence and is left associative.
- $(a) | ((b) *(c)) == alb* c.$
- Both expressions denote the set of strings that are either a, *single a* or are *zero* or *more b's* followed by *one c*.

Example 3.4 : Let $\Sigma = \{a, b\}$.

1. The regular expression $a|b$ denotes the language $\{a, b\}$.
2. $(ab)(ab)$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ . Another regular expression for the same language : $aa|ab|ba|bb$
3. a^* denotes the language consisting of all strings of zero or more a's: $\{\epsilon, a, aa, aaa, \dots\}$.
4. $(a|b)^*$ denotes the set of all strings consisting of zero or more instances of a or b,
 - all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$.
 - Another regular expression for same language: $(a^*b^*)^*$.
5. ala^*b denotes the language $\{a, b, ab, aab, aaab, \dots\}$,
 - the string a & all strings consisting of zero or more a's & ending in b .

- **Regular set**: A language that can be defined by a regular expression
- If two regular expressions r and s denote the same regular set, we say they are equivalent and write $r = s$. For instance, $(alb) = (b la)$.

Algebraic laws for regular expressions r , s , & t

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

Regular Definition

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

- $d_1 \rightarrow r_1$

- $d_2 \rightarrow r_2$

- $D_n \rightarrow r_n$

- 1. Each d_i is a new symbol, not in Σ and not the same as any other of the d 's,
- 2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$
- By restricting r_i to Σ & previously defined d 's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i
- How: first replacing uses of d_1 in r_2 (which cannot use any of the d 's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2 , and so on.
- Finally, in r_n replace each d_i , for $i = 1, 2, \dots, n - 1$, by the substituted version of r_i , each of which has only symbols of Σ .

- **Example:** C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.
- **letter $\rightarrow A | B | \dots | z | a | b | \dots | z | -$**
- **digit $\rightarrow 0 | 1 | \dots | 9$**
- **id $\rightarrow \text{letter} _ (\text{letter} - | \text{digit}) ^ *$**

- Example : Unsigned numbers (integer or floating point) are strings such as 5280, 0 . 0 1234, 6 . 336E4, or 1 . 89E-4.
- **digit** \rightarrow 0 | 1 | \dots | 9
- **digits** \rightarrow digit digit*
- **optionalFraction** \rightarrow . digits | ϵ
- **optionalExponent** \rightarrow (E (+ | - | ϵ) digits) | ϵ
- **number** \rightarrow digits optionalFraction optionalExponent

Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Typical examples:

Abbr.	Meaning	Notes
r^+	(rr^*)	1 or more occurrences
$r?$	$(r \mid \epsilon)$	0 or 1 occurrence
$[a-z]$	$(a b \dots z)$	1 character in given range
$[abxyz]$	$(a b x y z)$	1 of the given characters

Examples

<i>re</i>	Meaning
+	single + character
!	single ! character
=	single = character
!=	2 character sequence
<=	2 character sequence
xyzzzy	5 character sequence

Extensions of Regular Expressions

- 1. **One or more instances**. The unary, postfix operator **+** represents the positive closure of a regular expression and its language.
 - That is, if **r** is a regular expression, then **(r)⁺** denotes the language **(L(r))⁺**.
 - The operator **+** has the same precedence and associativity as the operator *****
 - Two useful algebraic laws, **r^{*} = r⁺ | e** and **r⁺ = rr^{*} = r*r** relate the Kleene closure & positive closure.
 - 2. **Zero or one instance**. The unary postfix operator **?** means "zero or one occurrence." That is, **r?** is equivalent to **r | ε**, or put another way, **L(r?) = L(r) ∪ {ε}**.
 - The **?** operator has the same precedence and associativity as ***** and **+**.
 - 3. **Character classes**. A regular expression **a₁ | a₂ | ... | a_n**, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand **[a₁a₂...a_n]**.
 - when **a₁, a₂, ..., a_n** form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by **a₁-a_n**, that is, just the first and last separated by a hyphen.
- **[abc] == a | b | c, [a-z] == a | b | ... | z**

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{array}{lll} \textit{letter_} & \rightarrow & [\textit{A-Za-z_}] \\ \textit{digit} & \rightarrow & [0-9] \\ \textit{id} & \rightarrow & \textit{letter_} (\textit{letter} \mid \textit{digit})^* \end{array}$$

The regular definition of Example 3.6 can also be simplified:

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

Recognition of Tokens

- Build a piece of code that examines the input string & finds a prefix that is a lexeme matching one of the patterns.

<i>stmt</i>	→	if <i>expr</i> then <i>stmt</i>
		if <i>expr</i> then <i>stmt</i> else <i>stmt</i>
		ϵ
<i>expr</i>	→	<i>term</i> relop <i>term</i>
		<i>term</i>
<i>term</i>	→	id
		number

Example:

Terminals:

if, then, else,
relop , id,
number---names
of tokens

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

<i>stmt</i>	→	<code>if <i>expr</i> then <i>stmt</i></code>
		<code>if <i>expr</i> then <i>stmt</i> else <i>stmt</i></code>
		<code>ε</code>
<i>expr</i>	→	<code><i>term</i> relop <i>term</i></code>
		<code><i>term</i></code>
<i>term</i>	→	<code>id</code>
		<code>number</code>

ws → (**blank** | **tab** | **newline**)⁺

blank, tab, newline are abstract symbols.

Token **ws** is different from the other tokens in that , when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.

Tokens, their patterns, and attribute values

- For each lexeme or family of lexemes, which **token name** is returned to the parser and what **attribute value**, is returned.

□ 06 relational operators are used as the **attribute value**, in order to indicate which instance of the token **relop** we have found

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

Transition Diagrams

- **convert patterns into stylized flowcharts:**
"transition diagrams"
 - Transition diagrams have a collection of **nodes or circles, called states.**
 - Each state represents **a condition** that could occur during the process of scanning the input looking **for a lexeme that matches one of several patterns.**

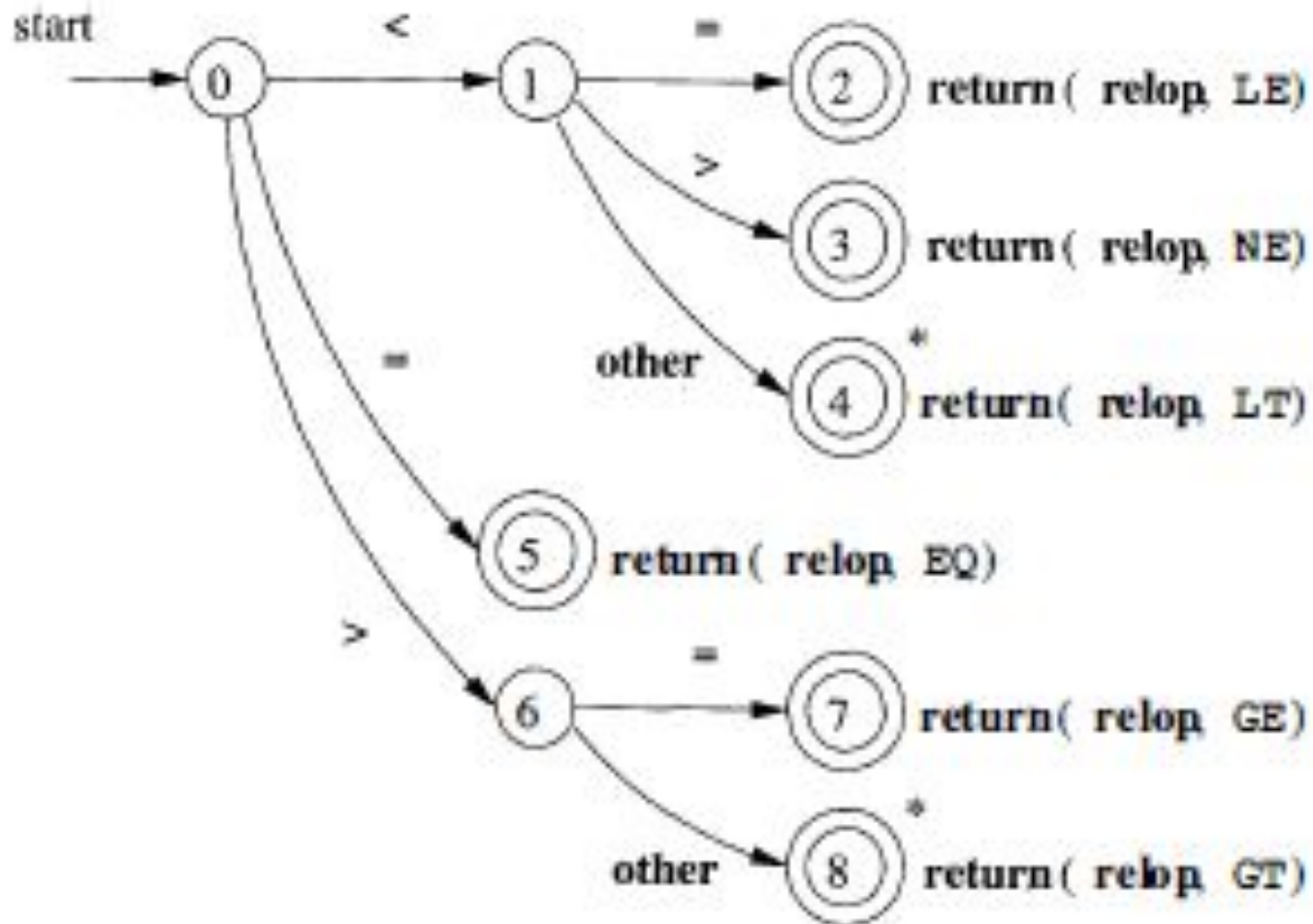
Transition Diagrams

- **Edges** are directed from **one state to another**.
- Each edge is **labeled** by a **symbol or set of symbols**.
- If we are in some **state s** , & the next input symbol is **a**, we look for an edge **out of state s labeled by a** (and perhaps by other symbols, as well).
- If such an edge found, advance the forward pointer & enter the state of the transition diagram to which that edge leads.
- **transition diagrams are deterministic**
 - there is **never more than one edge out of a given state with a given symbol** among its labels.

Some important conventions

1. Certain states are said to be **accepting, or final**. These states indicate that a lexeme has been found. We always indicate an **accepting state by a double circle**, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place **a*** near that accepting state.
3. One state is designated the **start state**, or **initial state**; it is indicated by an **edge, labeled "start,"** entering from nowhere. **The transition diagram always begins in the start state before any input symbols have been read.**

- **Example:** Transition diagram that recognizes the lexemes matching the token **relop**.



Recognition of Reserved Words and Identifiers

- Keywords (**if** or **then**) are reserved
- not identifiers
- Transition diagram for identifier lexemes, & recognize the keywords **if** , **then**, & **else**



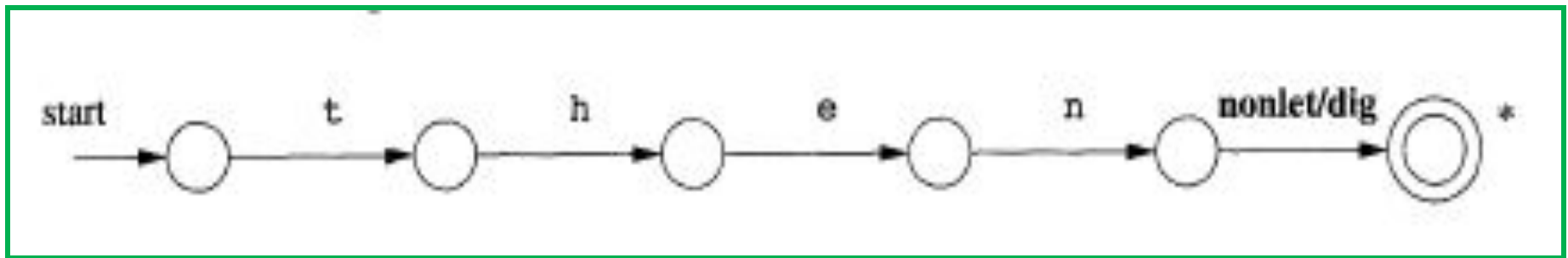
02 ways that we can handle reserve words that look like identifier

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.

When we find an identifier, a call to `installID` places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. The function `gettoken()` examines the lexeme and returns the token name, either `id` or a name corresponding to a reserved keyword.

2. Create separate transition diagrams for each keyword;

Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a *"nonletter-or-digit,"* i.e., any character that cannot be the continuation of an identifier.



Hypothetical transition diagram for the keyword then

$number \rightarrow digits (. digits)? (E [+ -]? digits)?$

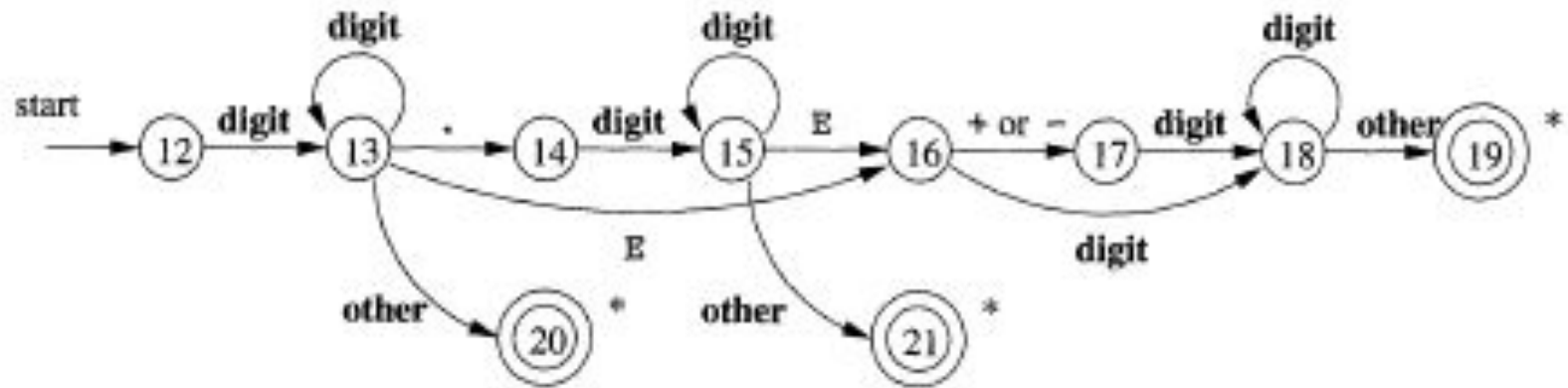


Figure : A transition diagram for unsigned numbers

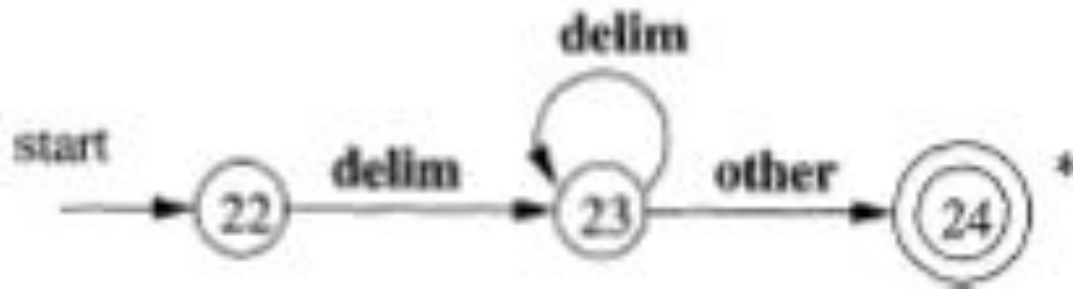


Figure : A transition diagram for whitespace

$ws \rightarrow (blank | tab | newline)^+$

Finite Automata

- Finite automata are **essentially graphs, like transition diagrams**, with a few differences:
 1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
 2. Finite automata come in two flavors:
 - (a) **Nondeterministic finite automata (NFA)** have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ , the empty string, is a possible label.
 - (b) **Deterministic finite automata (DFA)** have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.
- Both NFA & DFA are capable of recognizing the same languages (regular language)

Nondeterministic Finite Automata (NFA)

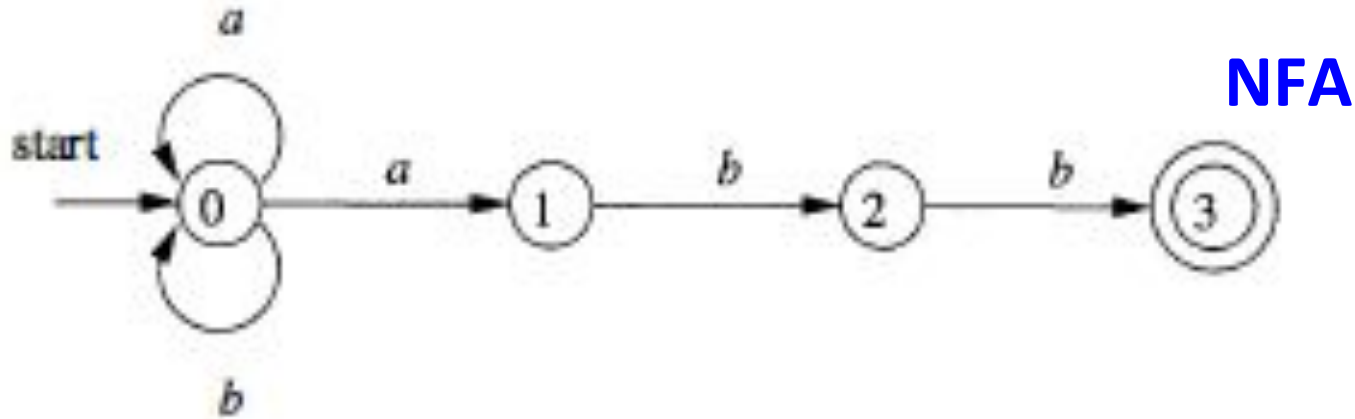
1. A finite set of states S .
2. A set of input symbols Σ , the *input alphabet*. The *empty string* (ϵ), is never a member of Σ
3. A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as 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)*.

- Any NFA/DFA can be represented by a transition graph, where the nodes are states and the labeled edges represent the transition function.
- There is an edge labeled **a** from state **s** to state **t**
 - ❖ if and only if **t** is one of the next states for state **s** and input **a**

This graph is very much like a transition diagram, except

- a) The same symbol can label edges from one state to several different states,
- b) An edge may be labeled by ϵ , the empty string, instead of, or in addition to, symbols from the input alphabet.

- Example: $R = (alb)^* abb$



- Double circle around state 3 indicates that this state is accepting.
- Only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading **abb** from the input.
- Thus, the only strings getting to the accepting state are those that end in **abb**.

Transition Tables

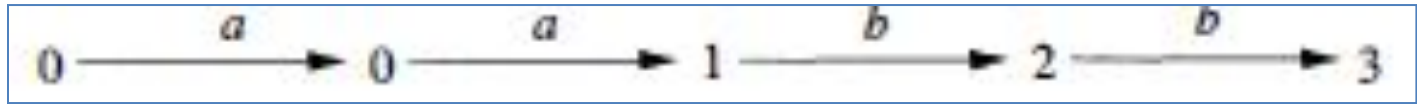
- **Rows** : states
- **Columns**: input symbols and ϵ .
- The entry for a given state & input is **value of the transition function** applied to those arguments.
- If the transition function has no information about that state-input pair, put Φ .
- **Adv**: Easily find the transitions on a given state and input.
- **Disadv**: takes a lot of space, when the input alphabet is large,

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

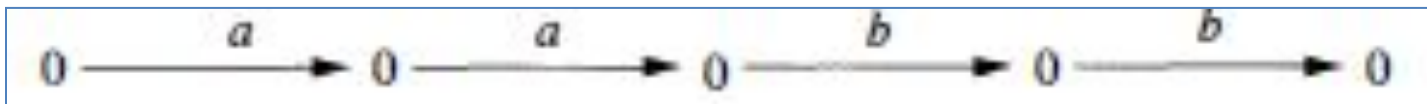
Acceptance of Input Strings by Automata

- An NFA accepts input string **x** if & only if there is some path in the transition graph from the **start state to one of the accepting states**
- ϵ labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

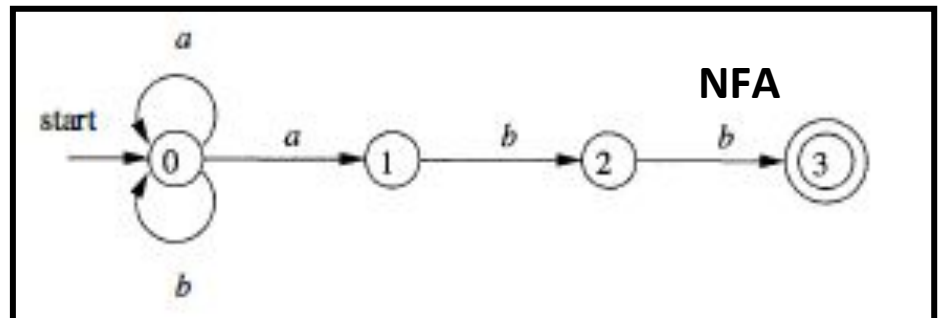
Example : The string **aabb** is accepted by the NFA



- Another path (not accepting)

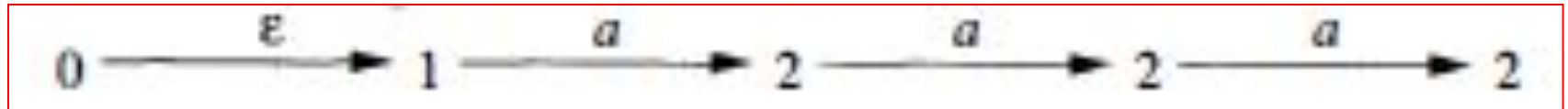
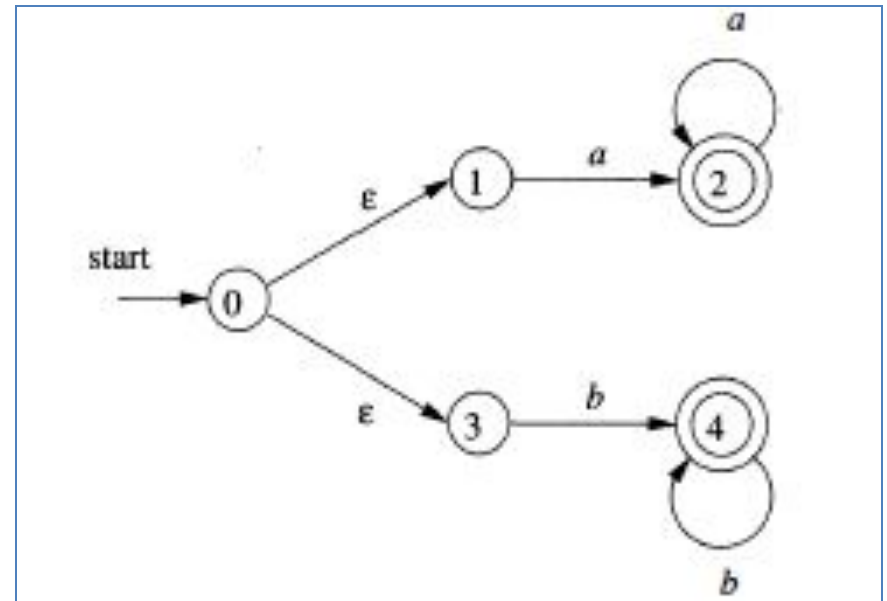


- ❑ NFA accepts a string as long as some path labeled by that string leads from the start state to an accepting state.



$L(aa^* \mid bb^*)$

- String **aaa** accepted
- ϵ is "disappear" in a concatenation

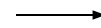
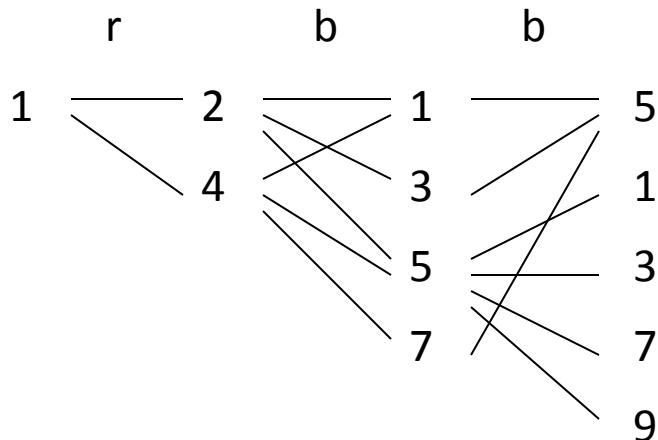


Example: Moves on a Chessboard

- States = squares.
- Inputs = r (move to an adjacent red square) and b (move to an adjacent black square).
- Start state, final state are in opposite corners.

Example: Chessboard – (2)

1	2	3
4	5	6
7	8	9



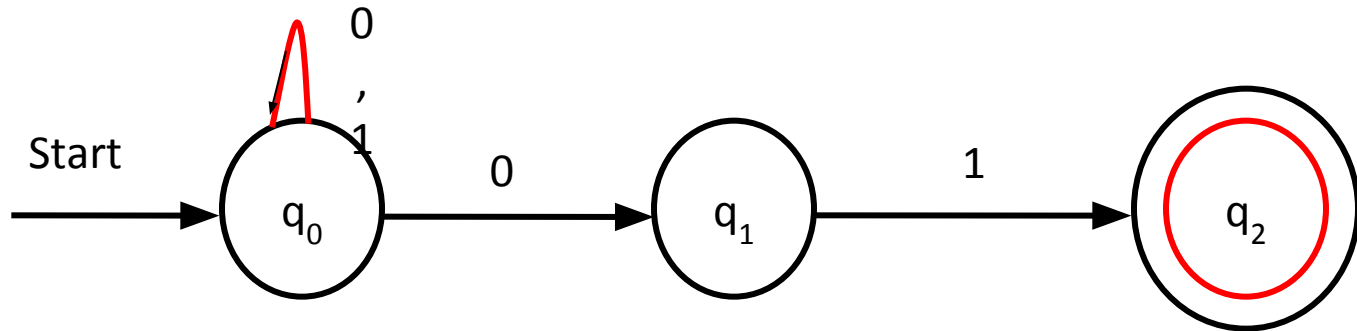
	r	b
1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
9	6,8	5

*

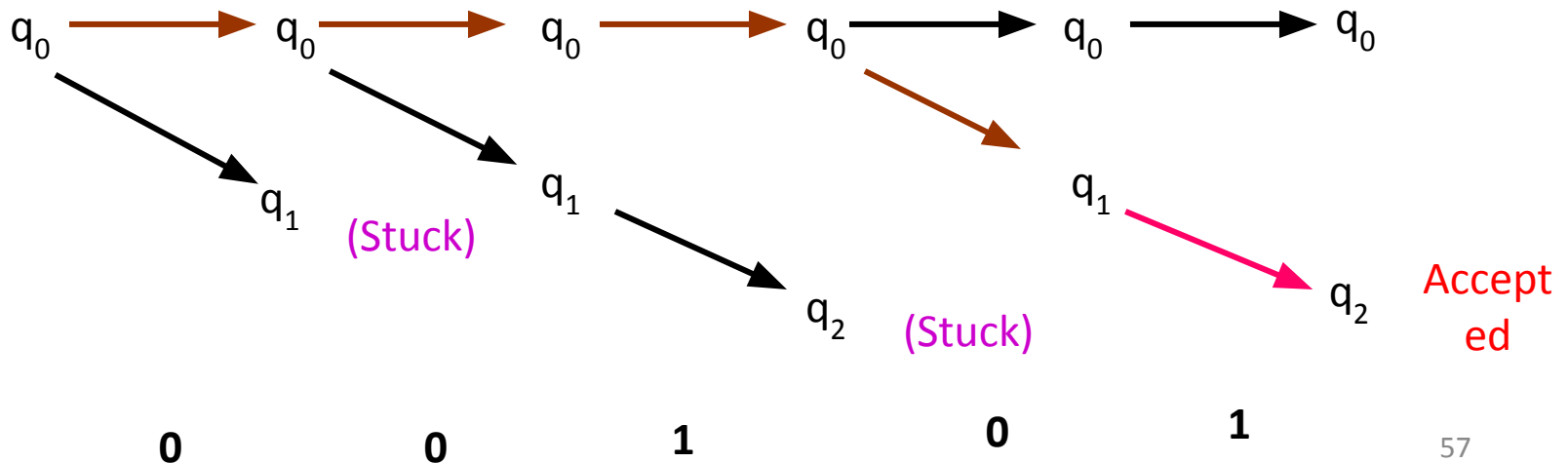
← Accept, since final state reached

Example

- An NFA accepting all strings that end in 01

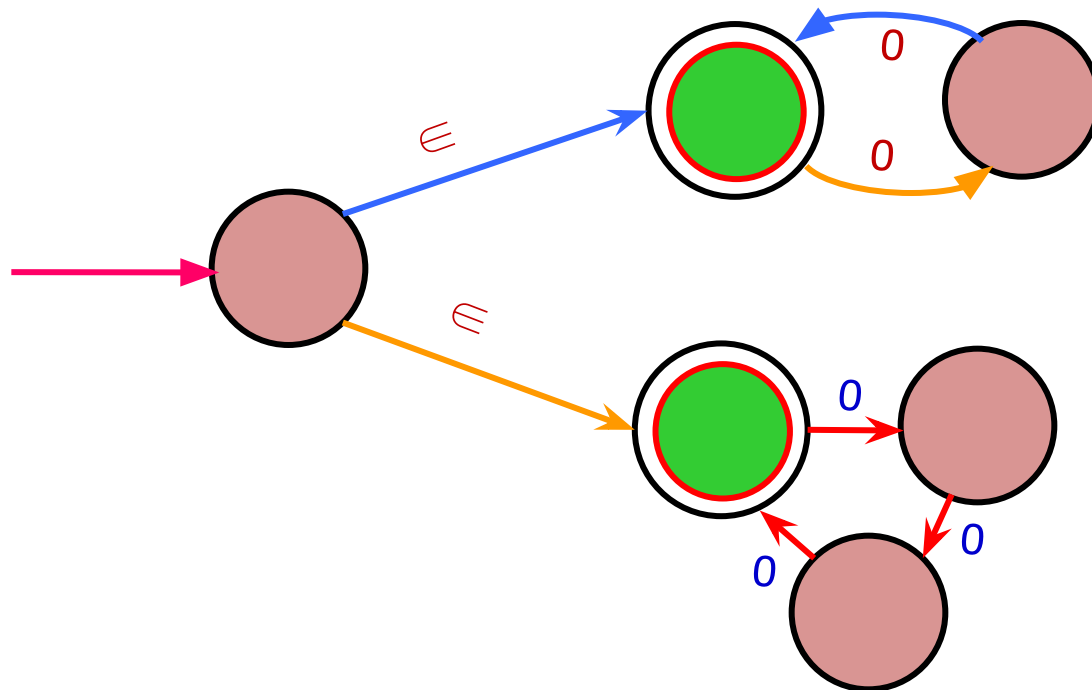


Input: **00101**



Example

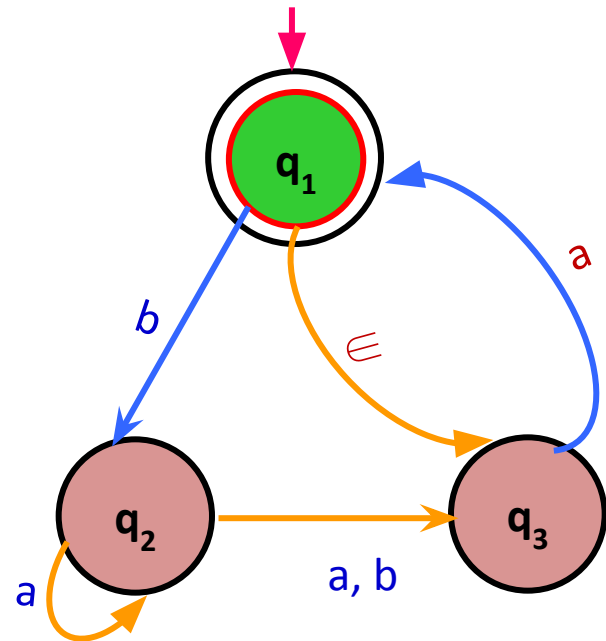
- NFA that has an input alphabet $\{0\}$ consisting of a single symbol. It accepts all strings of the form 0^k where k is a multiple of 2 or 3 (accept: ϵ , 00, 0000, 000000 but not 0, 00000)



Example

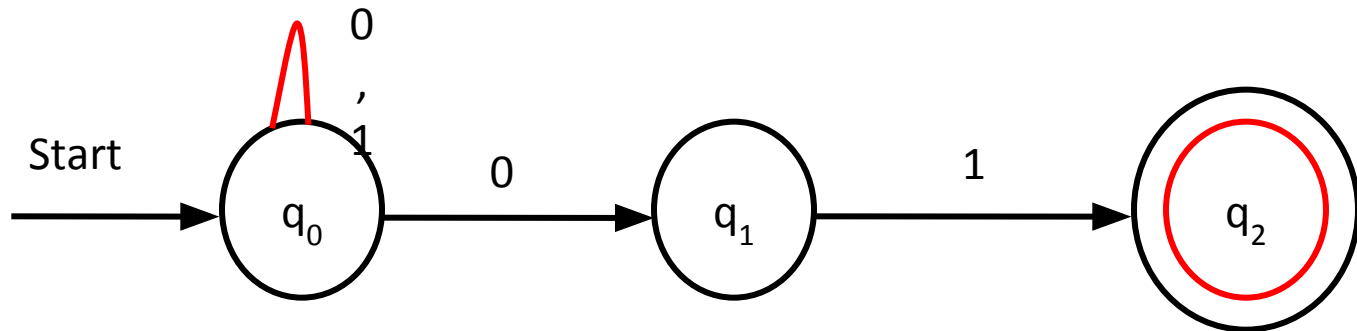
Accept: \in , a, baba, baa

Reject: b, bb, babba



Transition Table

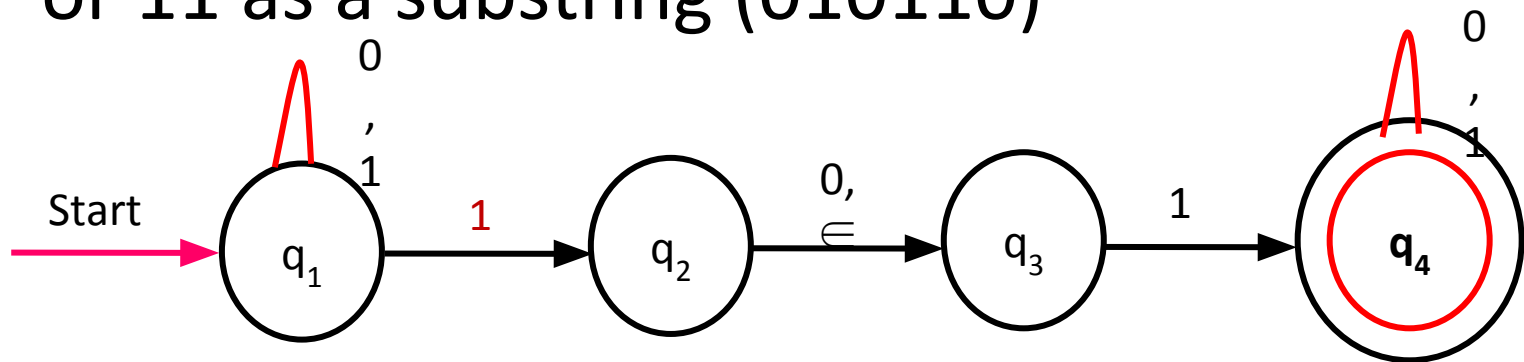
NFA $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Transition Table

- Accept all strings that contains either 101 or 11 as a substring (010110)



1. $Q = \{q_1, q_2, q_3, q_4\}$

2. $\Sigma = \{0, 1\}$

3. δ

4. Start state: q_1

5. $F = \{q_4\}$

	0	1	ϵ
$\rightarrow q_1$	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
$*q_4$	$\{q_4\}$	$\{q_4\}$	\emptyset

Deterministic Finite Automata (DFA)

1. There are no moves on input ϵ
2. For each state **s** & input symbol **a**, there is exactly one edge out of **s** labeled **a**
 - If we are using a transition table to represent a DFA, then each entry is a single state.
 - Represent this state without the curly braces that we use to form sets.
 - Lexical Analyzer---DFA

Algorithm: Simulating a DFA.

- **INPUT:** An input string **x** terminated by an end-of-file character eof. A DFA D with **start state** s_0 , **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.

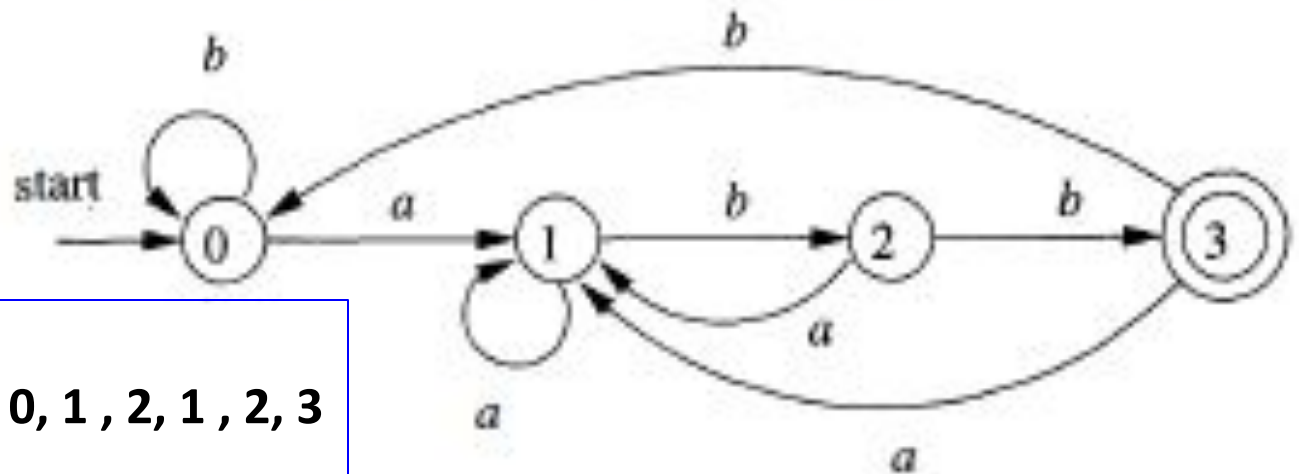
```

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

```

Figure 3.27: Simulating a DFA

(a | b)* abb

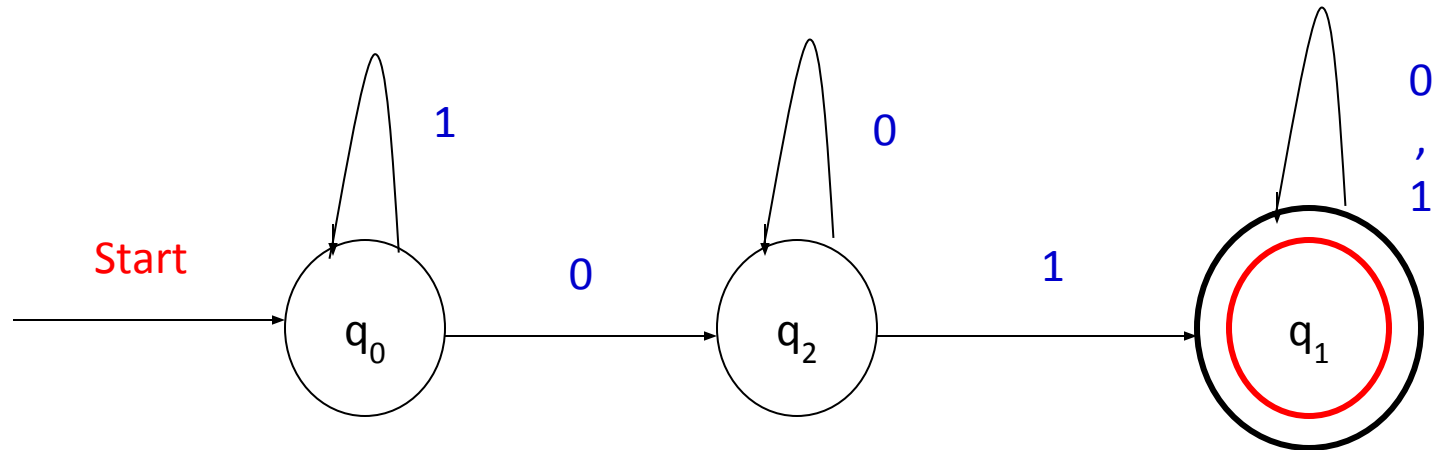


ababb,

Sequence of states: 0, 1, 2, 1, 2, 3
& returns "yes."

Example

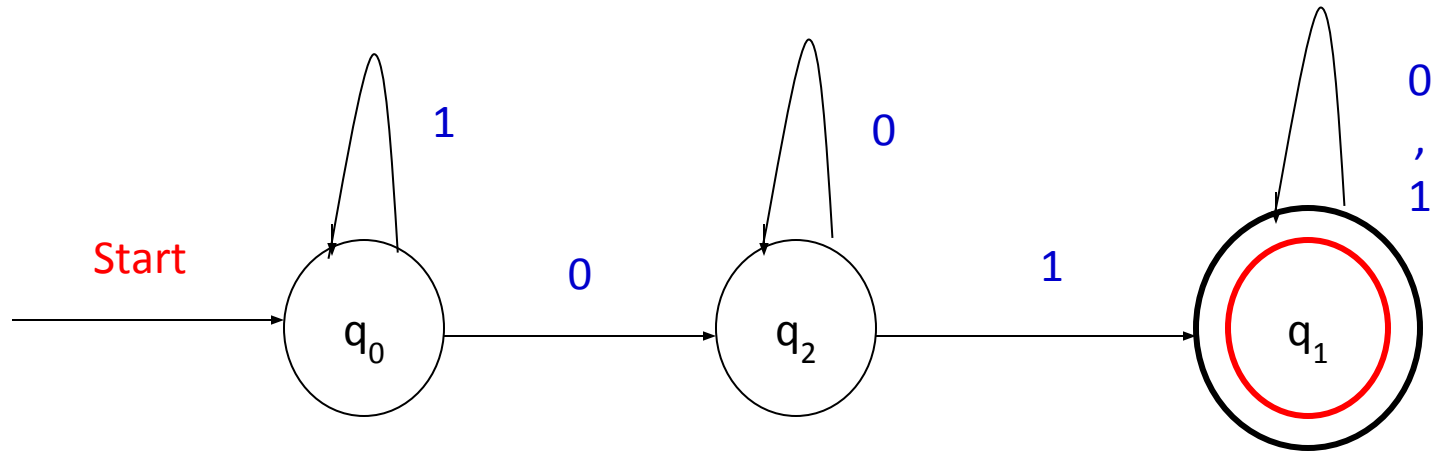
- Draw the Transition Diagram for the DFA accepting all string with a **substring 01**.



$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

Check with the string 01, 11010, 100011,
0111, 110101, 11101101, 111000

Transition Function & Table



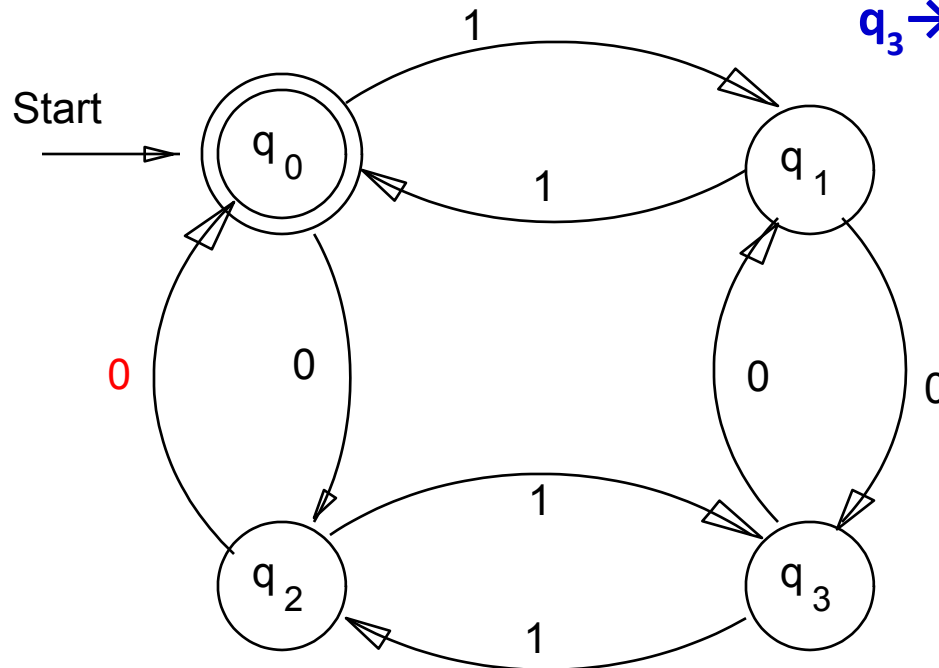
- $(q_0, 0) = q_2$
- $(q_0, 1) = q_0$
- $(q_1, 0) = q_1$
- $(q_1, 1) = q_1$
- $(q_2, 0) = q_2$
- $(q_2, 1) = q_1$

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Example

- Let us design a DFA to accept the language $L = \{w \mid w \text{ has both an even number of 0's and even number of 1's}\}$

$q_0 \rightarrow 0(\text{even}) \ 1(\text{even})$
 $q_1 \rightarrow 0(\text{even}) \ 1(\text{odd})$
 $q_2 \rightarrow 0(\text{odd}) \ 1(\text{even})$
 $q_3 \rightarrow 0(\text{odd}) \ 1(\text{odd})$

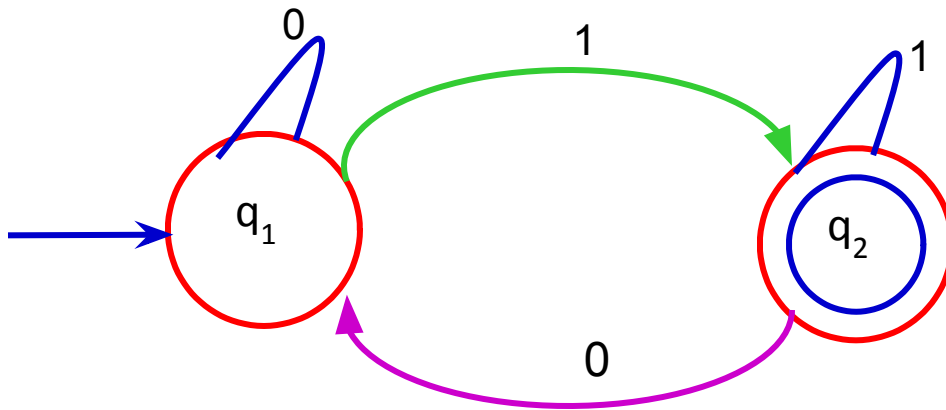


	0	1
$\rightarrow^* q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_3	q_1

Example: Try Yourself

- $A = \{w \mid w \text{ contains at least one } 1 \text{ and an even number of } 0\text{s follow the last } 1\}$
- Hints: $A_1 = (Q, \Sigma, \delta, q_1, F)$
 1. $Q = \{q_1, q_2, q_3\}$
 2. $\Sigma = \{0, 1\}$
 3. δ try yourself
 4. Start state: q_1
 5. Final state: $\{q_2\}$

Example



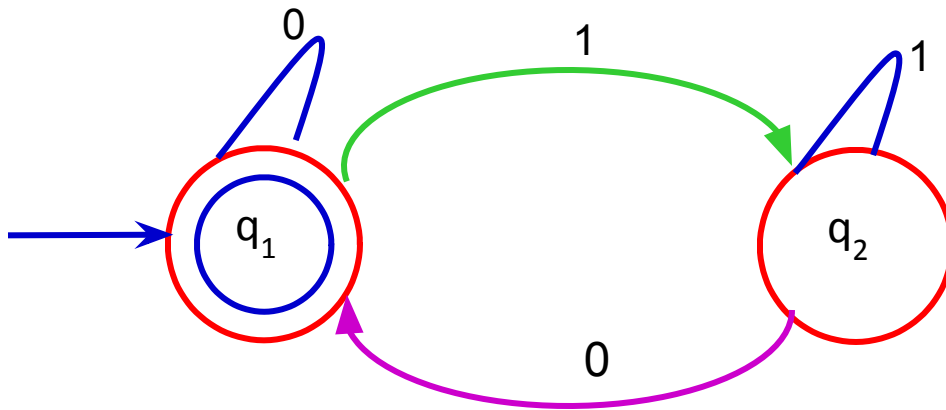
- $A_2 = (\{q_1, q_2\}, (0,1), \delta, q_1, \{q_2\})$
- Transition function, δ

Try: 1101, 11010, 0011010

$L(A_2) = \{w \mid w \text{ ends in a } 1\}$

	0	1
$\rightarrow q_1$	q_1	q_2
$*q_2$	q_1	q_2

Example



- $A_3 = (\{q_1, q_2\}, (0,1), \delta, q_1, \{q_1\})$
- Transition function, δ

Try: 1101, 11010, 0011010

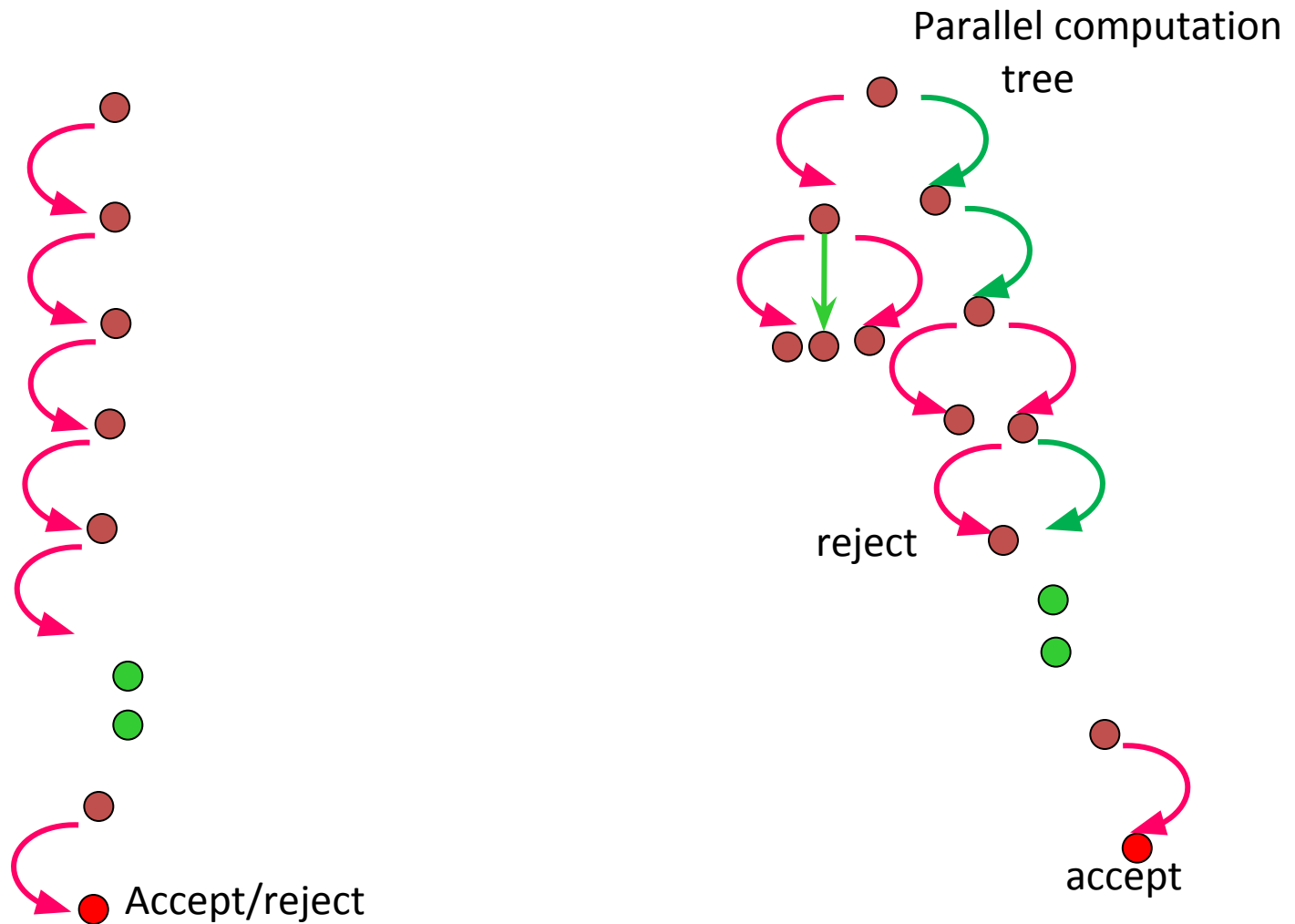
$L(A_3) = \{w \mid w \text{ is } \varepsilon \text{ or ends in a } 0\}$

	0	1
$\rightarrow^* q_1$	q_1	q_2
q_2	q_1	q_2

DFA vs. NFA

- ◇ DFA: δ returns a single state
- ◇ Every state of a DFA always has exactly **one exiting transition arrow** for **each symbol** in the alphabet
- ◇ Labels on the transition arrows are symbols from the alphabet
- NFA: δ returns a set of states
- NFA has an arrow with label \in
- NFA may have arrows labeled with members of alphabet/ \in .
- Zero, one, or many arrows may exit from each state with label \in

DFA vs. NFA



NFA to DFA

- Subset Construction Algorithm

Subset Construction

- Given an NFA with states Q , inputs Σ , transition function δ_N , start state q_0 , and final states F , construct equivalent DFA with:
 - **States 2^Q** (Set of subsets of Q).
 - Inputs Σ .
 - Start state $\{q_0\}$.
 - Final states = all those with a member of F .

Subset Construction

- Given, NFA: $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$
- Goal: DFA, $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$
- $L(D) = L(N)$

States

- Q_D is the set of subsets of Q_N
 - Q_D is the power set of Q_N
 - If Q_N has n states, Q_D will have 2^n states
- Inaccessible states can be thrown away, so effectively, the number of states $D \ll 2^n$

Subset construction

Final States

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is F_D is all sets of N 's states that include at least one accepting state of N .

Transition Function

- The transition function δ_D is defined by:
 $\delta_D(\{q_1, \dots, q_k\}, a)$ is the union over all $i = 1, \dots, k$ of $\delta_N(q_i, a)$.

Subset Construction: Example 1

- **Example:** We'll construct the DFA equivalent of our “chessboard” NFA.

1	2	3
4	5	6
7	8	9

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}		
{5}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
* 8	4,6	5,7,9
9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}		
{2,4,6,8}		
{1,3,5,7}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}	{2,4,6,8}	{1,3,7,9}
{2,4,6,8}		
{1,3,5,7}		
* {1,3,7,9}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}	{2,4,6,8}	{1,3,7,9}
{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
{1,3,5,7}		
* {1,3,7,9}		
* {1,3,5,7,9}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}	{2,4,6,8}	{1,3,7,9}
{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
* {1,3,7,9}		
* {1,3,5,7,9}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

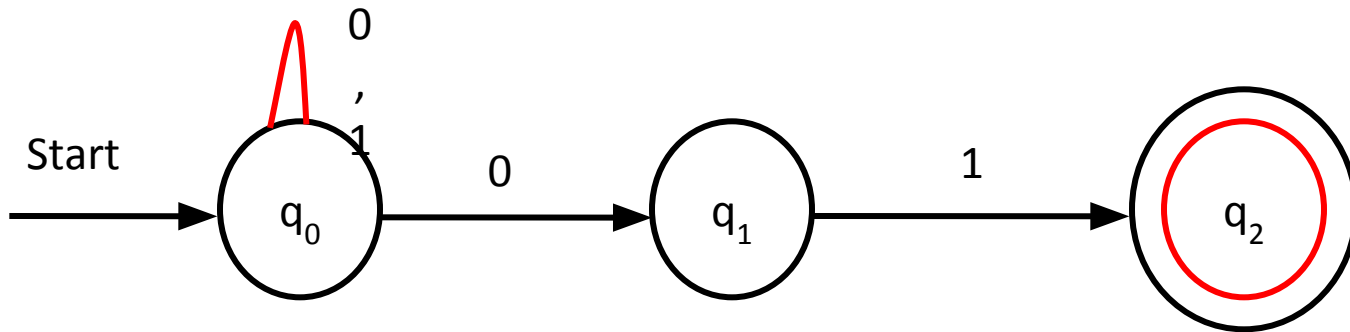
	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}	{2,4,6,8}	{1,3,7,9}
{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
* {1,3,7,9}	{2,4,6,8}	{5}
* {1,3,5,7,9}		

Example: Subset Construction

	r	b
→ 1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
* 9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}	{2,4,6,8}	{1,3,7,9}
{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
* {1,3,7,9}	{2,4,6,8}	{5}
* {1,3,5,7,9}	{2,4,6,8}	{1,3,5,7,9}

Example 2



$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(\{q_0, 0\}) \cup \delta_N(\{q_2, 0\}) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$

$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(\{q_0, 1\}) \cup \delta_N(\{q_2, 1\}) = \{q_0\} \cup \emptyset = \{q_0\}$$

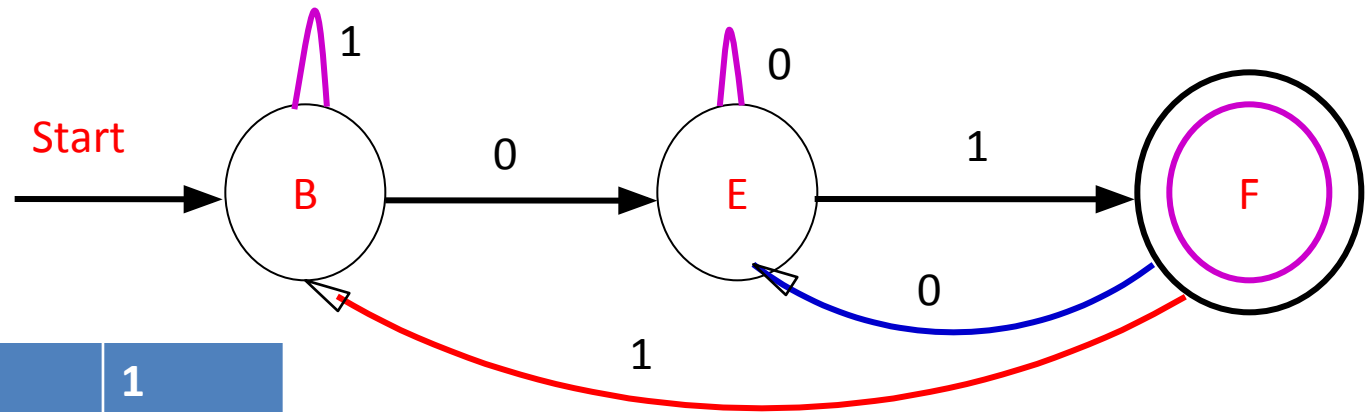
	0	1
\emptyset	\emptyset	\emptyset
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	\emptyset	$\{q_2\}$
$\ast\{q_2\}$	\emptyset	\emptyset
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\ast\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\ast\{q_1, q_2\}$	\emptyset	$\{q_2\}$
$\ast\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Example 2

- NFA N Accepts all strings that end in 01
- N's set of states: $\{q_0, q_1, q_2\} = 03$
- Subset construction: DFA need $2^3 = 8$ states
- Assign new names: A for \emptyset , B for $\{q_0\}$

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
*D	A	A
E	E	F
*F	E	B
*G	A	D
*H	E	F

Example 2

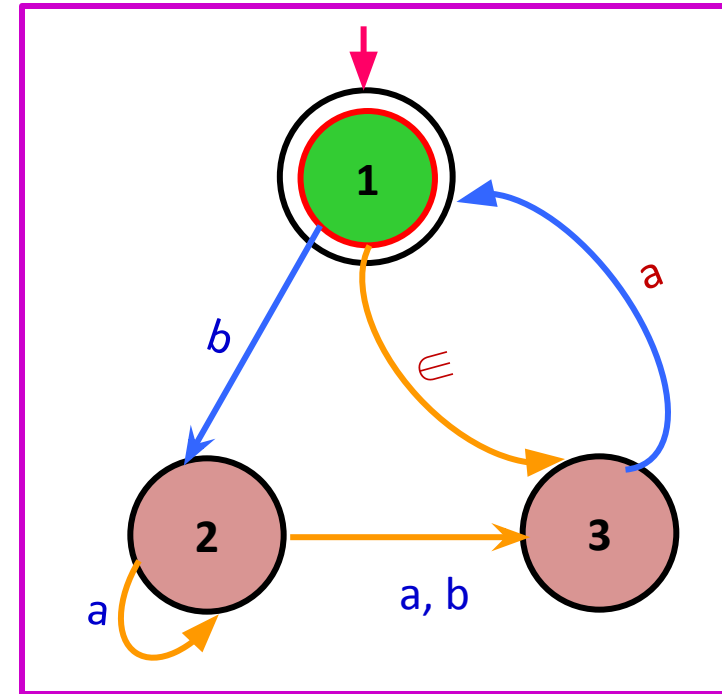


	0	1
A	A	A
→B	E	B
C	A	D
*D	A	A
E	E	F
*F	E	B
*G	A	D
*H	E	F

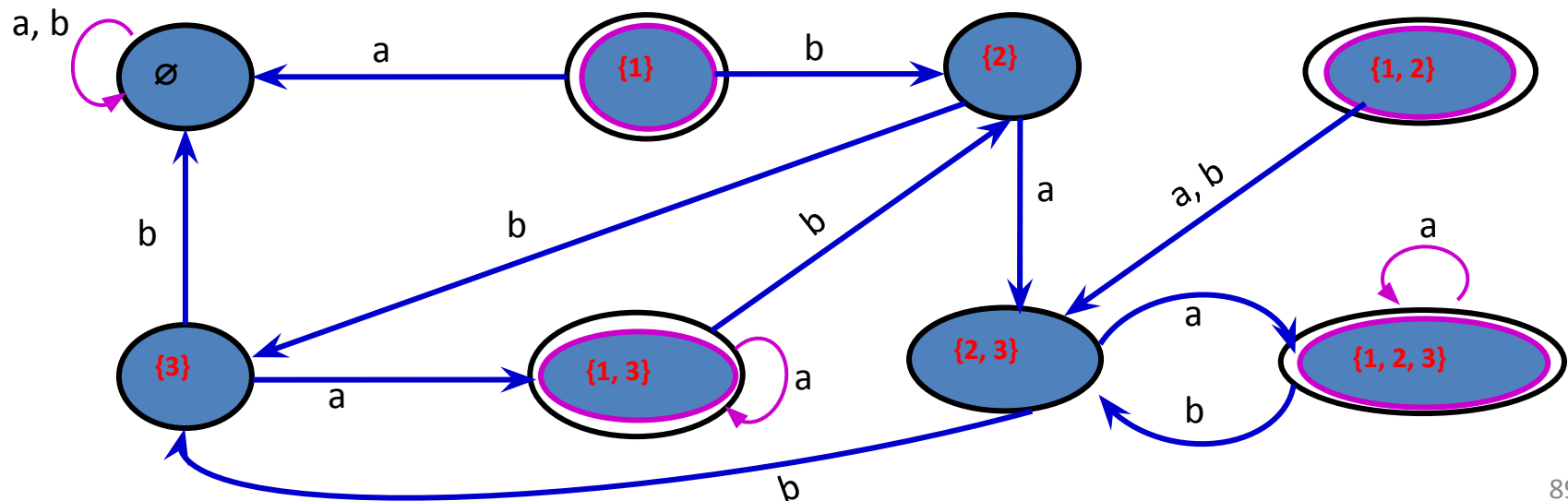
- From 08 states, starting in start state B, can only reach states B, E & F
- other 05 states are inaccessible from B

Example 3

- $N = (Q, \{a, b\}, \delta, 1, \{1\})$
- $Q = \{1, 2, 3\} = 03$ states
- DFA states = 08
- $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

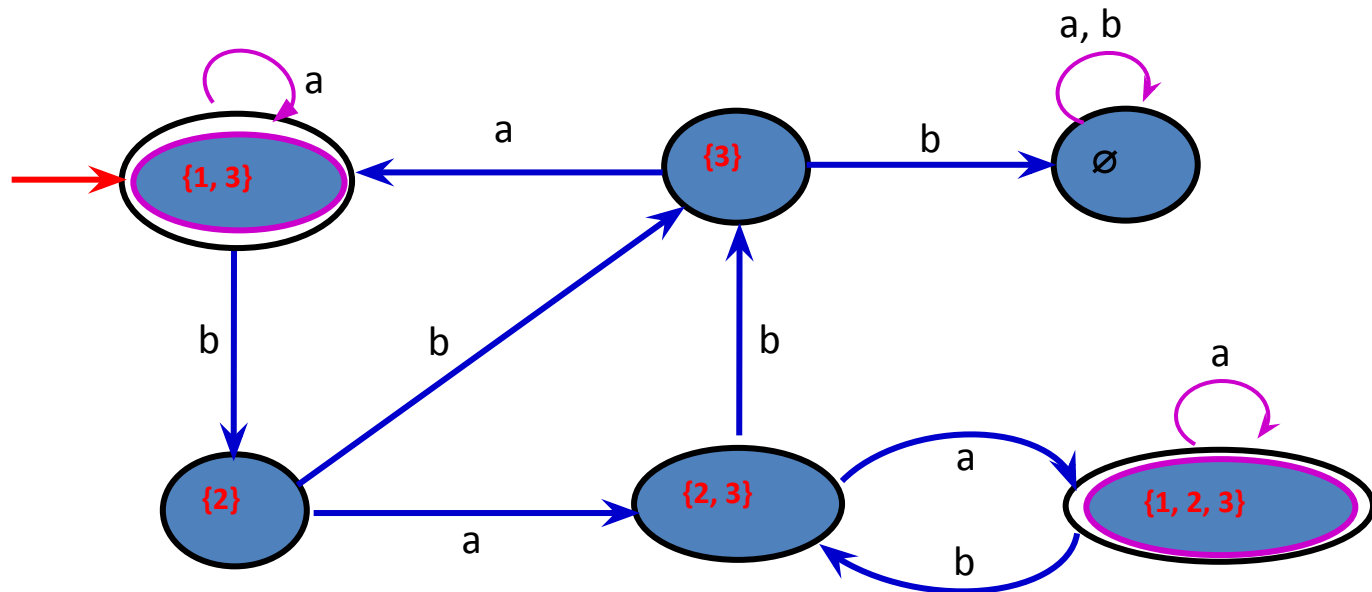


	a	b	ϵ
\emptyset	\emptyset	\emptyset	\emptyset
{1}	\emptyset	{2}	{3}
{2}	{2, 3}	{3}	\emptyset
{3}	{1, 3}	\emptyset	\emptyset
{1, 2}	{2, 3}	{2, 3}	\emptyset
{1, 3}	{1, 3}	{2}	\emptyset
{2, 3}	{1, 2, 3}	{3}	\emptyset
{1, 2, 3}	{1, 2, 3}	{2, 3}	\emptyset



Example 3

Simplified: no incoming arrows point at states $\{1\}$ & $\{1, 2\}$
May be removed without affecting the performance

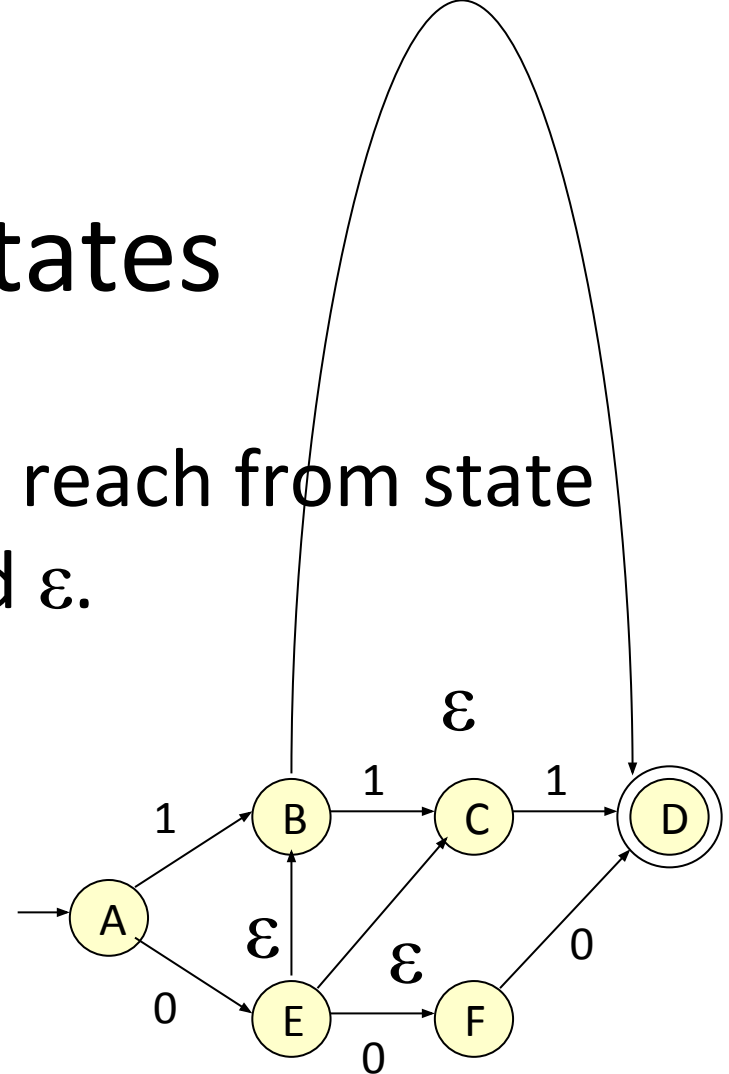


Closure of States

- $CL(q)$ = set of states you can reach from state q following only arcs labeled ϵ .

- **Example:** $CL(A) = \{A\}$;

$$CL(E) = \{B, C, D, E\}.$$



- Closure of a set of states = union of the closure of each state.

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

Input: An NFA N .

OUTPUT: A DFA D accepting the same language as N

METHOD: constructs a transition table **Dtran** for D .

Each state of D is a set of NFA states, and construct **Dtran** so D will simulate

"in parallel" all possible moves N can make on a given input string.

s is a single state of N , while **T** is a set of states of N .

Operations on NFA states

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



Set of states

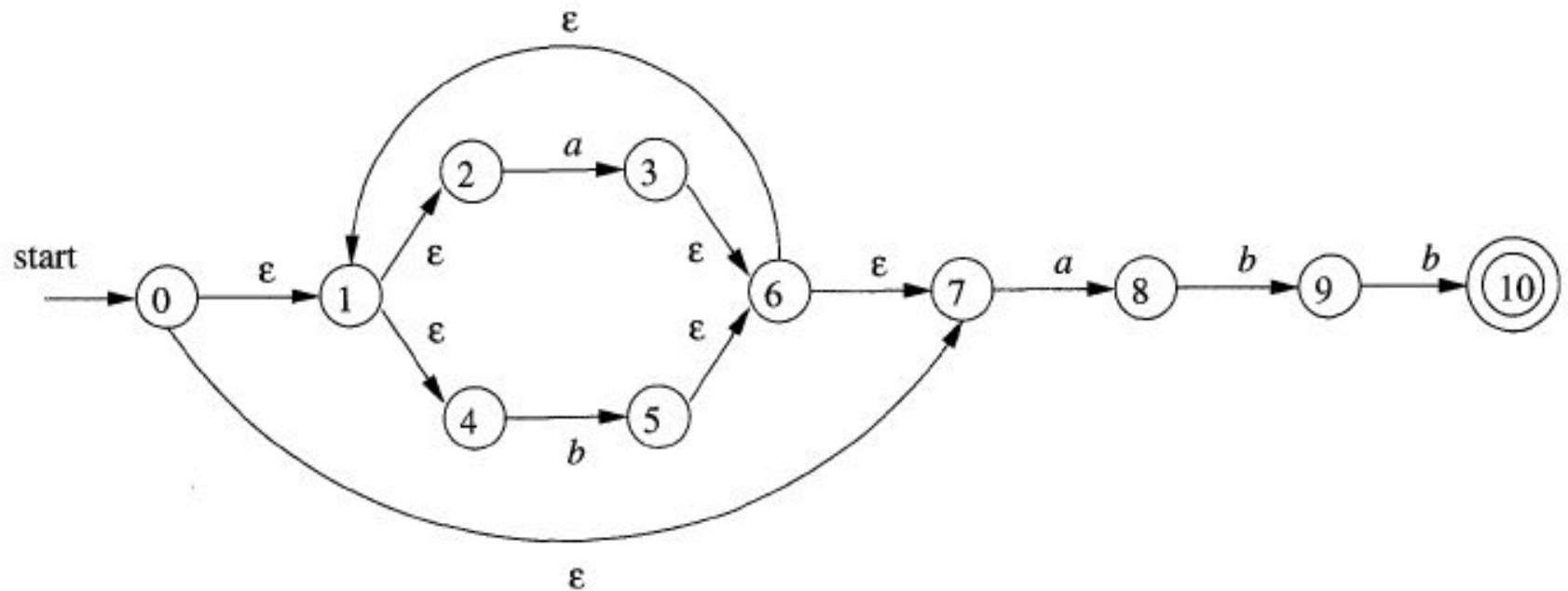
The subset construction

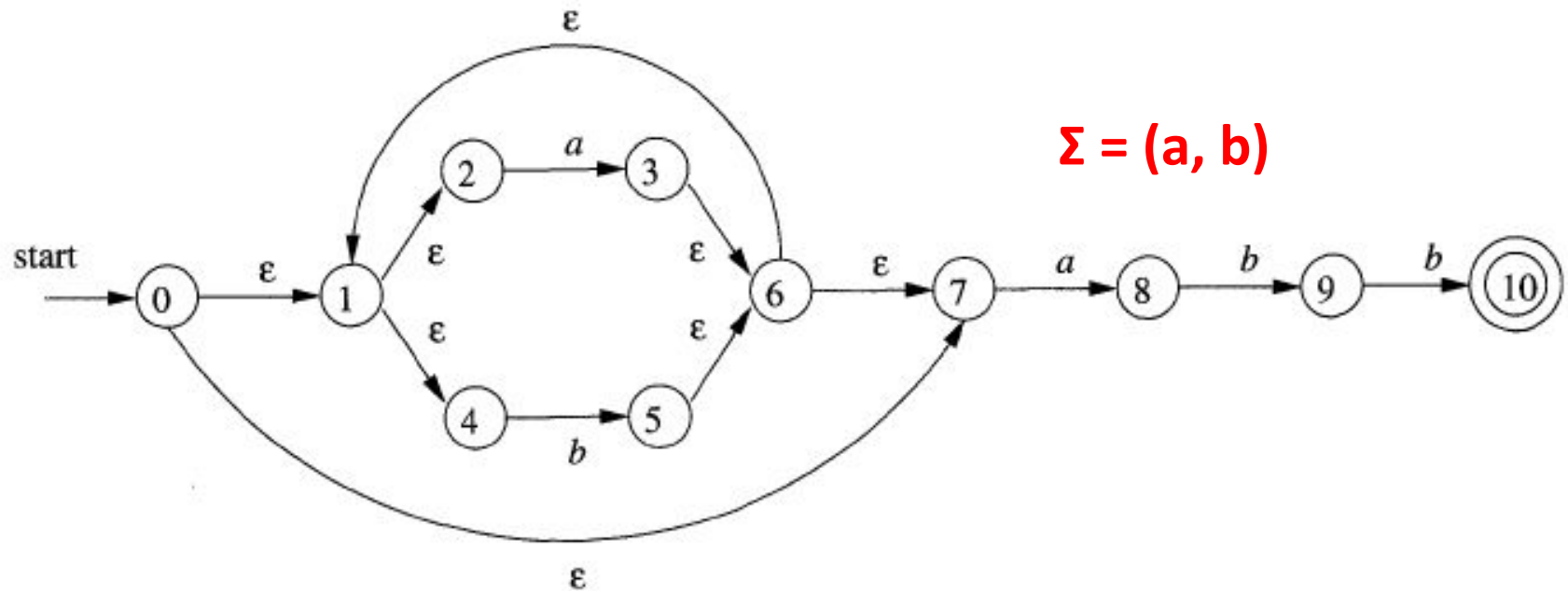
```
while ( there is an unmarked state  $T$  in  $Dstates$  ) {  
    mark  $T$ ;  
    for ( each input symbol  $a$  ) {  
         $U = \epsilon\text{-closure}(\text{move}(T, a))$ ;  
        if (  $U$  is not in  $Dstates$  )  
            add  $U$  as an unmarked state to  $Dstates$ ;  
        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

```
push all states of  $T$  onto  $stack$ ;  
initialize  $\epsilon\text{-closure}(T)$  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  $\epsilon$  )  
        if (  $u$  is not in  $\epsilon\text{-closure}(T)$  ) {  
            add  $u$  to  $\epsilon\text{-closure}(T)$ ;  
            push  $u$  onto  $stack$ ;  
        }  
}
```

Computing $\epsilon\text{-closure}(T)$

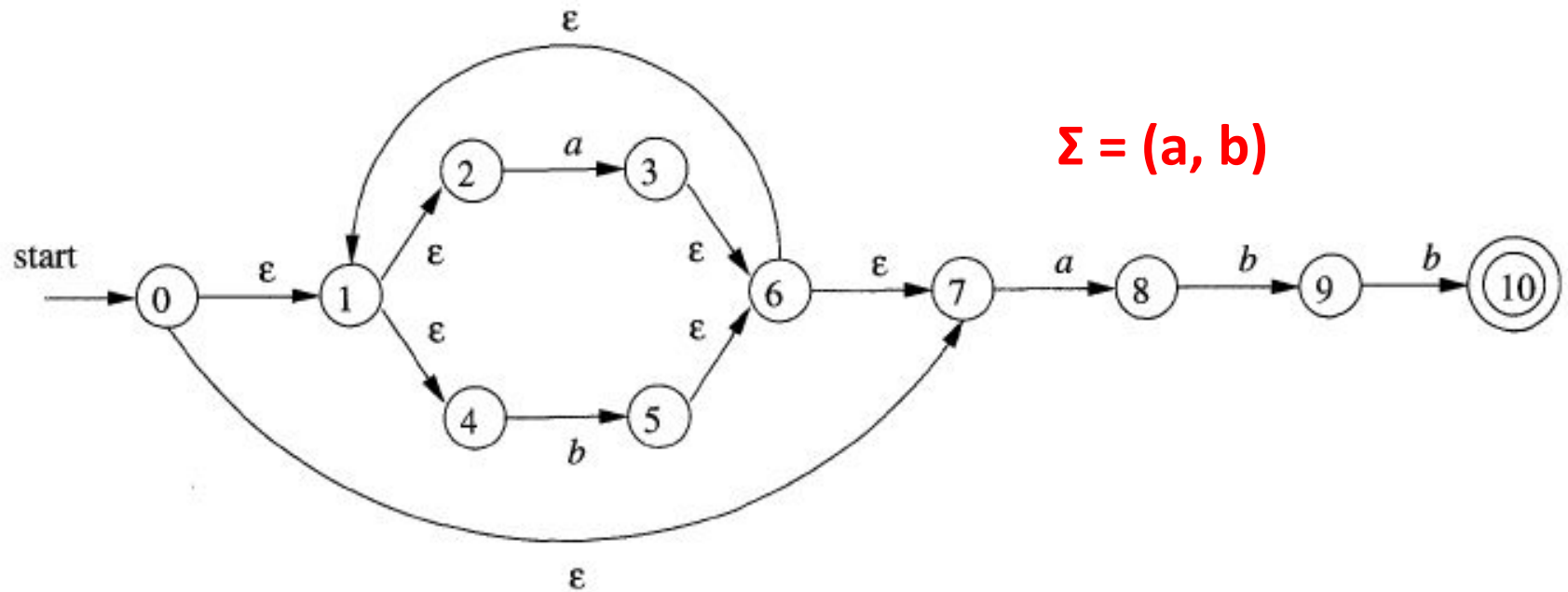
Example: NFA accepting $R = (alb)^*abb$





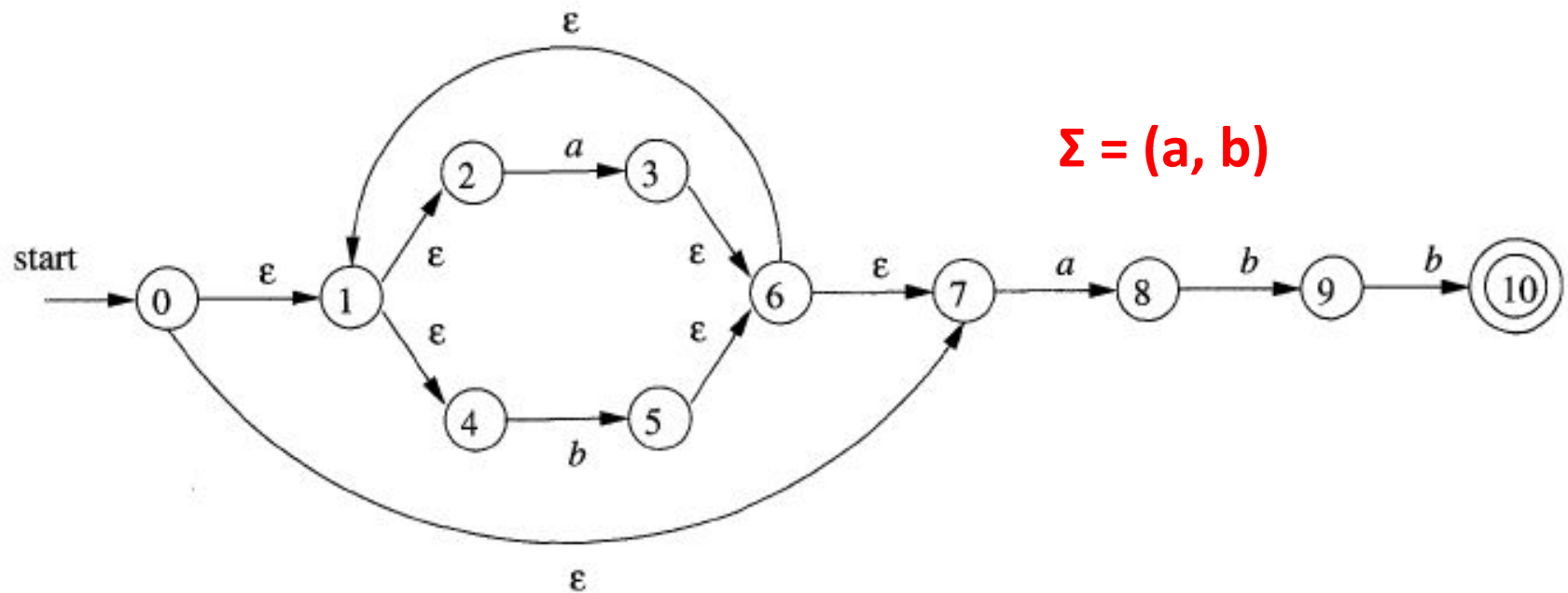
Marked

- ϵ -closure(0) = {0, 1, 2, 4, 7} = **A**
- **Mark A, Compute Dtran [A, a] & Dtran [A, b]**
- **Dtran [A, a]** = ϵ -closure (move(A, a))
 - = ϵ -closure (move({0, 1, 2, 4, 7}, a))
 - = ϵ -closure ({3, 8})
 - = {3, 6, 7, 1, 2, 4} \cup {8}
 - = **{1, 2, 3, 4, 6, 7, 8} = B**

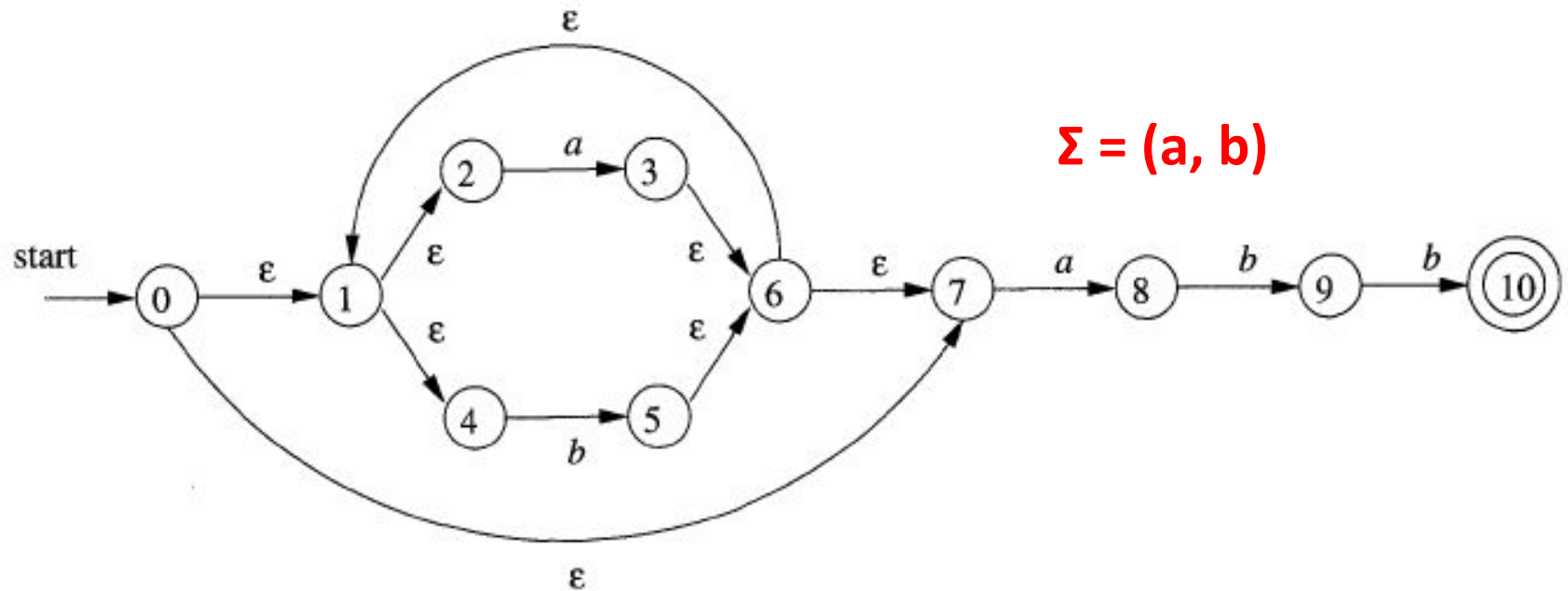


- **Dtran [A, b]** = ϵ -closure (move(A, b))
 = ϵ -closure (move({0, 1, 2, 4, 7}, b))
 = ϵ -closure ({5})
 = {5, 6, 7, 1, 2, 4}

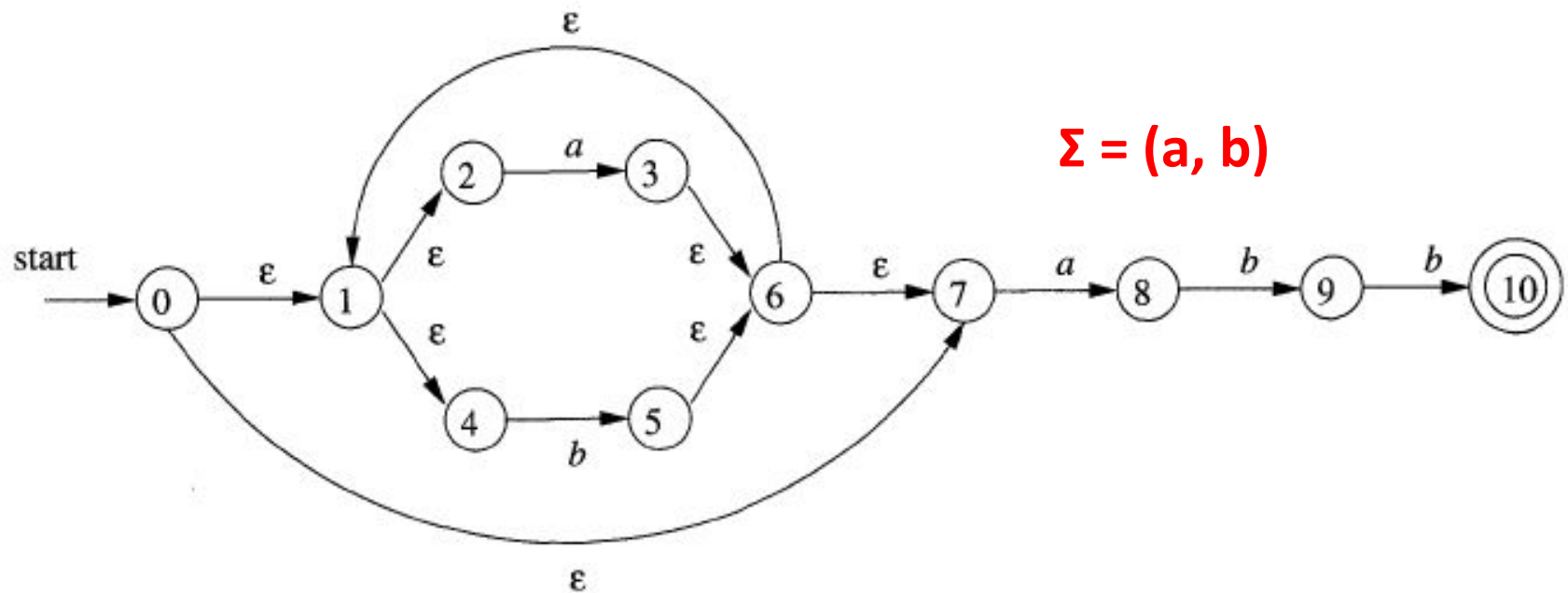
Dtran [A, b] = {1, 2, 4, 5, 6, 7} = C



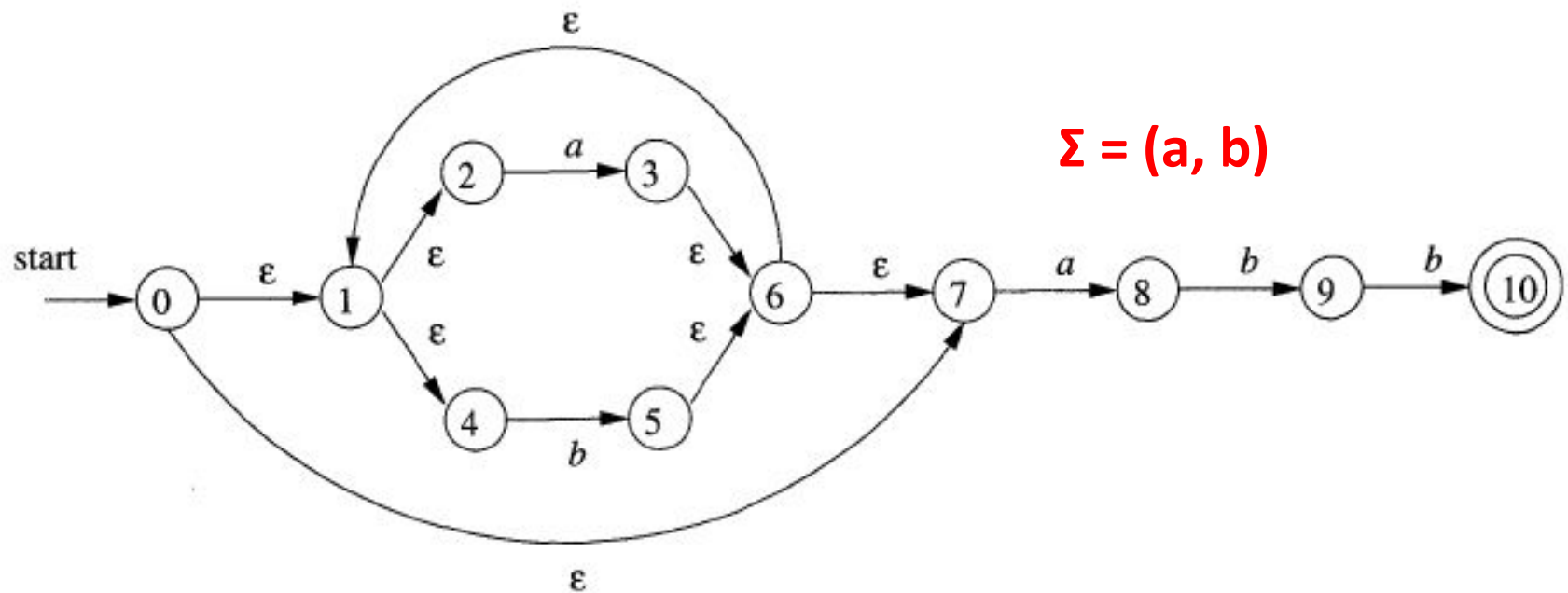
- **Mark B, Compute Dtran [B, a] & Dtran [B, b]**
 - **Dtran [B, a] = ϵ -closure (move(B, a))**
 - = ϵ -closure (move({1, 2, 3, 4, 6, 7, 8}, a))
 - = ϵ -closure ({3, 8})
 - = {3, 6, 7, 1, 2, 4} \cup {8}
- Dtran [B, a] = {1, 2, 3, 4, 6, 7, 8} = B**



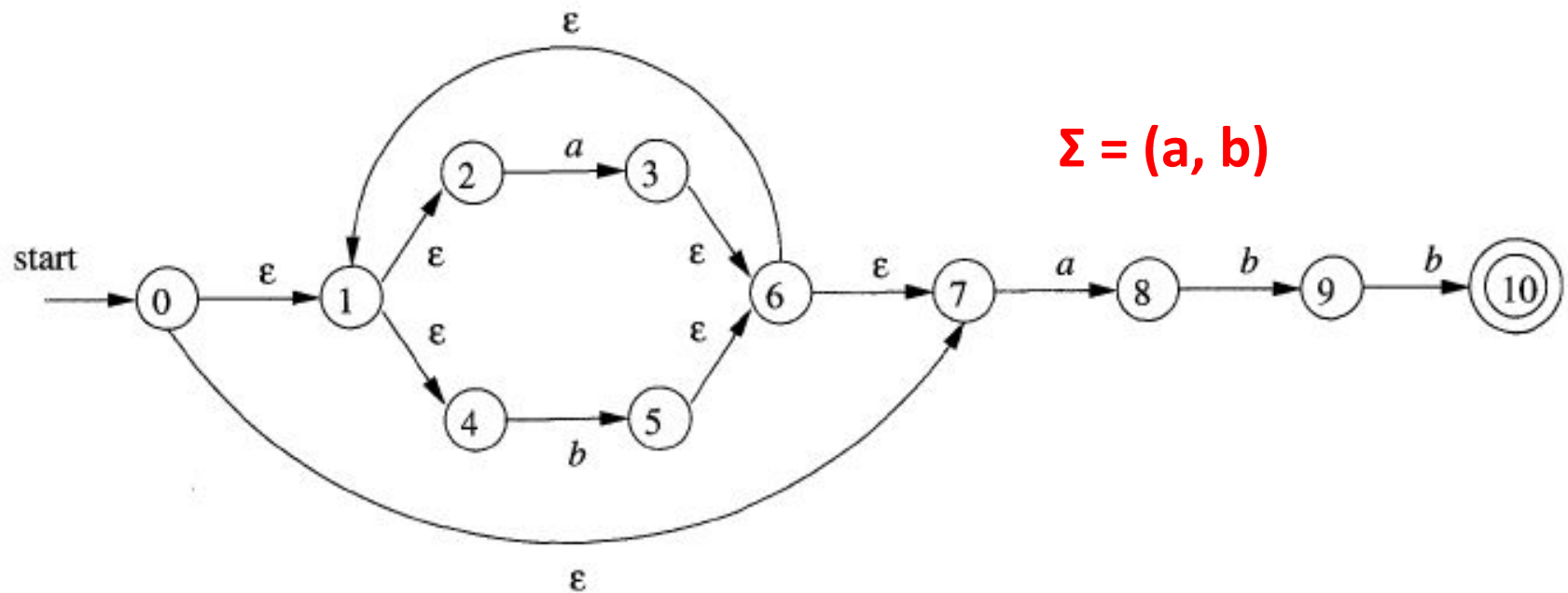
- **Compute Dtran [B, b]**
 - **Dtran [B, b] = ϵ -closure (move(B, b))**
 - = ϵ -closure (move({1, 2, 3, 4, 6, 7, 8}, b))**
 - = ϵ -closure ({5, 9})**
 - = {5, 6, 7, 1, 2, 4} \cup {9}**
- Dtran [B, b] = {1, 2, 4, 5, 6, 7, 9} = D**



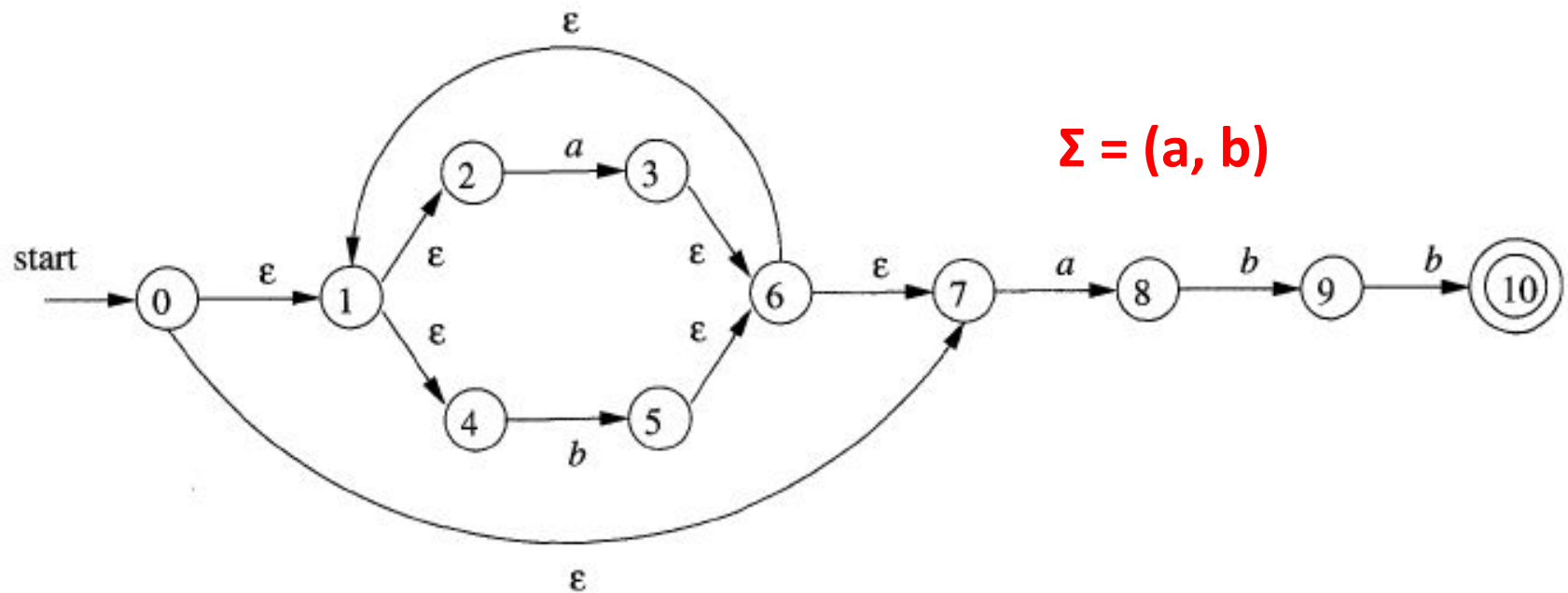
- **Mark C, Compute Dtran [C, a] & Dtran [C, b]**
 - **Dtran [C, a] = ϵ -closure (move(C, a))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7}, a))
 - = ϵ -closure ({3, 8})
 - = {3, 6, 7, 1, 2, 4} \cup {8}
- Dtran [C, a] = {1, 2, 3, 4, 6, 7, 8} = B**



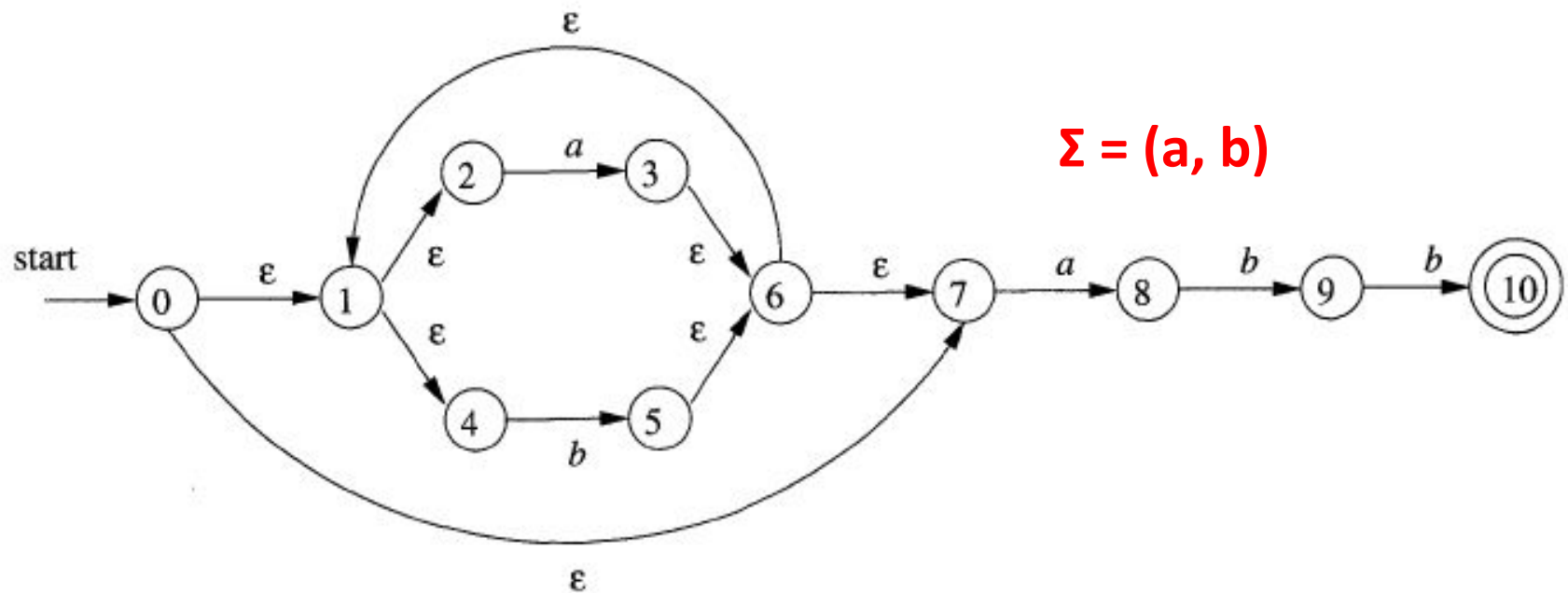
- **Compute Dtran [C, b]**
 - **Dtran [C, b] = ϵ -closure (move(C, b))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7}, b))
 - = ϵ -closure ({5})
 - = {5, 6, 7, 1, 2, 4}
- Dtran [C, b] = {1, 2, 4, 5, 6, 7} = C**



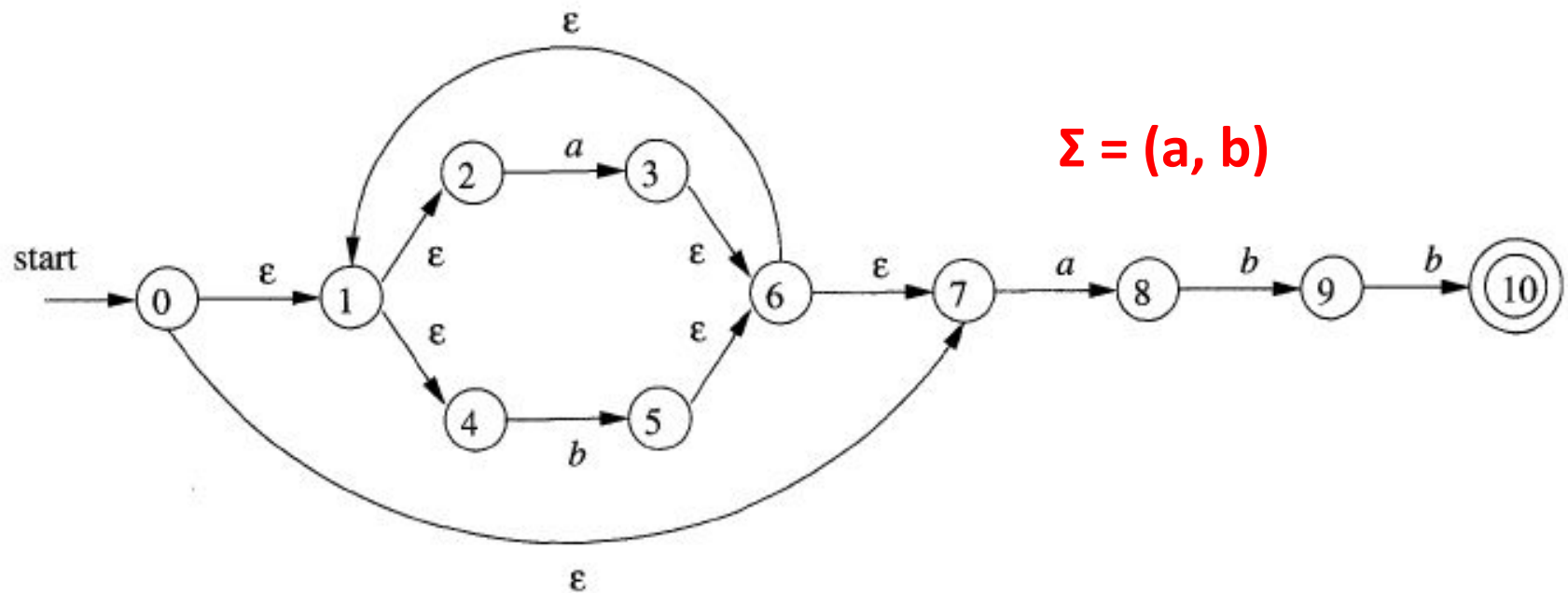
- **Mark D, Compute Dtran [D, a] and Dtran [D, b]**
 - **Dtran [D, a] = ϵ -closure (move(D, a))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7, 9}, a))**
 - = ϵ -closure ({3, 8})**
 - = {3, 6, 7, 1, 2, 4} \cup {8}**
- Dtran [D, a] = {1, 2, 3, 4, 6, 7, 8} = B**



- **Compute Dtran [D, b]**
 - **Dtran [D, b] = ϵ -closure (move(D, b))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7, 9}, b))**
 - = ϵ -closure ({5, 10})**
 - = {5, 6, 7, 1, 2, 4} \cup {10}**
- Dtran [D, b] = {1, 2, 4, 5, 6, 7, 10} = E**



- **Mark E, Compute Dtran [E, a] and Dtran [E, b]**
- **Dtran [E, a] = ϵ -closure (move(E, a))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7, 10}, a))**
 - = ϵ -closure ({3, 8})**
 - = {3, 6, 7, 1, 2, 4} \cup {8}**
- Dtran [E, a] = {1, 2, 3, 4, 6, 7, 8} = B**



- **Compute Dtran [E, b]**
 - **Dtran [E, b] = ϵ -closure (move(E, b))**
 - = ϵ -closure (move({1, 2, 4, 5, 6, 7, 10}, b))
 - = ϵ -closure ({5})
 - = {5, 6, 7, 1, 2, 4}
- Dtran [E, b] = {1, 2, 4, 5, 6, 7} = C**

Summary

Dtran [A, a] = {1, 2, 3, 4, 6, 7, 8} = B

Dtran [A, b] = {1, 2, 4, 5, 6, 7} = C

Dtran [B, a] = {1, 2, 3, 4, 6, 7, 8} = B

Dtran [B, b] = {1, 2, 4, 5, 6, 7, 9} = D

Dtran [C, a] = {1, 2, 3, 4, 6, 7, 8} = B

Dtran [C, b] = {1, 2, 4, 5, 6, 7} = C

Dtran [D, a] = {1, 2, 3, 4, 6, 7, 8} = B

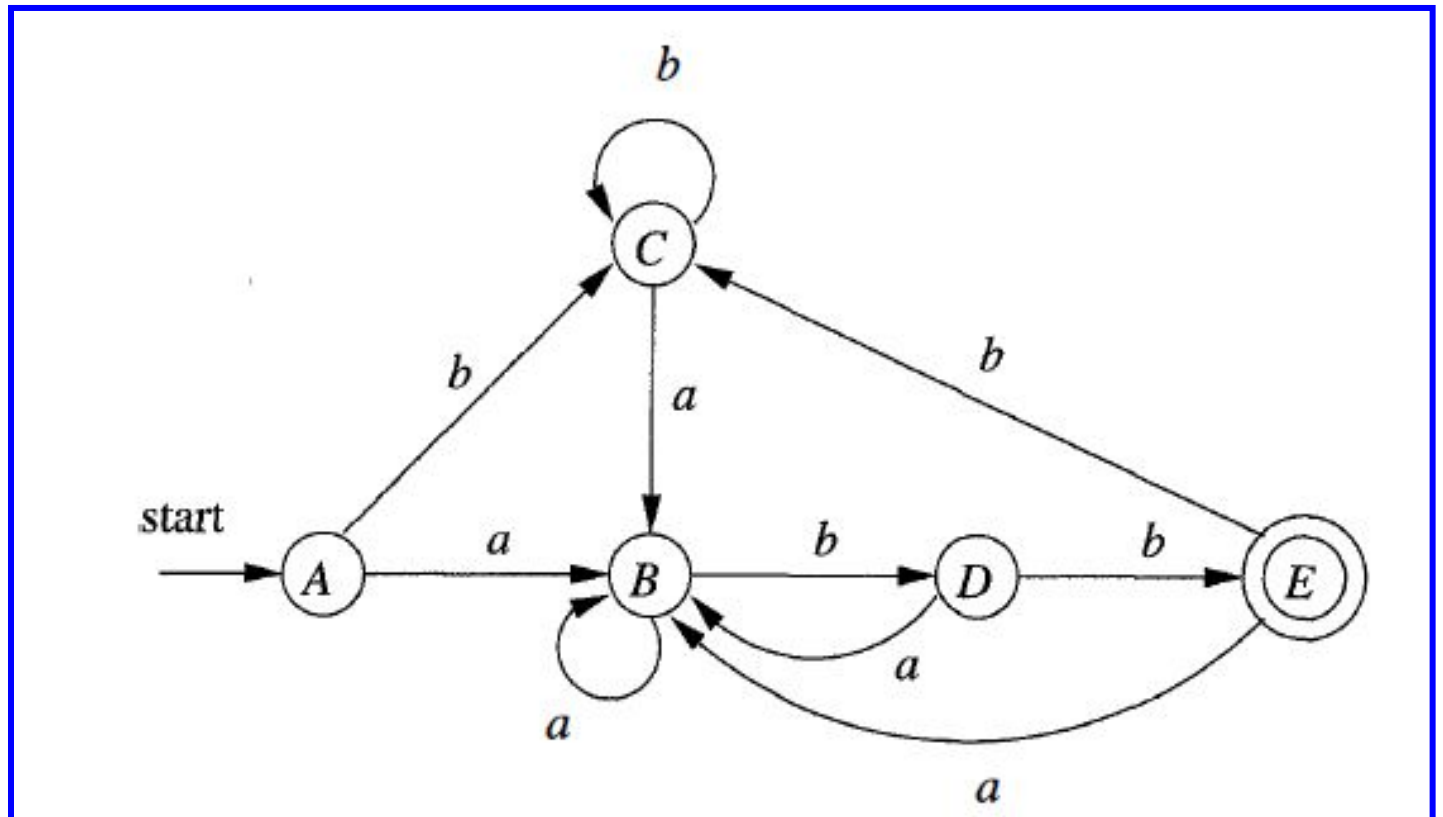
Dtran [D, b] = {1, 2, 4, 5, 6, 7, 10} = E

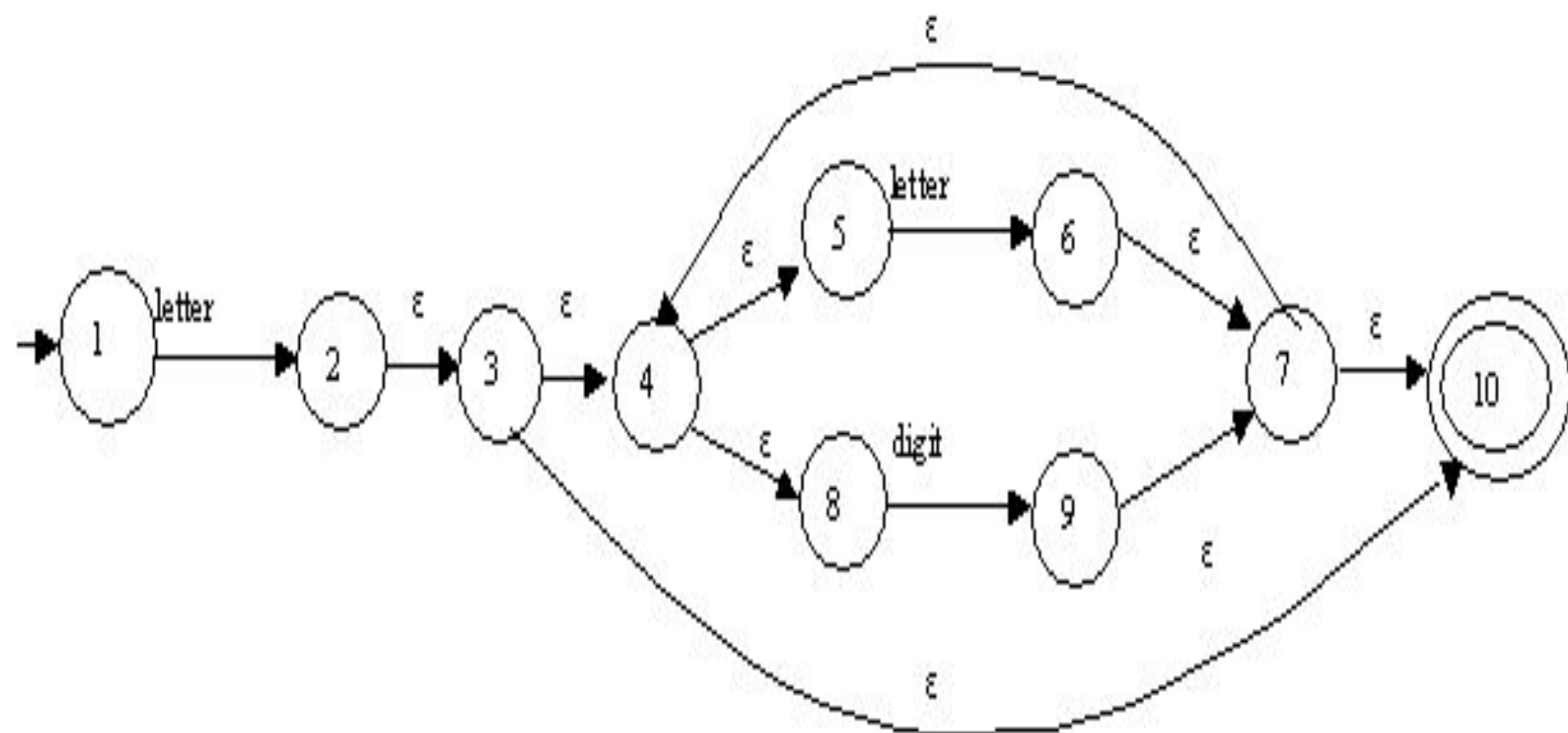
Dtran [E, a] = {1, 2, 3, 4, 6, 7, 8} = B

Dtran [E, b] = {1, 2, 4, 5, 6, 7} = C

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

NFA State	DFA State	a	b
$\rightarrow\{0, 1, 2, 4, 7\}$	$\rightarrow A$	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





Regular Expression to NFA

- McNaughton-Yamada-Thompson Algorithm

Construction of an NFA from a Regular Expression

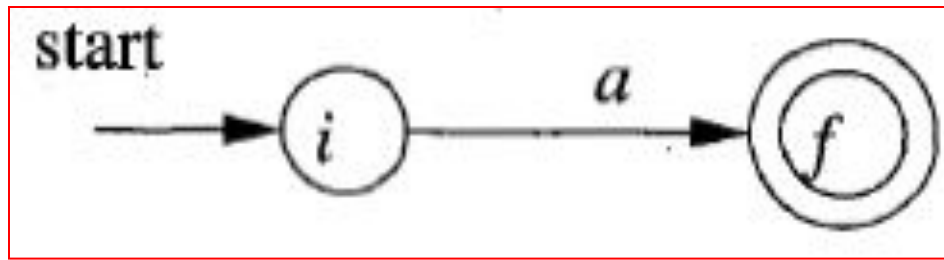
- **Algorithm:** The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.
- **INPUT:** A regular expression r over alphabet Σ
- **OUTPUT:** An NFA N accepting $L(r)$.
- **METHOD:**
 - Begin by parsing r into its constituent sub expressions.
 - The rules for constructing an NFA consist of
 - basis rules for handling sub expressions with no operators,
 - inductive rules for constructing larger NFA's from the NFA's for the immediate sub expressions of a given expression

Basis

1. For expression ϵ ($r = \epsilon$) construct the NFA



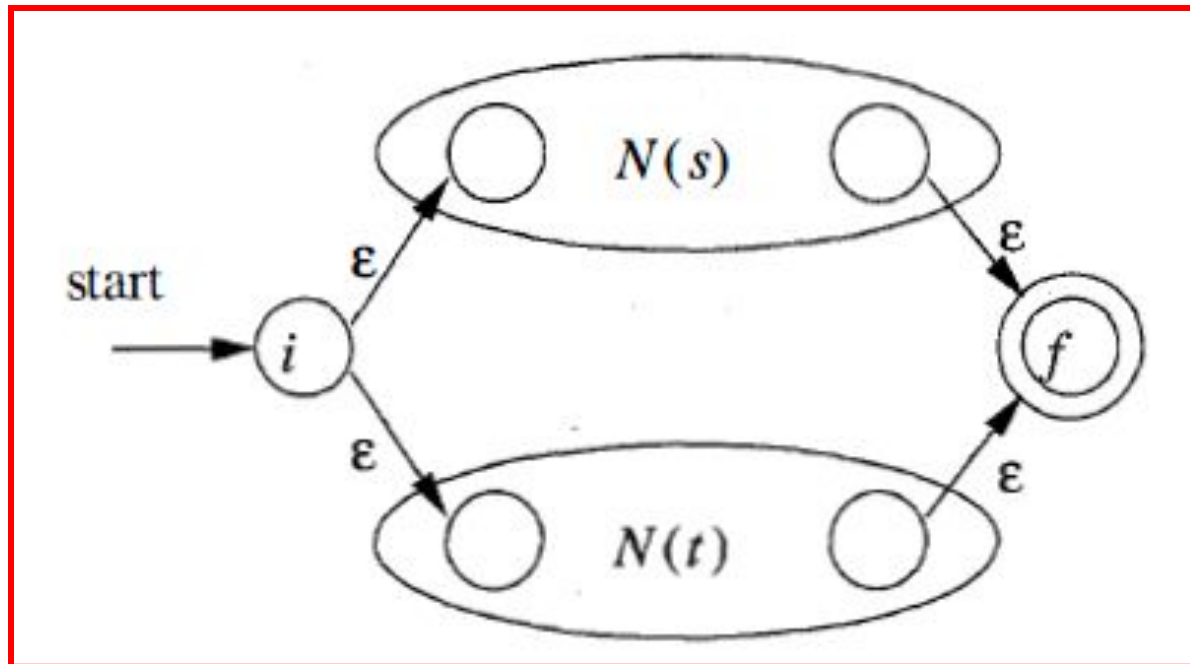
2. For any sub expression a ($r = a$), construct NFA



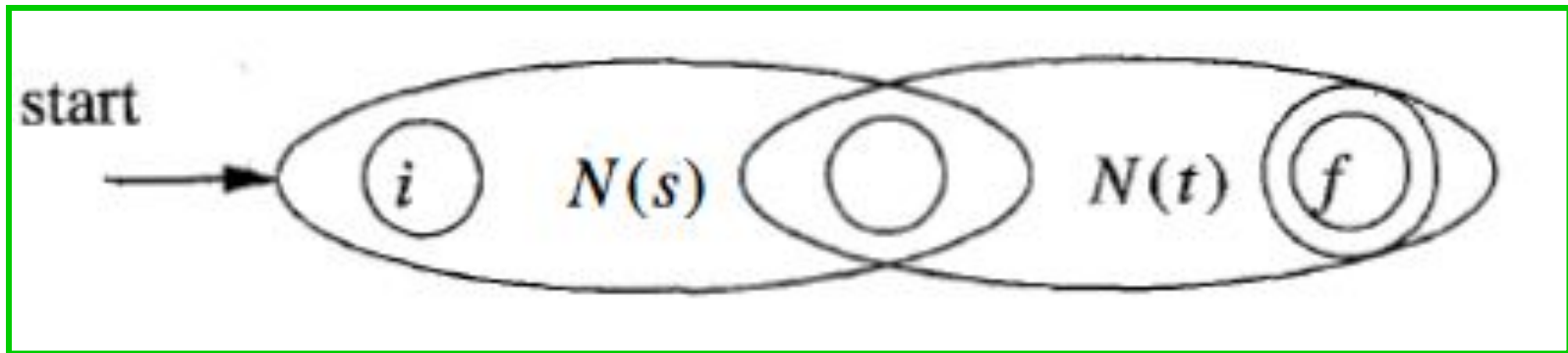
INDUCTION

- Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t , respectively.

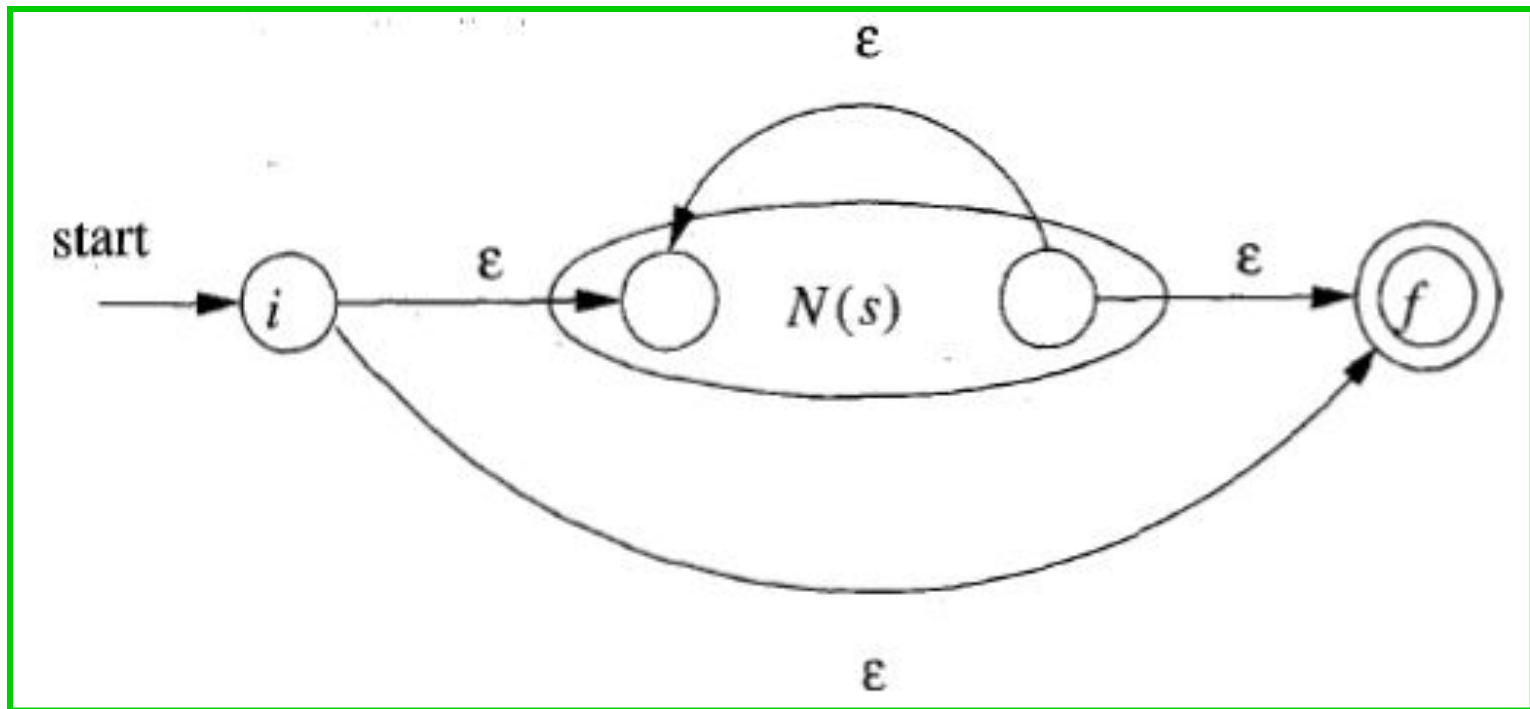
1. $r = s/t$ (union)



2. $r = st$ (Concatenation)



3. $r = s^*$ (Closure/star)

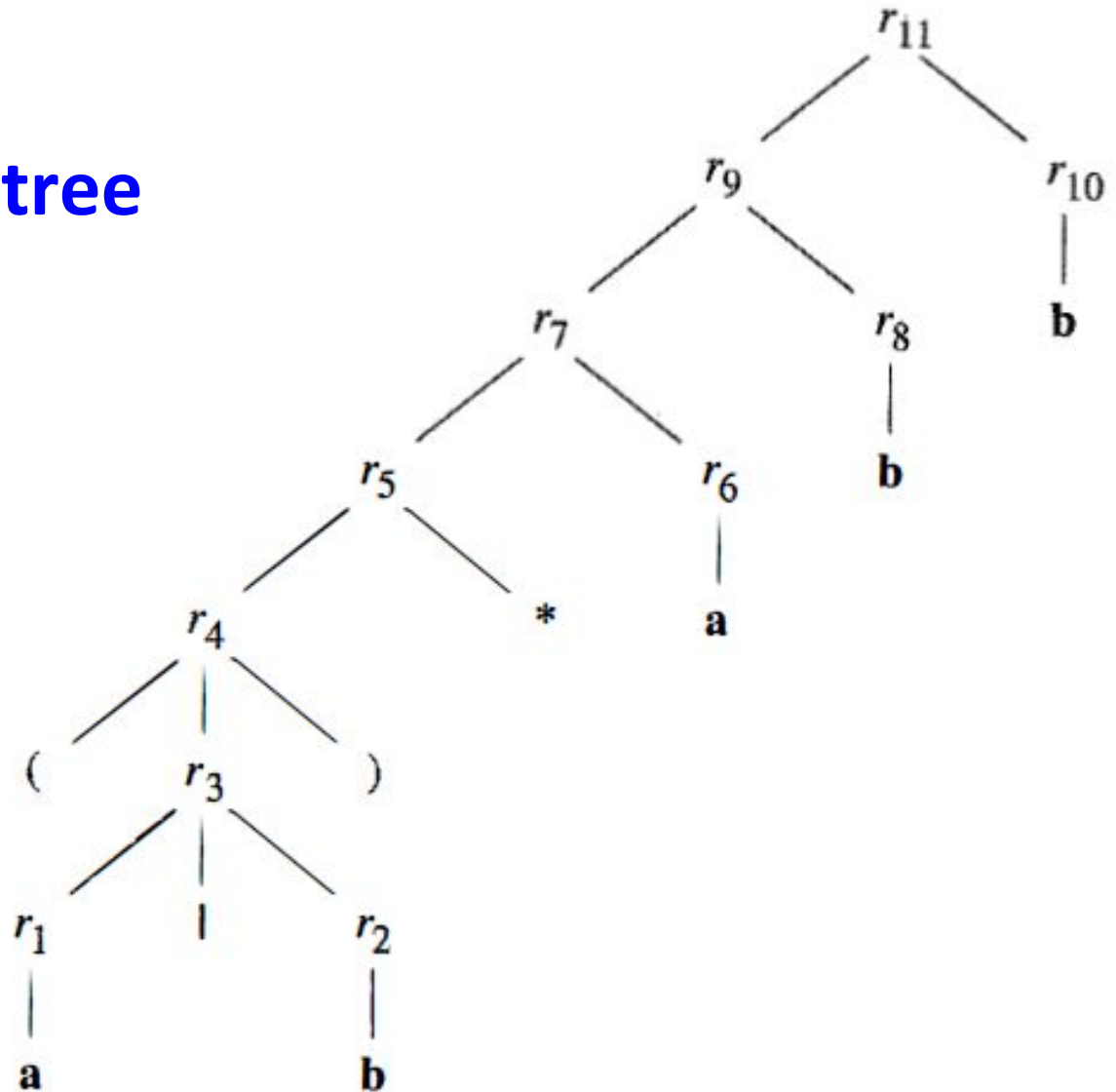


Observations

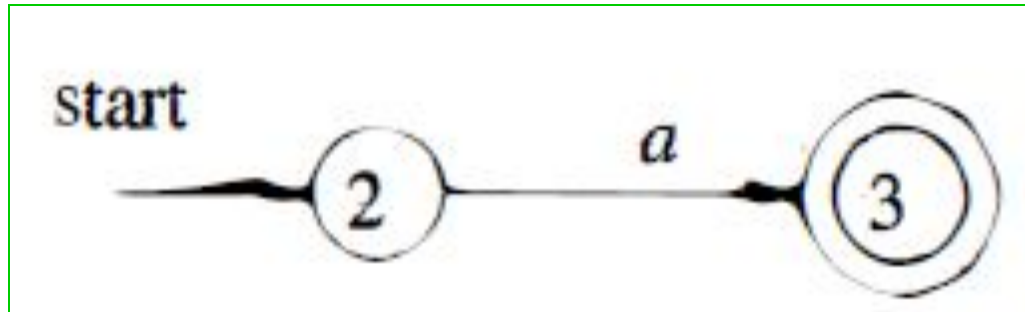
- 1. $N(r)$ has at most twice as many states as there are operators and operands in r .
 - This bound follows from the fact that each step of the algorithm creates at most two new states.
- 2. $N(r)$ has one start state and one accepting state.
 - The accepting state has no outgoing transitions,
 - start state has no incoming transitions.
- 3. Each state of $N(r)$ other than the accepting state has either
 - one outgoing transition on a symbol in Σ
 - or two outgoing transitions, both on ϵ

Example: Construct an NFA for $r = (alb)^*abb$

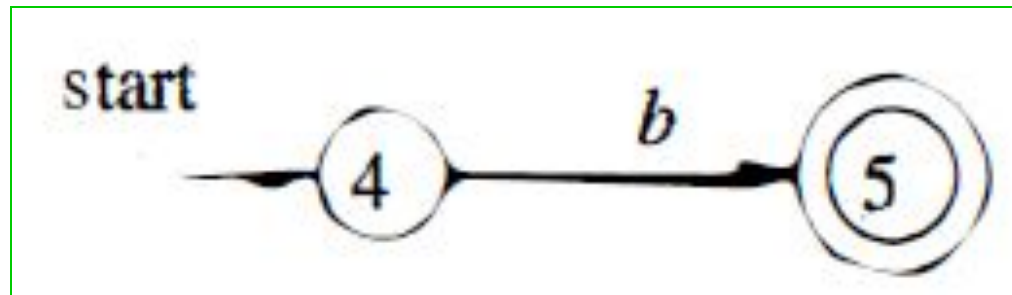
Parse tree



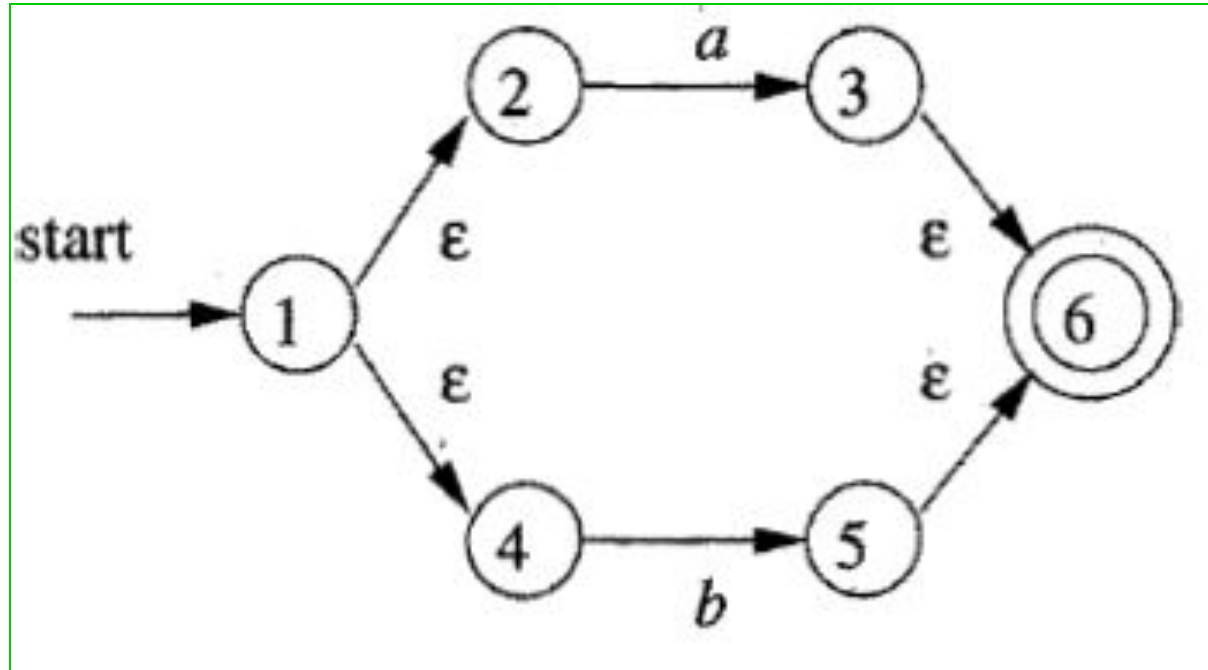
Step 1: For sub expression $r_1 = a$



Step 2: For sub expression $r_2 = b$

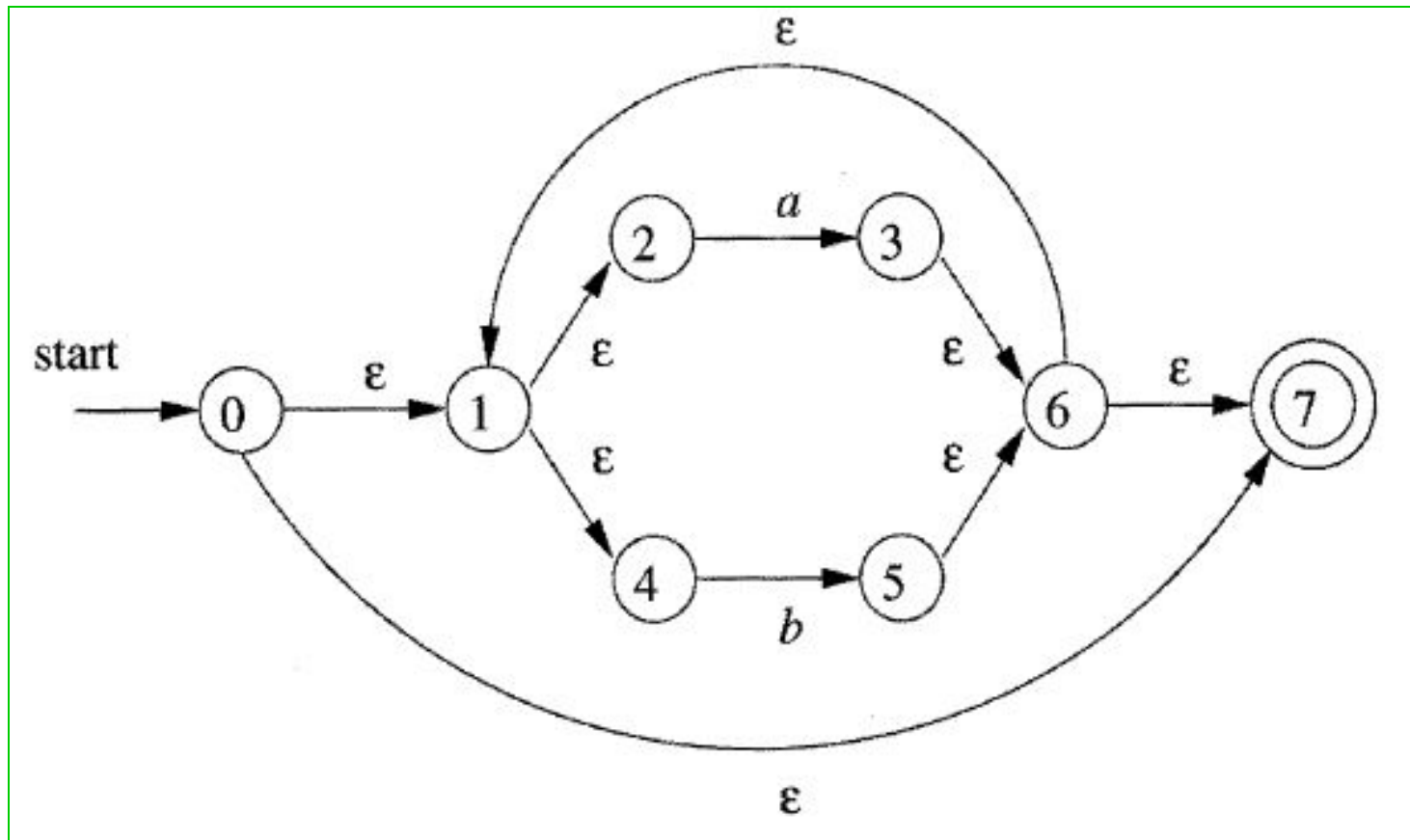


Step 3: For sub expression $r_3 = r_1 / r_2$

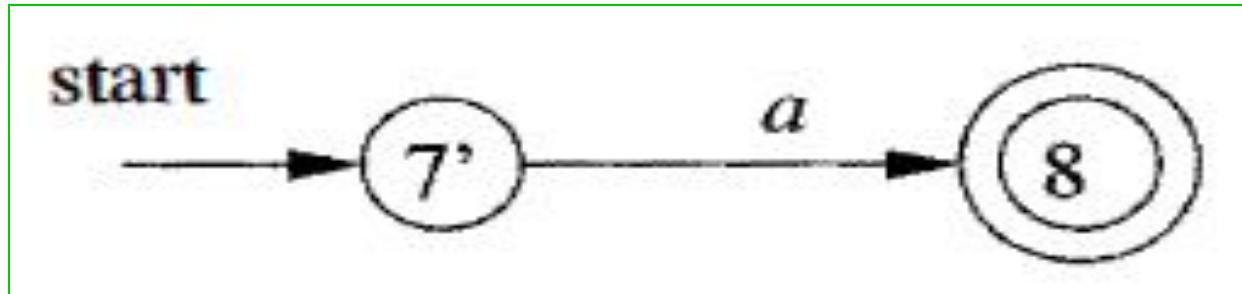


Step 4: For sub expression $r_4 = (r_3)$
Same As r_3

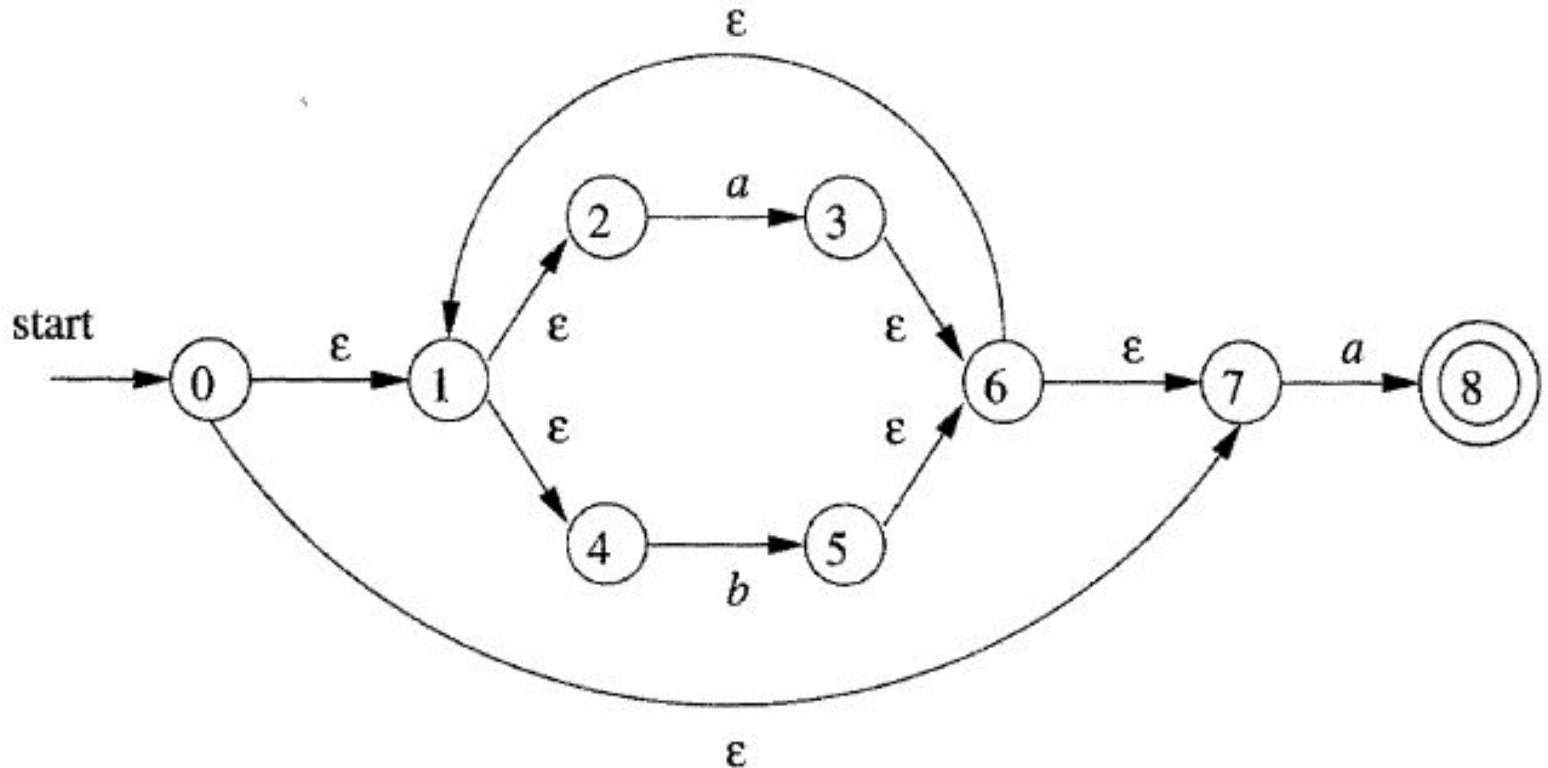
Step 5: For sub expression $r_5 = (r_3)^*$



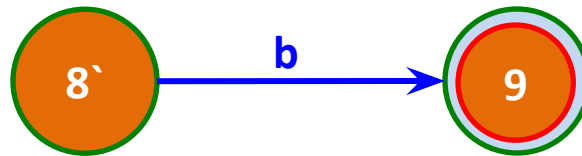
Step 6: For sub expression $r_6 = a$



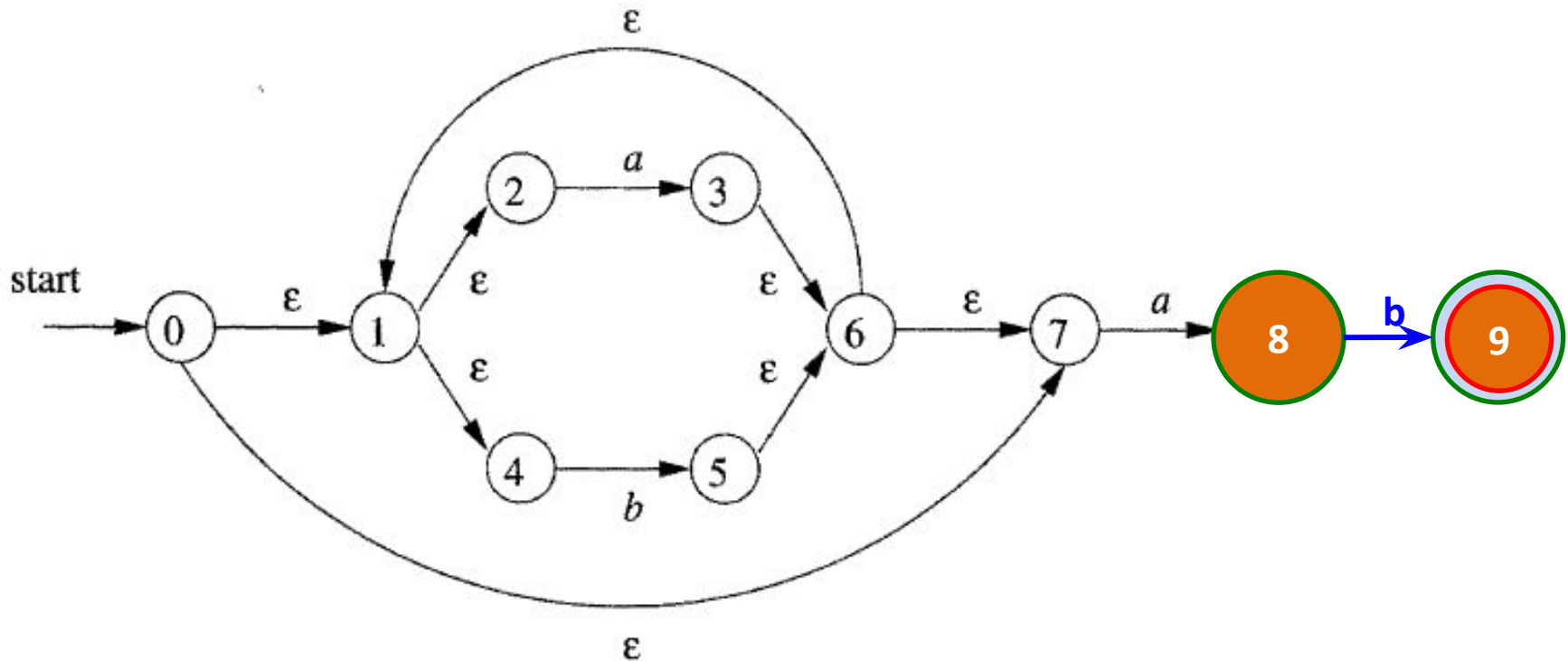
Step 7: For sub expression $r_7 = r_5 r_6$



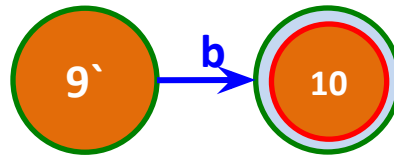
Step 8: For sub expression $r_8 = b$



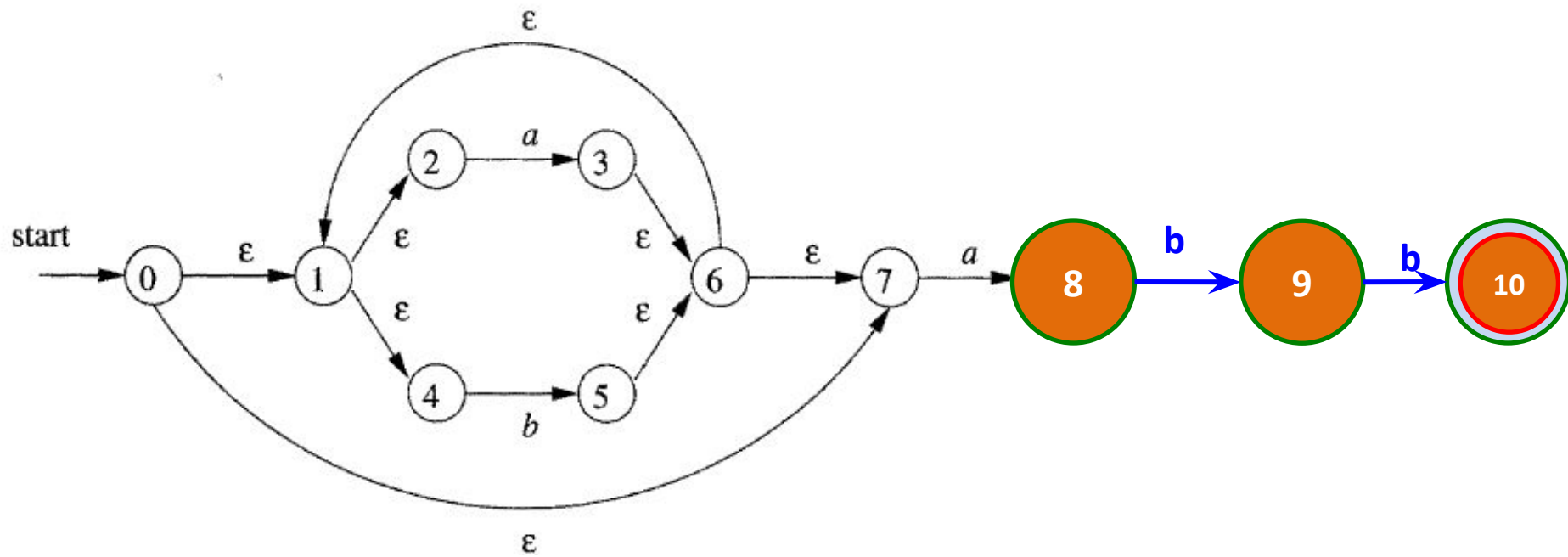
Step 9: For sub expression $r_9 = r_8 r_7$



Step 9: For sub expression $r_{10} = b$



Step 10: For sub expression $r_{11} = r_{10}r_9$



Important States of NFA

- A state of an NFA important if it has a *non- ϵ out-transition*.
- Notice that the subset construction uses only the important states in a **set T** when it computes

ϵ -closure (move(T, a)),

-the set of states reachable from T on **input a**.

- The set of states *move(s, a)* is nonempty only if **state s is important**.
- During the subset construction, two sets of NFA states can be identified (treated as if they were the same set) if they:
 - 1. **Have the same important states**, and
 - 2. **Either both have accepting states or neither does**.

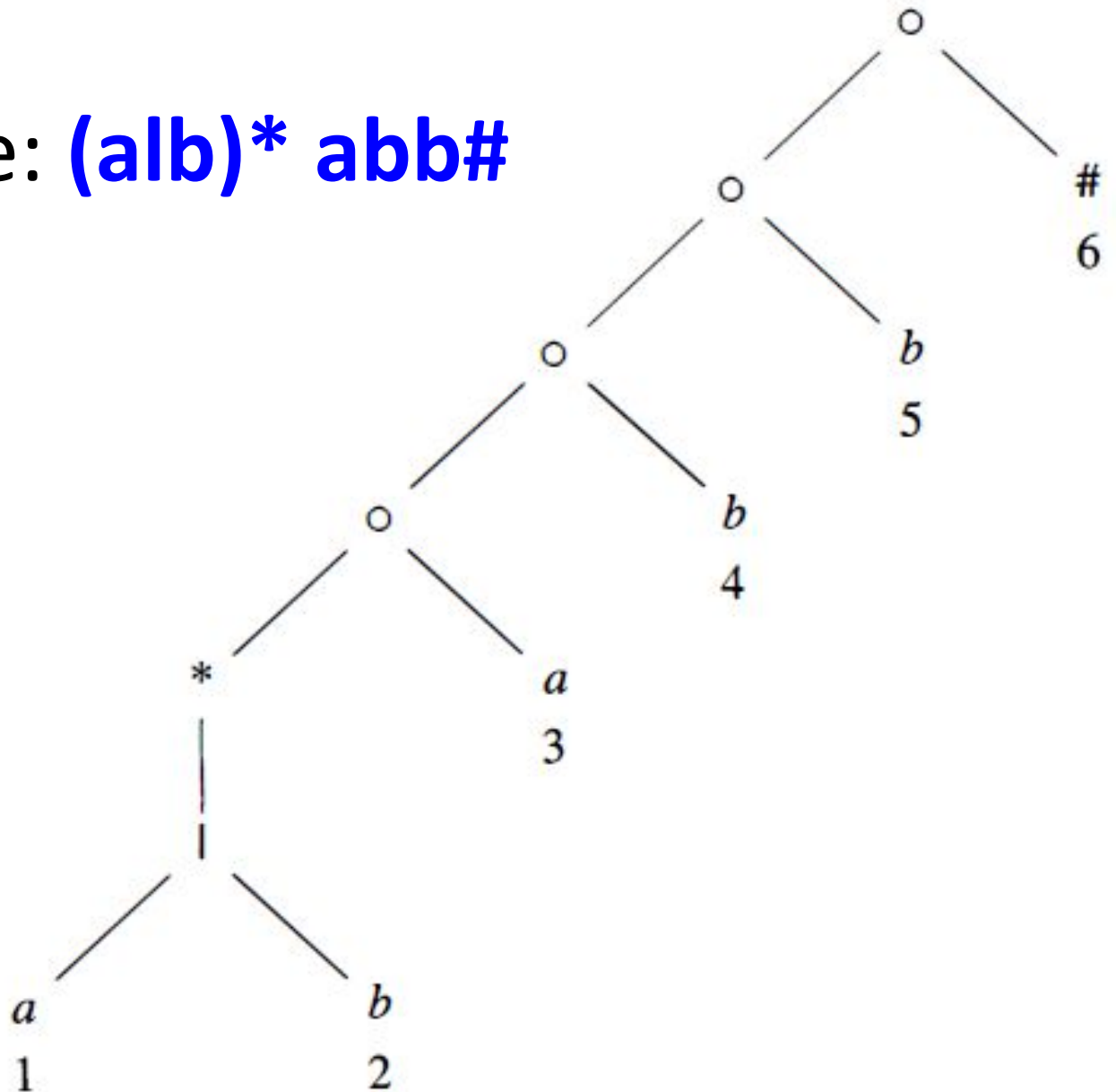
- *The only important states are those introduced as initial states in the basis part for a particular symbol position in the regular expression.*
- Each important state corresponds to a **particular operand in the regular expression**.
- The constructed NFA has **only one accepting state, but this state, having no out-transitions, is not an important state**

- ❑ By concatenating a unique **right end marker #** to a regular expression r , we give the accepting state for r a transition on $\#$, making it an important state of the NFA for $(r) \#$.
- ❑ **Augmented regular expression $(r) \#$,**
- ❑ When the construction is complete, any state with a **transition on $\#$ must be an accepting state**.

Nodes

- The important states of the NFA correspond **directly to the positions in the regular expression that hold symbols**
- present the regular expression by its **syntax tree**
 - leaves correspond to operands
 - interior nodes correspond to operators
- An interior nodes:
 - **cat-node**: concatenation operator (**.**dot)
 - **or-node**: union operator (**|**)
 - **star-node**: star operator (*****)

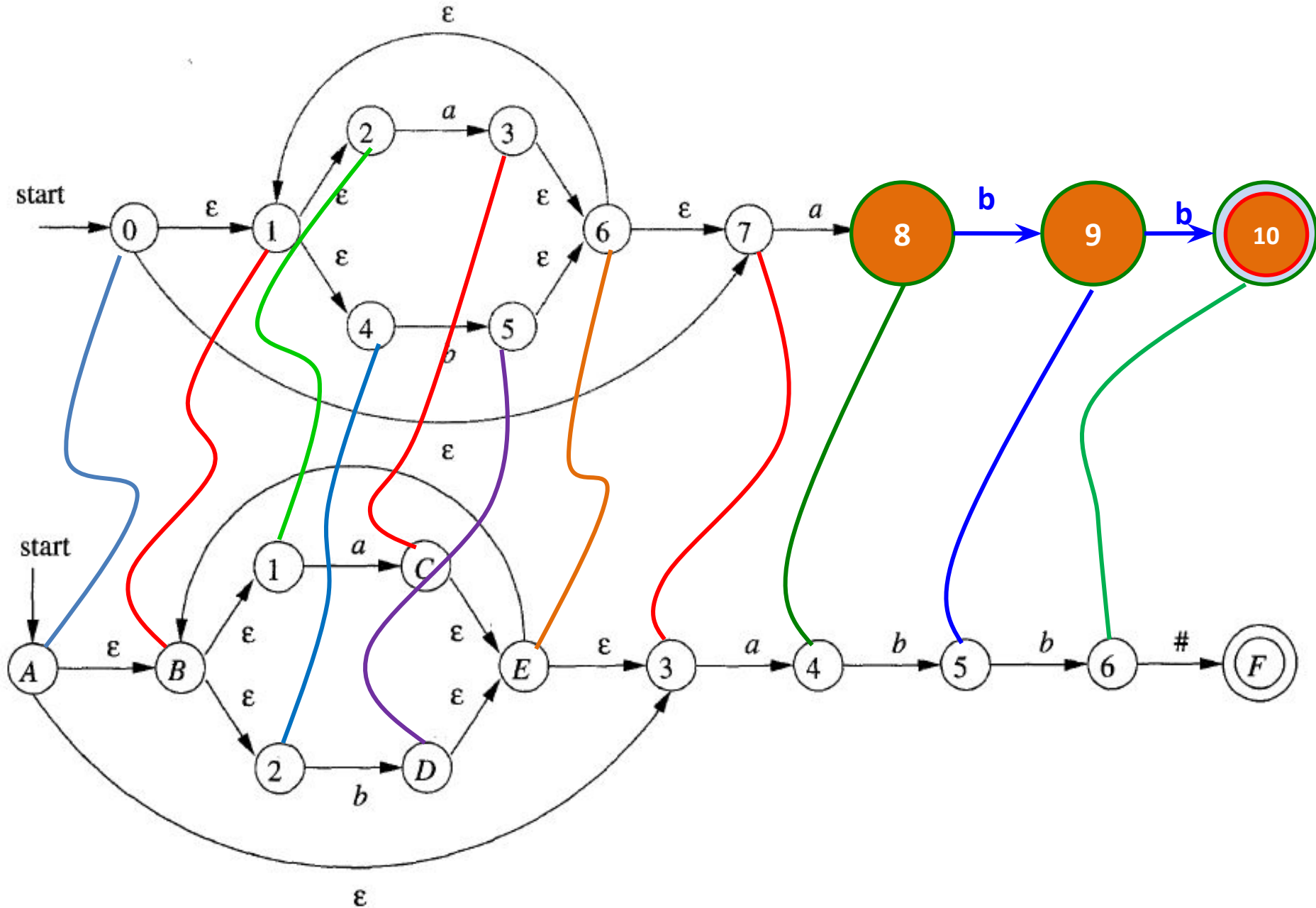
Syntax tree: **(alb)* abb#**



Syntax tree: **(alb)* abb#**

- Leaves in a syntax tree are labeled by ϵ or by an alphabet symbol.
- ❑ To each leaf **not labeled ϵ** , attach a unique integer.
- ❑ (the position of the leaf and also as a position of its symbol)
- ❑ a symbol can have several positions (**a**: 1 & 3)
- The **positions** in the syntax tree correspond to the **important states** of the constructed NFA.

Example: NFA [for $r=(a|b)^*abb\#$] with the important states numbered and other states represented by letters



Functions Computed From the Syntax Tree

- To construct a DFA directly from a regular expression, we construct its syntax tree and then compute *four functions*:

□ *nullable*

□ *firstpos*

□ *lastpos*

□ *followpos*

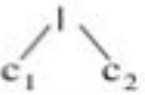


04 Functions

1. ***nullable(n)*** is true for a syntax-tree node n if & only if the sub expression represented by n has ϵ in its language.
 - sub expression can be "made null" or the empty string, even though there may be other strings it can represent as well.
2. ***firstpos(n)*** is the set of positions in the subtree rooted at n that correspond to the **first symbol** of at least one string in the language of the sub expression rooted at n .
3. ***lastpos(n)*** is the set of positions in the subtree rooted at n that correspond to **the last symbol** of at least one string in the language of the sub expression rooted at n .
4. ***followpos(p)***, for a position p , is the set of positions q in the entire syntax tree such that there is some string $x = a_1 a_2 \dots a_n$ in $L(r \#)$ such that for some i , there is a way to explain the membership of x in $L(r \#)$ by **matching a_i to position p** of the syntax tree and **a_{i+1} to position q** .

Computing *nullable*, *firstpos*, & *lastpos*

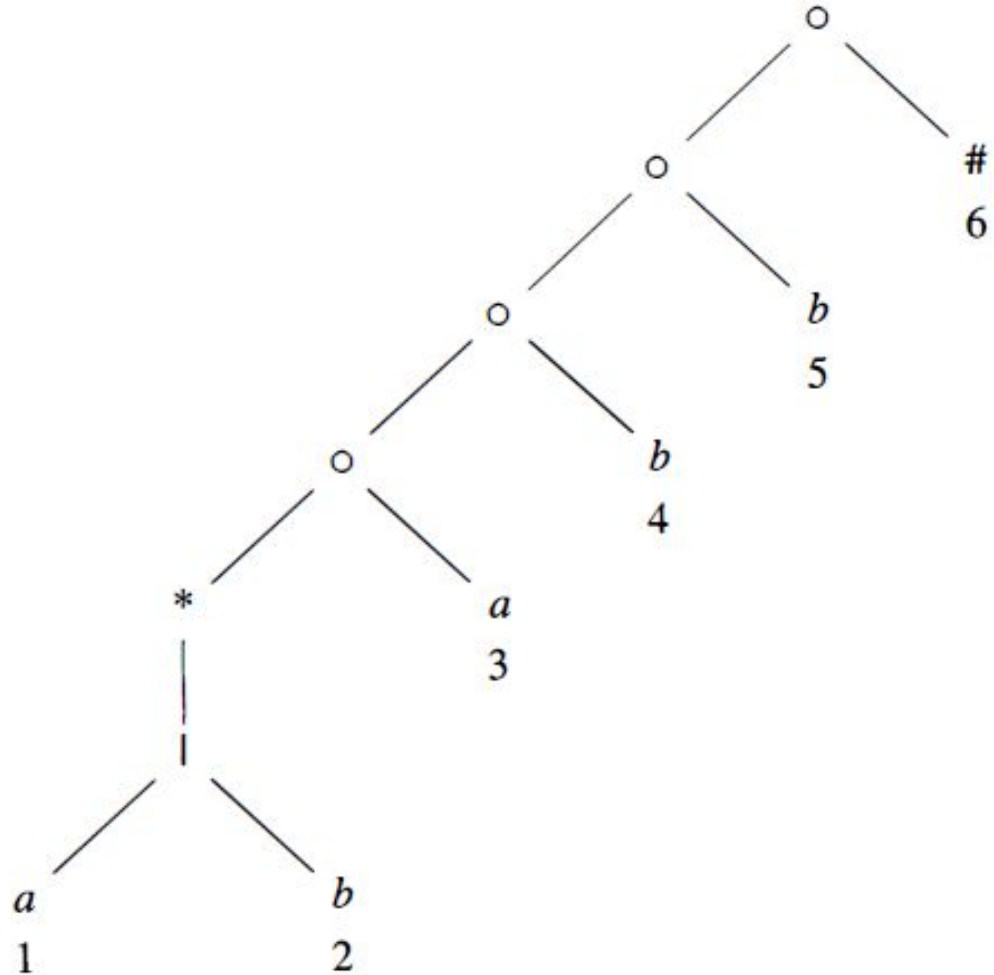
- Compute *nullable*, *firstpos*, & *lastpos* by a straightforward recursion on height of the tree

How to evaluate *firstpos*, *lastpos*, *nullable*

<u>n</u>	<u>nullable(n)</u>	<u>firstpos(n)</u>	<u>lastpos(n)</u>
leaf labeled ϵ	true	Φ	Φ
leaf labeled with position i	false	{i}	{i}
	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	if ($\text{nullable}(c_1)$) $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ else $\text{firstpos}(c_1)$	if ($\text{nullable}(c_2)$) $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$
	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

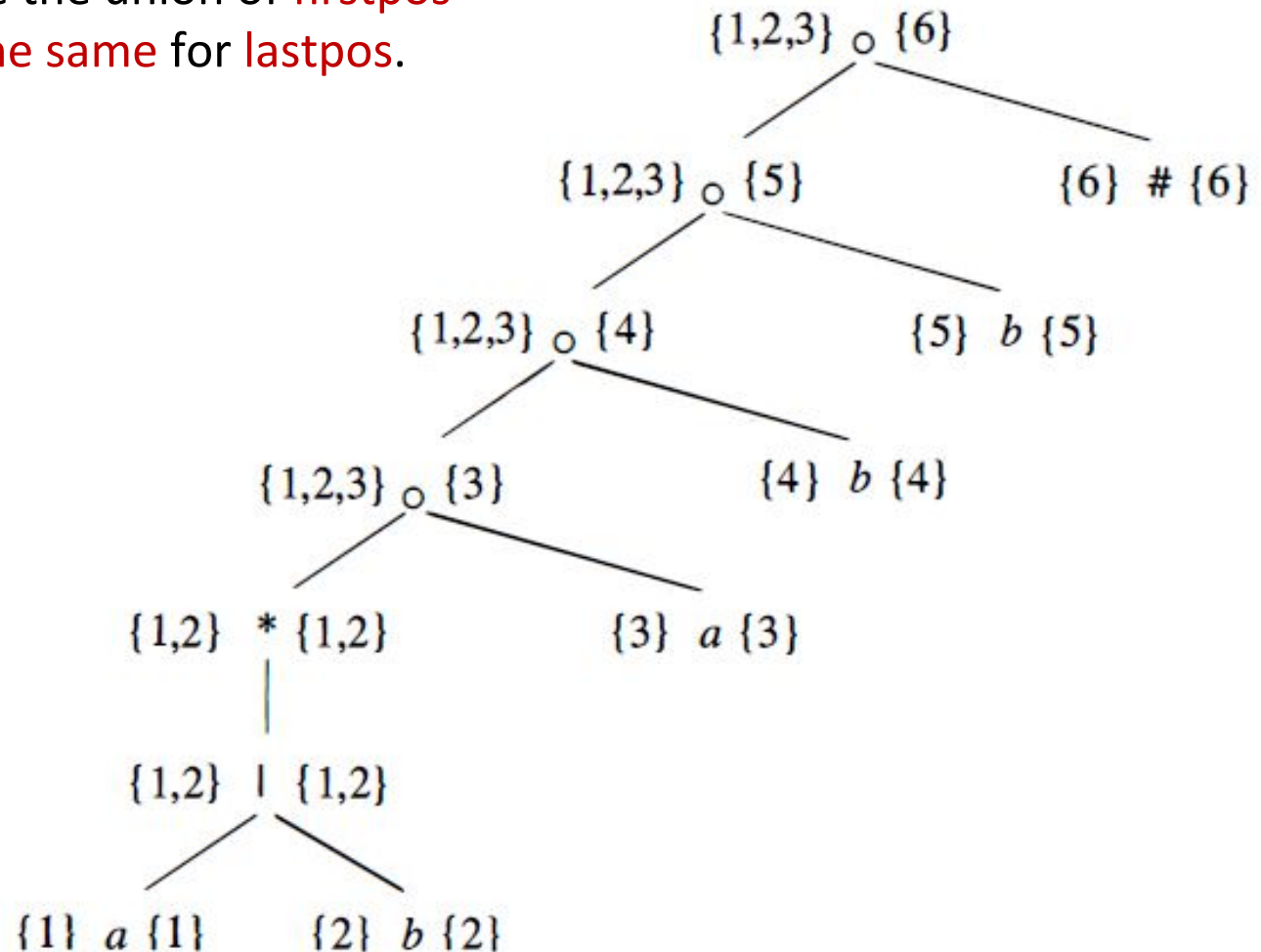
Example : only the **star-node** is **nullable**.

- none of the leaves are nullable, because they each correspond to non- ϵ operands.
- The **or-node** is not **nullable**, because neither of its children is.
- The **star-node** is **nullable**, because every star-node is nullable.
- each of the **cat-nodes**, having at least one non null able child, is not nullable.



▪ *firstpos*(*n*) to the **left** of node *n*, and *lastpos*(*n*) to its **right**.
 Each of the leaves has only itself for *firstpos* & *lastpos*, as required by the rule for non-ε leaves

For the or-node, we take the union of *firstpos* at the children and **do the same** for *lastpos*.



- consider the **lowest cat-node, which we shall call n .**
- To compute **$firstpos(n)$** , we first consider whether the left operand is nullable, which it is in this case.
- Therefore, **$firstpos$ for n** is the **union of $firstpos$ for each of its children**, that is $\{1, 2\} \cup \{3\} = \{1, 2, 3\}$.
- The rule for $lastpos$ are the same as for $firstpos$, with the children interchanged.
- To compute **$lastpos(n)$** we must ask whether its right child (the leaf with position 3) is **nullable, which it is not**.
- Therefore, **$lastpos(n)$** is the **same as $lastpos$ of the right child, or $\{3\}$.**

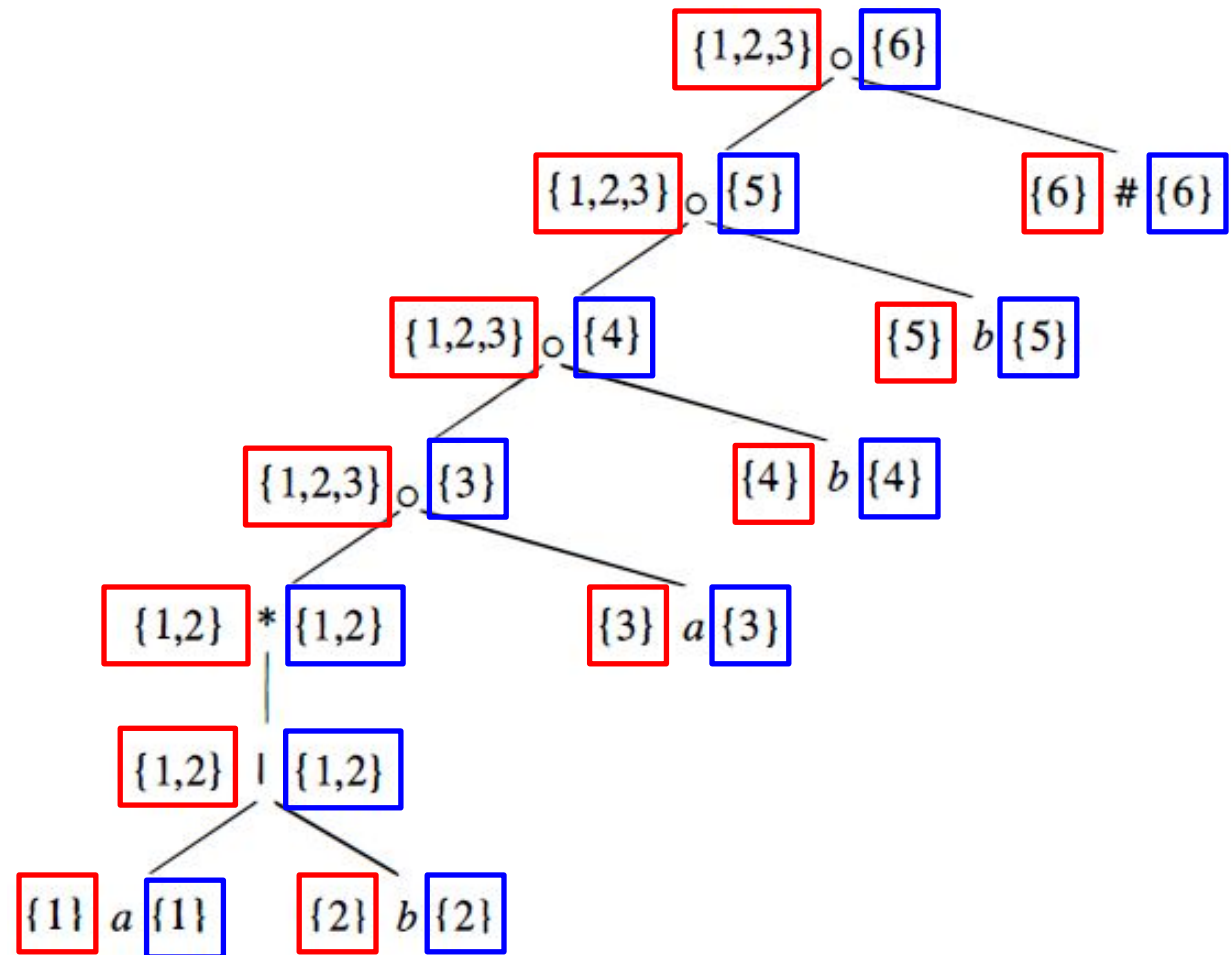
Computing *Followpos*

- two ways that a position of a regular expression can be made to follow another:
 1. If n is a cat-node with left child C_1 & right child C_2 , then for every position i in $\text{lastpos}(C_1)$, all positions in $\text{firstpos}(C_2)$ are in $\text{followpos}(i)$.
 2. If n is a star-node, & i is a position in $\text{lastpos}(n)$, then all positions in $\text{firstpos}(n)$ are in $\text{followpos}(i)$.

Example: Rule 1 for *followpos* requires that we look at each *cat-node*, & put each position in *firstpos* of its *right child* in *followpos* for each position in *lastpos* of its left child.

firstpos

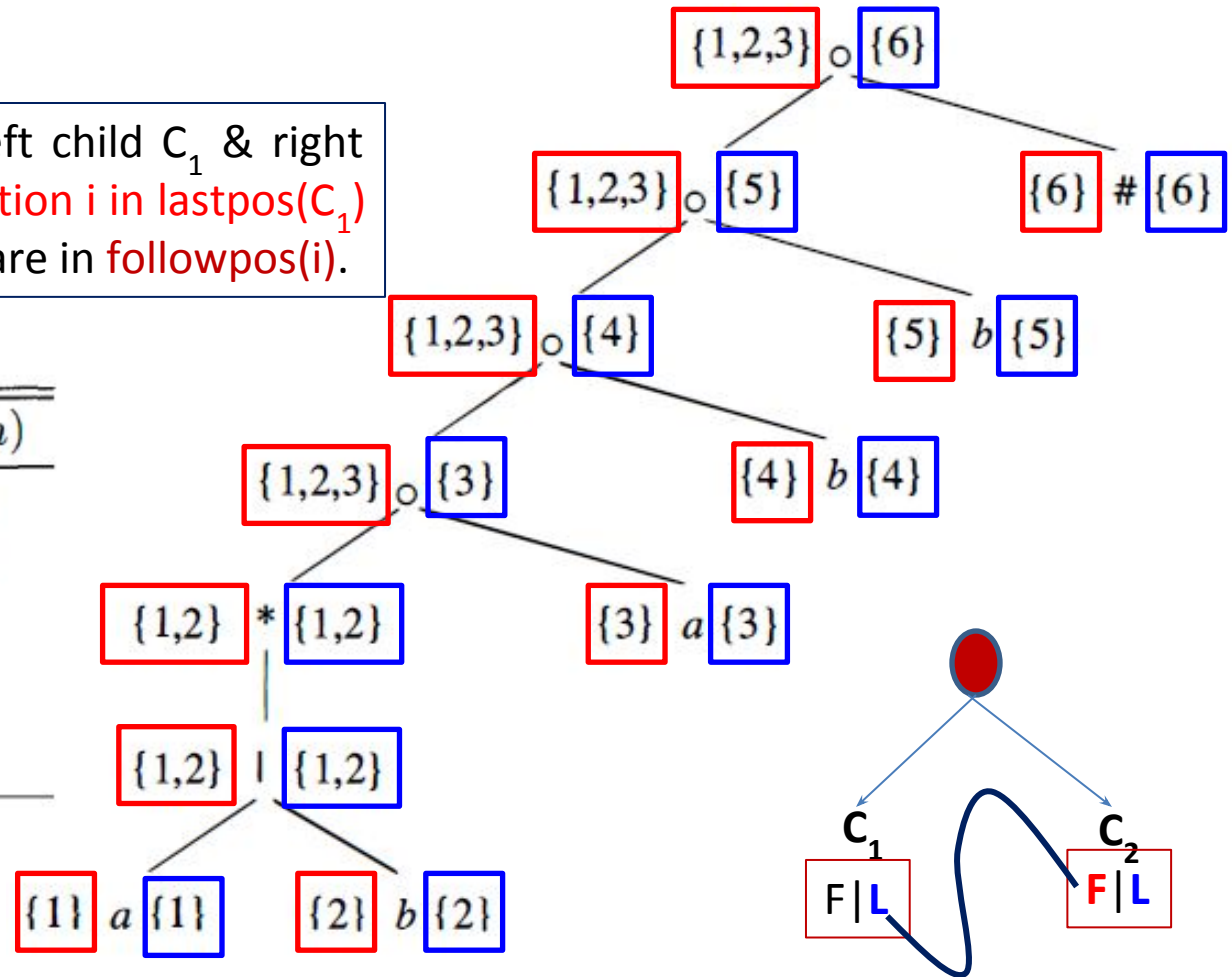
lastpos



- For the **lowest cat-node**, that rule says **position 3** is in **followpos(1)** and **followpos(2)**
- The next cat-node says that **4** is in **followpos (3)**,
- remaining two cat-nodes give us **5** in **followpos (4)** & **6** in **followpos(5)**

1. If n is a cat-node with left child C_1 & right child C_2 , then for **every position i in $\text{lastpos}(C_1)$** , **all positions in $\text{firstpos}(C_2)$** are in **followpos(i)**.

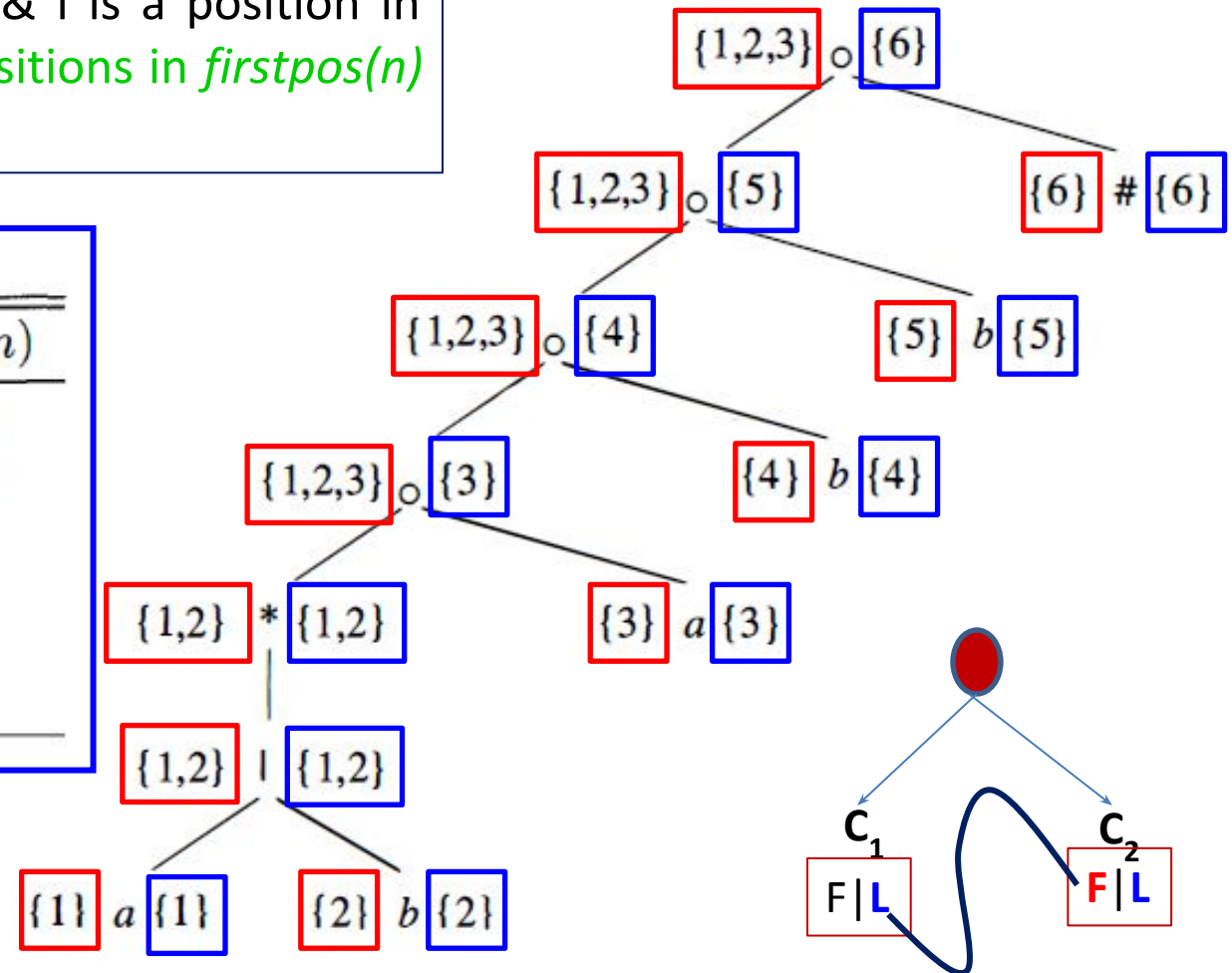
NODE	n	$\text{followpos}(n)$
1		$\{1, 2, 3\}$
2		$\{1, 2, 3\}$
3		$\{4\}$
4		$\{5\}$
5		$\{6\}$
6		\emptyset



- For the **lowest cat-node**, that rule says **position 3** is in **followpos(1)** & **followpos(2)**
- Rule 2 to the star-node. **positions 1 & 2** are in both **followpos(1)** & **followpos(2)**, since both **firstpos** & **lastpos** for this node are **{1, 2}**.

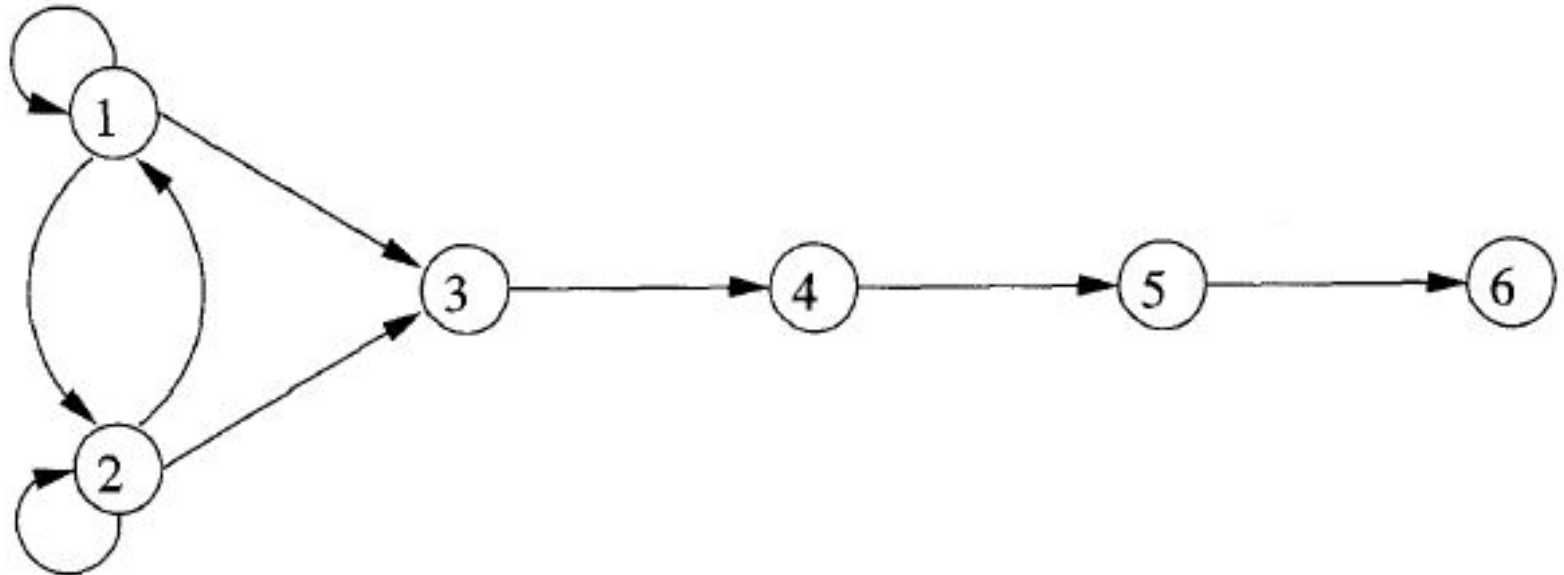
2. If **n** is a **star-node**, & **i** is a position in **lastpos(n)**, then **all positions in firstpos(n)** are in **followpos(i)**.

NODE	<i>n</i>	<i>followpos(n)</i>
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



Directed graph for the function *followpos*

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	\emptyset



Converting a Regular Expression Directly to a DFA

Algorithm: Construction of a DFA from a regular expression r .

INPUT : A regular expression r .

OUTPUT: A DFA D that recognizes $L(r)$.

METHOD:

- 1 . Construct a **syntax tree T** from the augmented regular expression $(r) \#$.
 2. Compute *nullable, firstpos, lastpos, & followpos for T*
 3. Construct **Dstates**, the set of states of DFA D , & **Dtran**, the transition function for D . The states of D are sets of positions in T . Initially, each state is "**unmarked**," & a state becomes "marked" just before we consider its out-transitions.
- The start state of D is $\text{firstpos}(n_0)$, where node n_0 is the root of T .
 - The **accepting states** are those containing the position for **endmarker symbol $\#$**

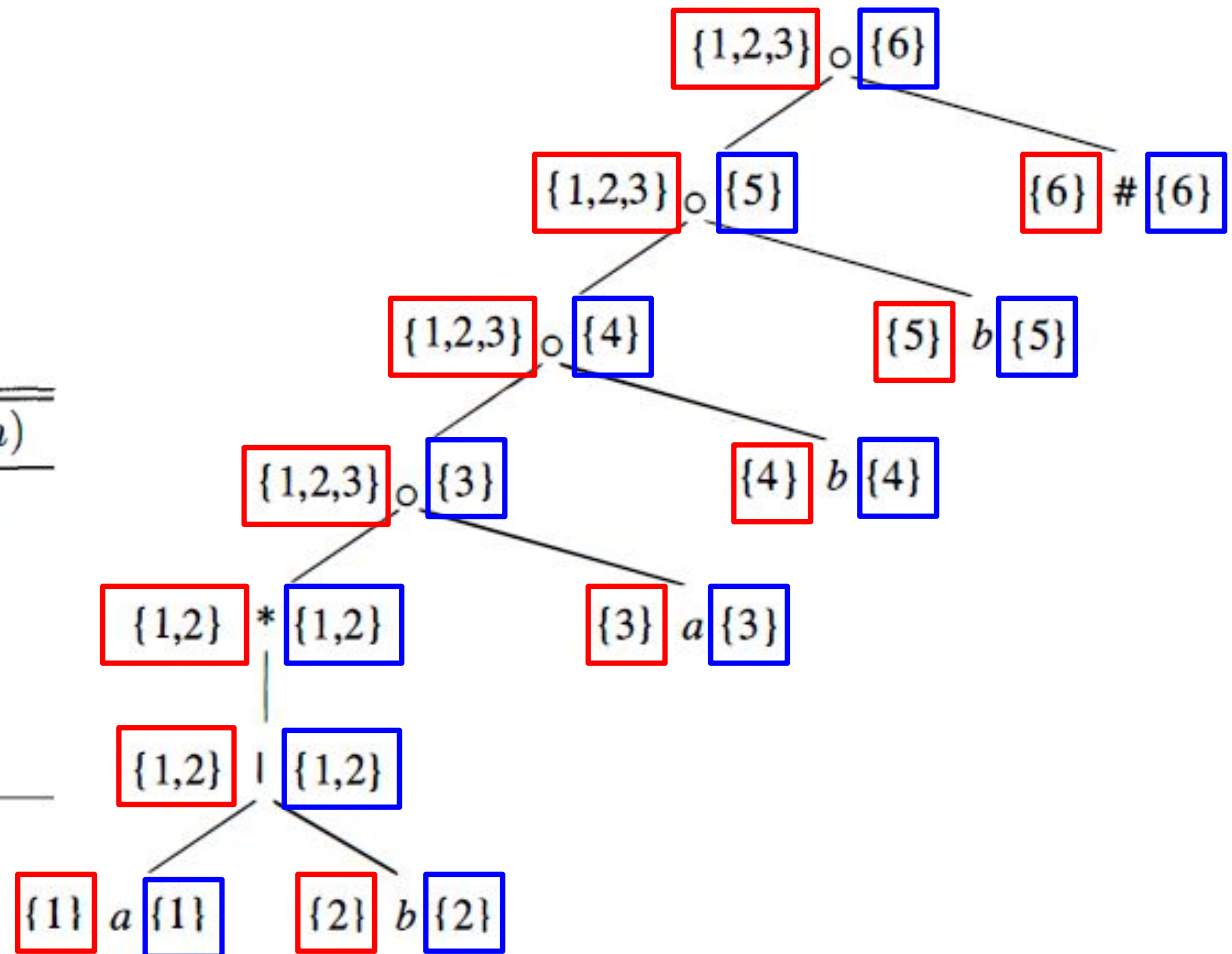
Construction of a DFA directly from a regular expression

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

Example: construct a DFA for the regular expression $r = (a|b)^*abb$.

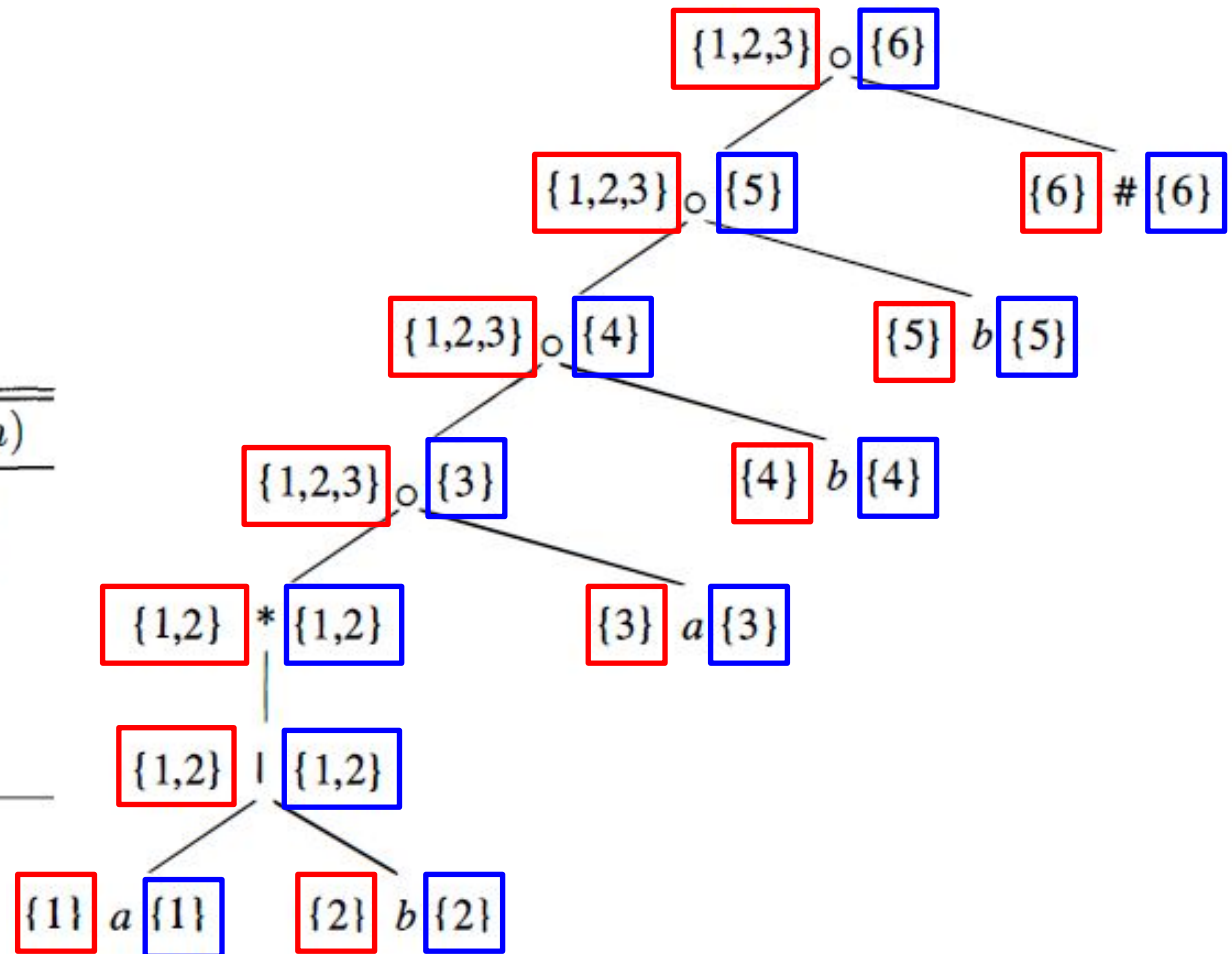
- The value of *firstpos* for the root of the tree: {1, 2, 3}
- A = {1, 2, 3} ----Start state**

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	\emptyset



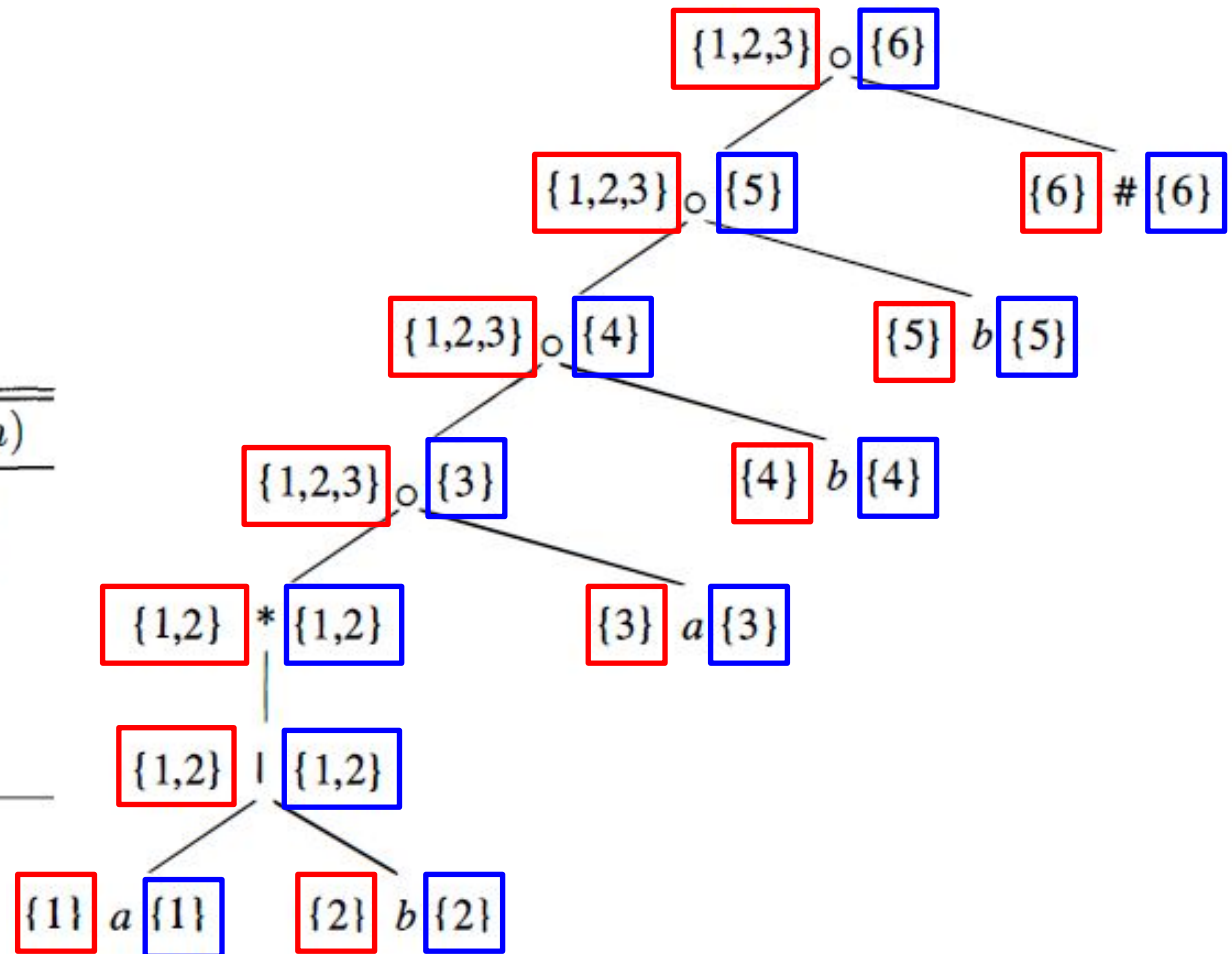
- Compute **Dtran[A, a] & Dtran[A, b]**.
- Among the positions of A, **1 & 3** correspond to **a**, while **2** corresponds to **b**.
- **Dtran[A, a] = followpos(1) U followpos(3) = {1, 2, 3, 4} = B**

NODE	<i>n</i>	<i>followpos</i> (<i>n</i>)
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		∅



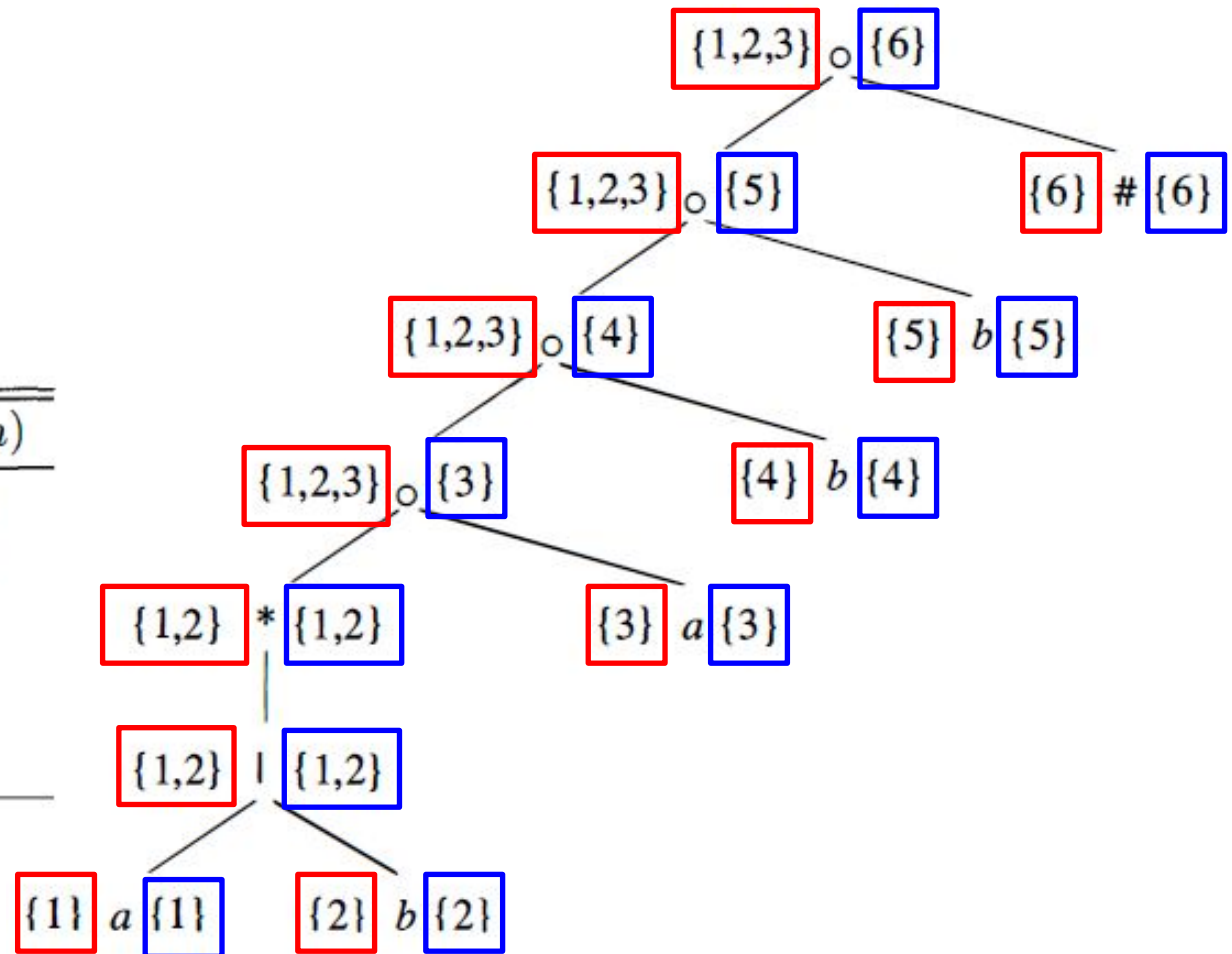
- Compute **Dtran[A, b]**.
- Among the positions only **2** corresponds to **b**.
- **Dtran[A, b] = followpos(2) = {1, 2, 3} = A**

NODE	n	$followpos(n)$
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



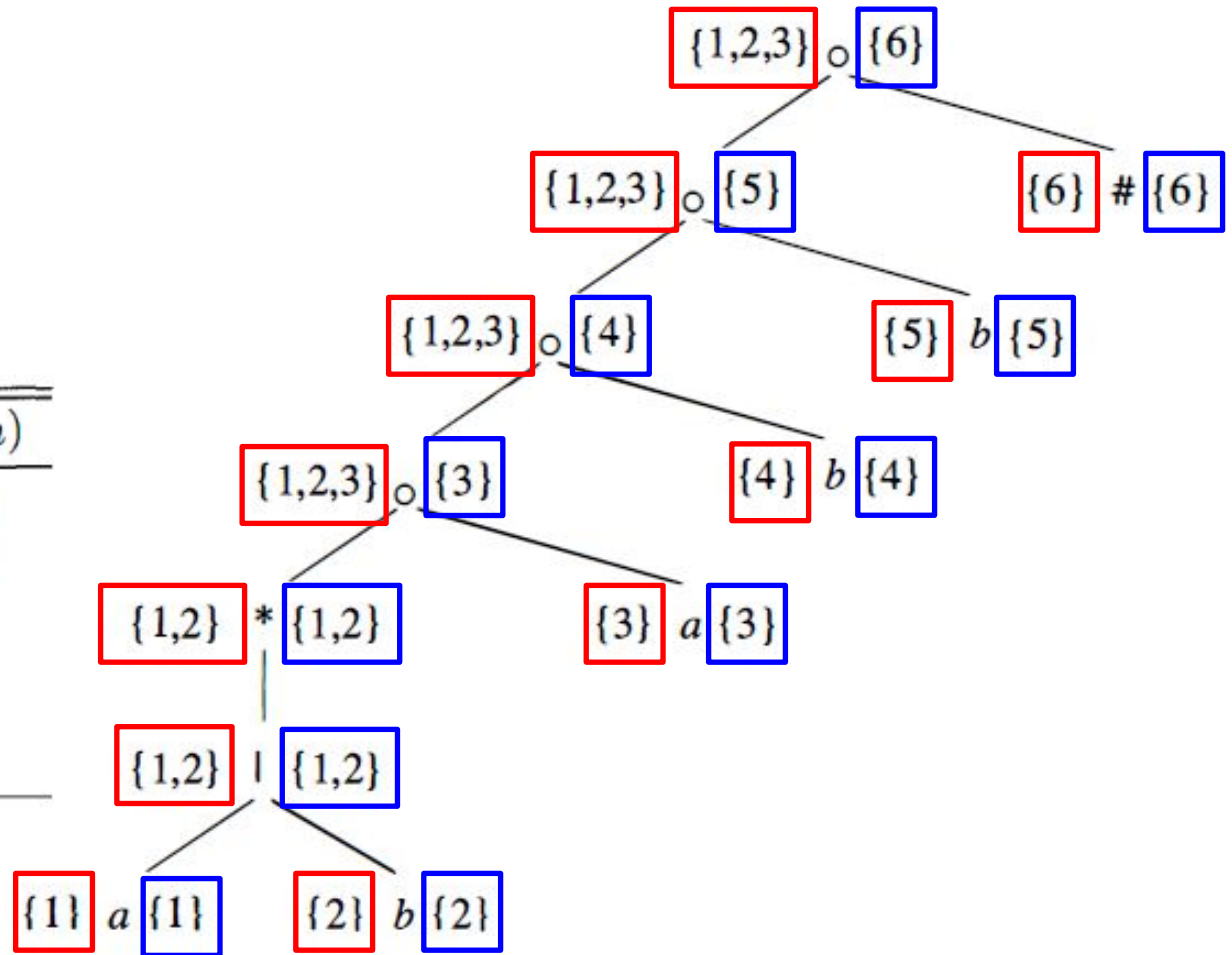
- Compute **Dtran[B, a]** = Dtran[{1, 2, 3, 4}, a]
- Among the positions 1, 3 **corresponds to a**.
- **Dtran[B, a] = followpos(1) U followpos(3)**
= {1, 2, 3, 4} = B

NODE	<i>n</i>	<i>followpos(n)</i>
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



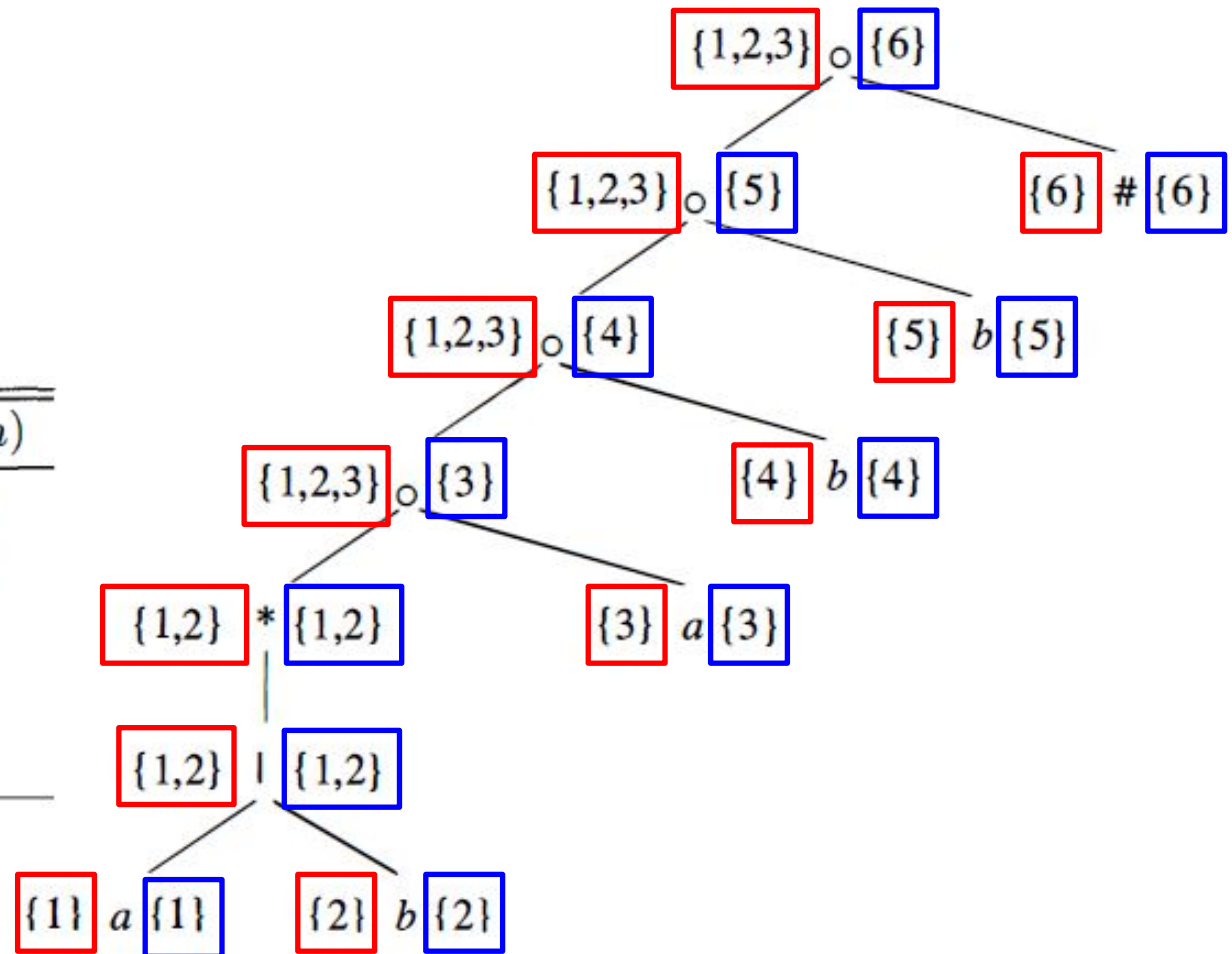
- $\text{Dtran}[B, b] = \text{followpos}(2) \cup \text{followpos}(4)$
 $= \{1, 2, 3, 5\} = C$

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	\emptyset



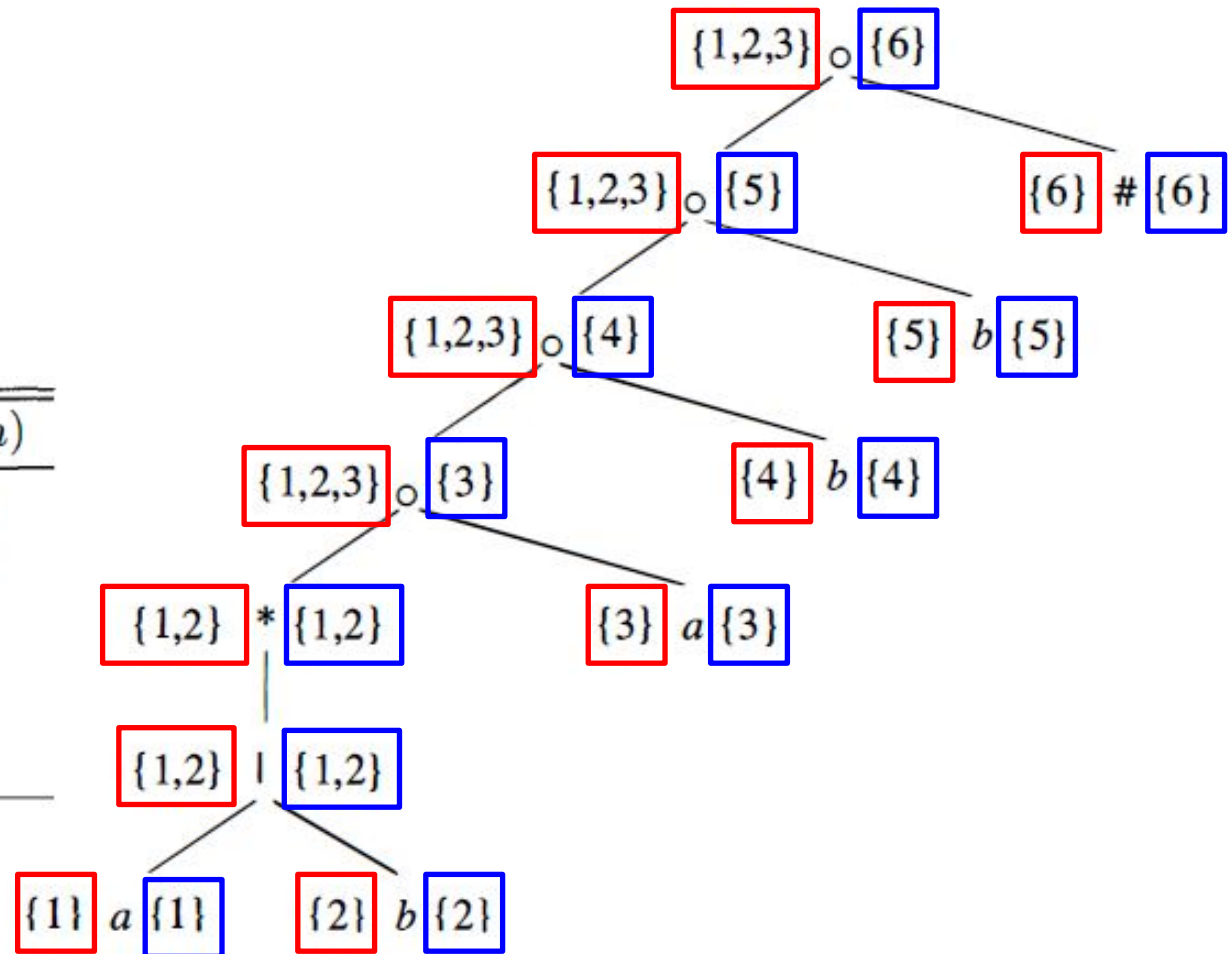
- Compute **Dtran[C, a]** = Dtran[{1, 2, 3, 5}, a]
- Among the positions 1 & 3 **corresponds to a.**
- **Dtran[C, a] = followpos(1) U followpos(3)**
= {1, 2, 3, 4} = B

NODE	<i>n</i>	<i>followpos(n)</i>
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



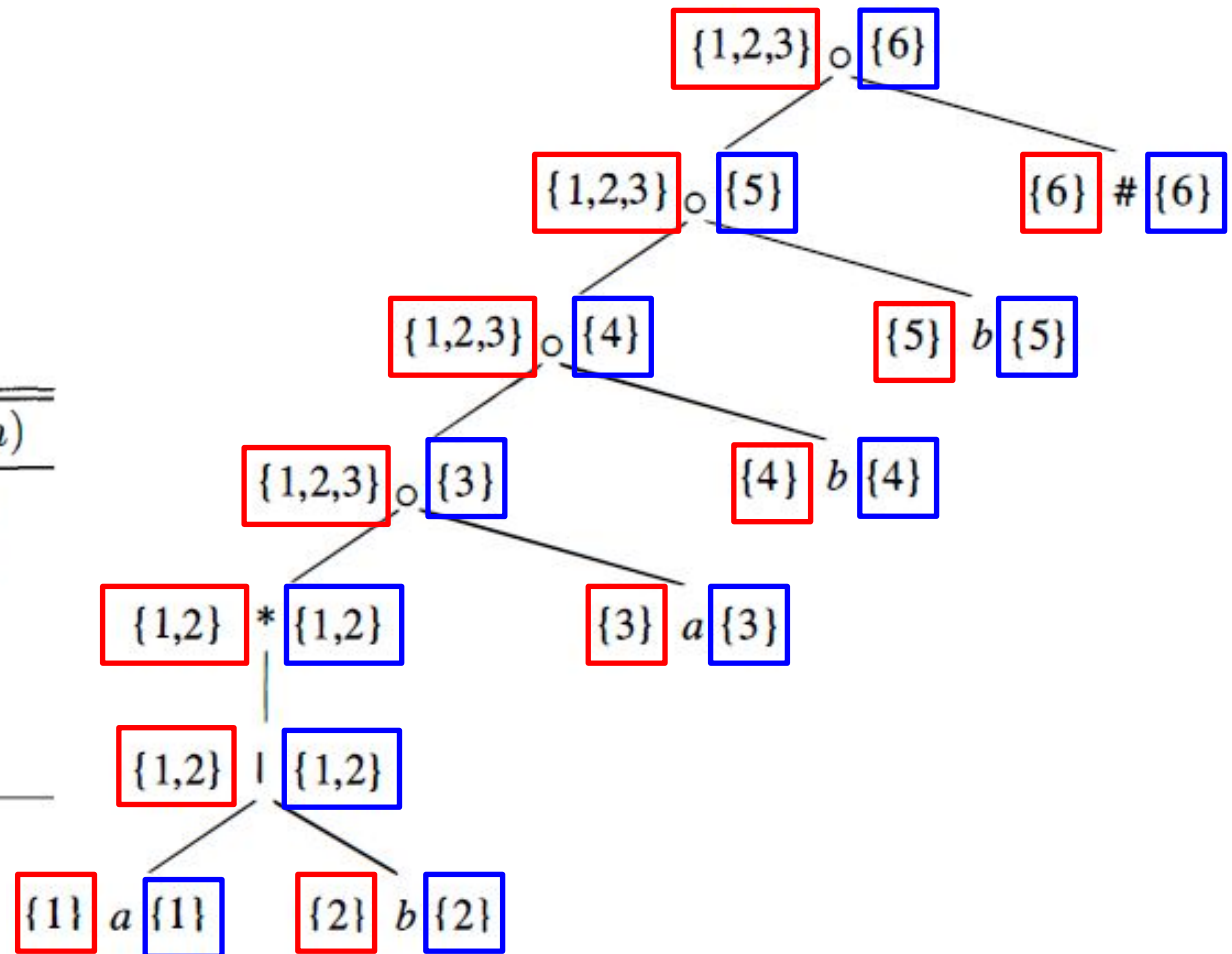
- Compute **Dtran[C, b]** = Dtran[{1, 2, 3, 5}, b]
- Among the positions 2 & 5 **corresponds to b**.
- **Dtran[C, b] = followpos(2) U followpos(5)**
= {1, 2, 3, 6} = D

NODE	<i>n</i>	<i>followpos(n)</i>
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



- Compute **Dtran[D, a]** = Dtran[{1, 2, 3, 6}, a]
- Among the positions 1 & 3 **corresponds to a**.
- **Dtran[D, a] = followpos(1) U followpos(3)**
= {1, 2, 3, 4} = B

NODE	n	$followpos(n)$
1		{1, 2, 3}
2		{1, 2, 3}
3		{4}
4		{5}
5		{6}
6		\emptyset



$$A = \{1, 2, 3\}$$

$$\text{Dtran}[A, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} = B$$

$$\text{Dtran}[A, b] = \text{followpos}(2) = \{1, 2, 3\} = A$$

$$\text{Dtran}[B, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} = B$$

$$\text{Dtran}[B, b] = \text{followpos}(2) \cup \text{followpos}(4) = \{1, 2, 3, 5\} = C$$

$$\text{Dtran}[C, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} = B$$

$$\text{Dtran}[C, b] = \text{followpos}(2) \cup \text{followpos}(5) = \{1, 2, 3, 6\} = D$$

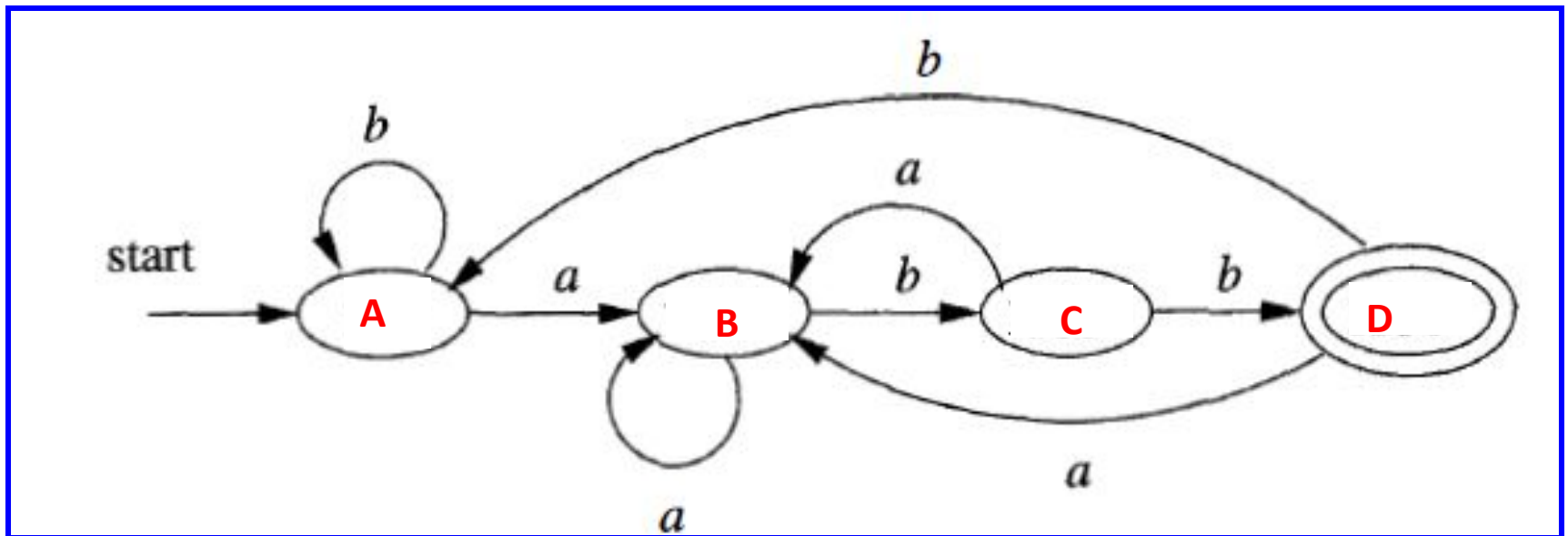
$$\text{Dtran}[D, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} = B$$

$$\text{Dtran}[D, b] = \text{followpos}(2) = \{1, 2, 3\} = A$$

States		a	b
$\{1, 2, 3\}$	A	B	A
$\{1, 2, 3, 4\}$	B	B	C
$\{1, 2, 3, 5\}$	C	B	D
$\{1, 2, 3, 6\}$	D	B	A

States		a	b
{1, 2, 3}	A	B	A
{1, 2, 3, 4}	B	B	C
{1, 2, 3, 5}	C	B	D
{1, 2, 3, 6}	D	B	A

DFA Construction



Assignment

1. $(a|b)^*a(a|b)$

2. $(a|b)^*a(a|b)(a|b)$

3. $(a|b)^*a(a|b)(a|b)(a|b)$

Conclusion

- Tokens
- Lexemes
- Patterns
- Regular Expressions
- Regular Definitions
- Transition Diagrams
- Finite Automata
- DFA & NFA
- Conversion (NFA to DFA, Regular Expression to NFA/DFA)