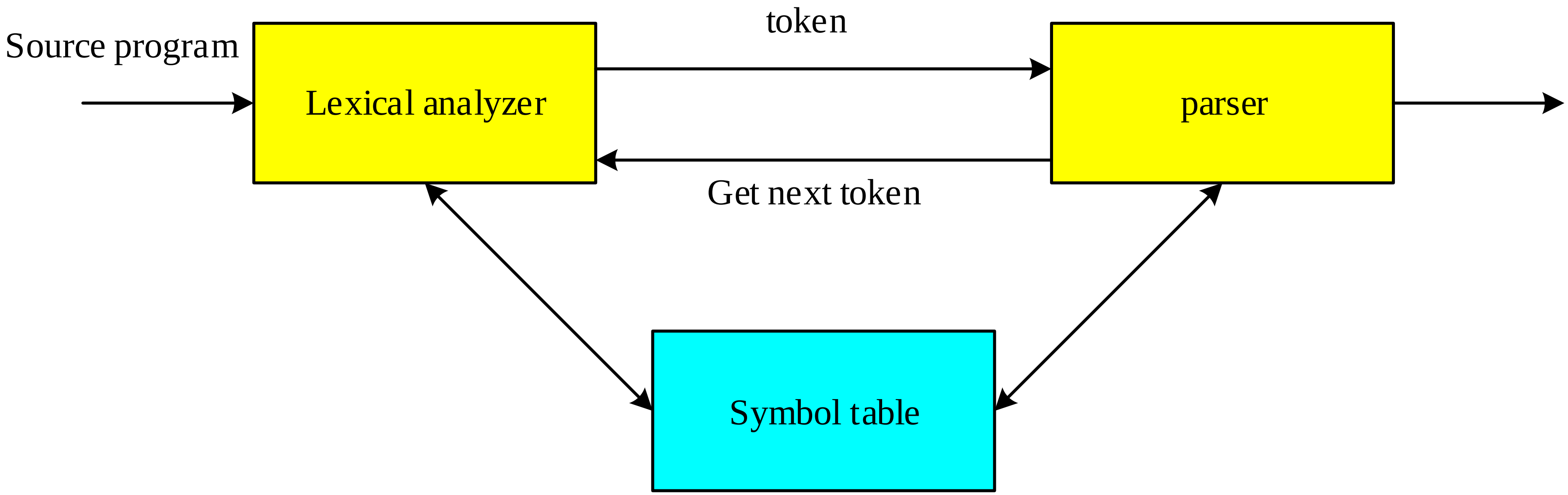
Lexical Analysis

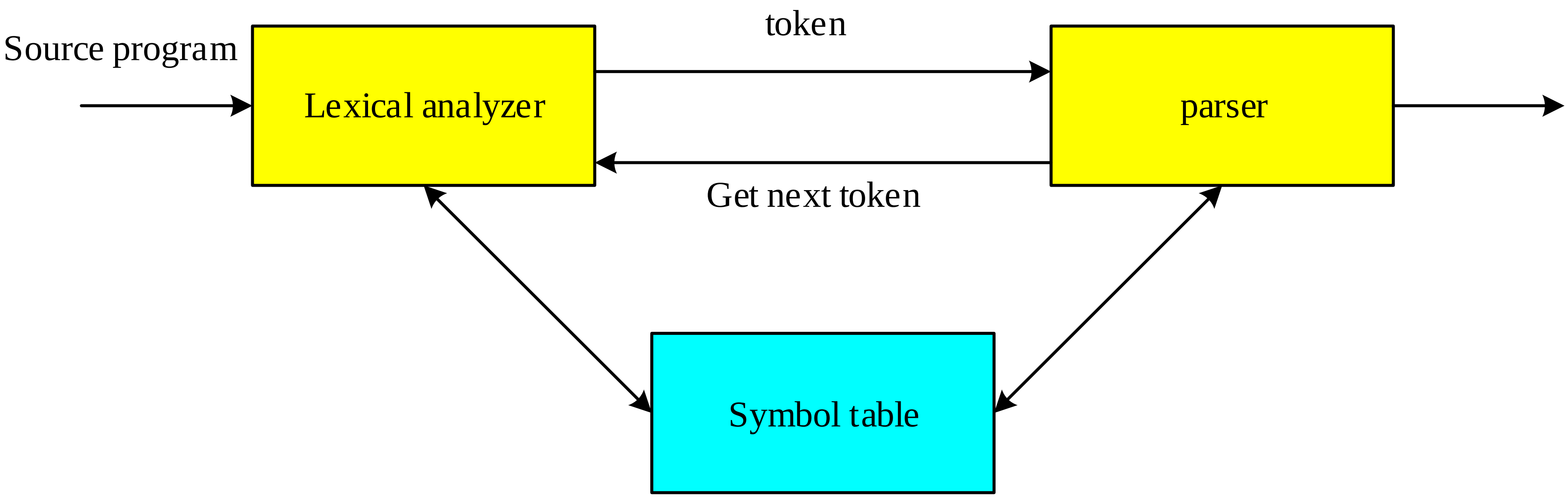
Lexical Analysis



- What does a Lexical Analyzer do?
  - Partition input string into substrings
  - Where the substrings are tokens

- How does it Work?

Lexical Analysis



- Goal: Partition input string into substrings
  - Where the substrings are tokens
- What do we want to do? Example:

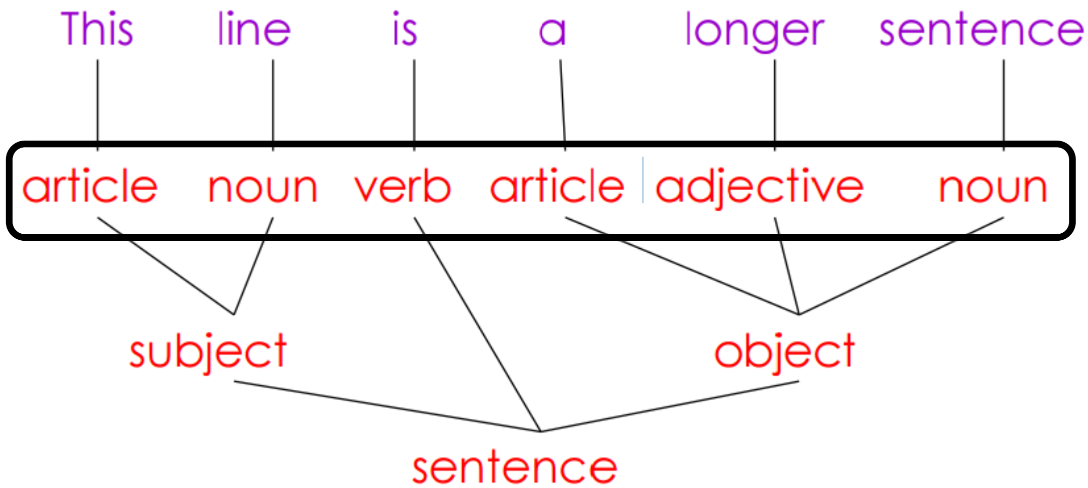
```
if (x == y)
    i = 1;
else
    i = 0;
```

- The input is just a string of characters:

\tif (x == y)\n\t\ti = 1;\n\telse\n\t\ti = 0;

What is a token?

- A syntactic category
  - In English: noun, verb, adjective, …



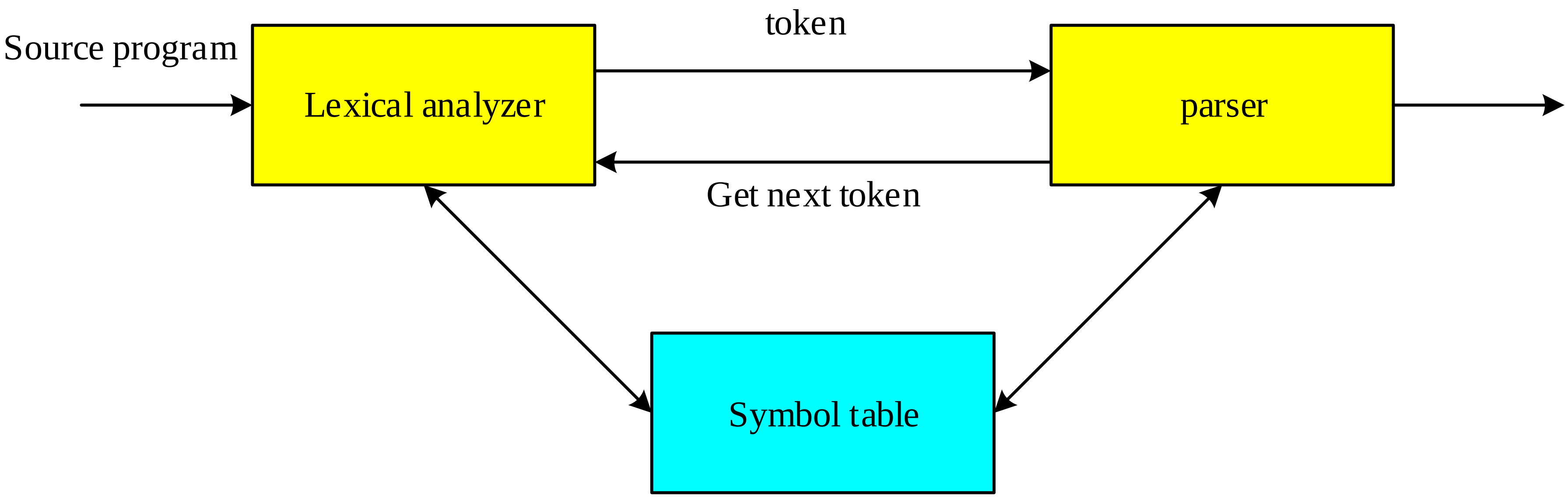

```
if (x == y)
        i = 1;
else
        i = 0;
```

  - In a programming language: Identifier, Integer, Keyword, Whitespace, …

\tif (x == y)\n\t\ti = 1;\n\telse\n\t\ti =0;

- Identifier: x, y, i (strings of letters or digits, starting with a letter)
- Keyword: if, else (strings of letters)
- Integer: 0,1 (string of digits)
- Whitespace: \t,\n
- delimiters: ; (,)

What are Tokens For?



- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens, which is input to the parser
- Parser relies on token distinctions
  – An identifier is treated differently than a keyword

\tif (x == y)\n\t\ti = 1;\n\telse\n\t\ti = 0;

WSIF(ID==ID)WSWSWSID=NUM;WSWSELSEWSWSWSID=NUM;

WS=Whitespace

Token, pattern, lexemes

- Token is a logical unit in the scanner.
- A lexeme is an instance of token.

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
string	"core dumped"	any characters between " and " except "

Classifies
Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Attributes for Tokens

- An attribute of the token : any value associated to a token
- The lexical analyzer collects information about tokens into their associated attributes.
- The tokens influence parsing decisions

\tif (x == y)\n\t\ti = 1;\n\telse\n\t\ti =0;

WSIF(ID==ID)WSWSWSID=NUM;WSWSELSEWSWSWSID=NUM;

(ID,x) (NUM,1)

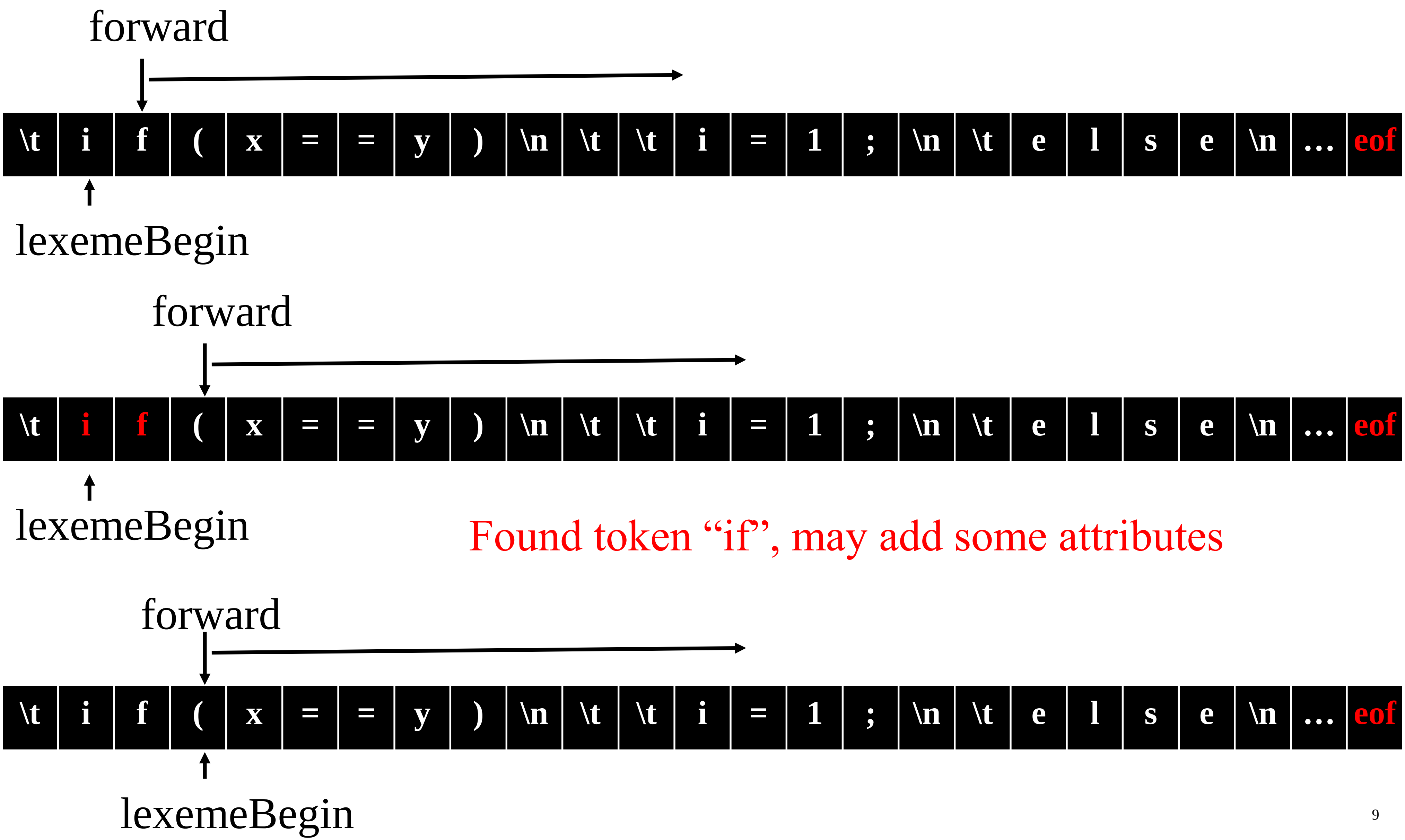
Token representation

- A token record :

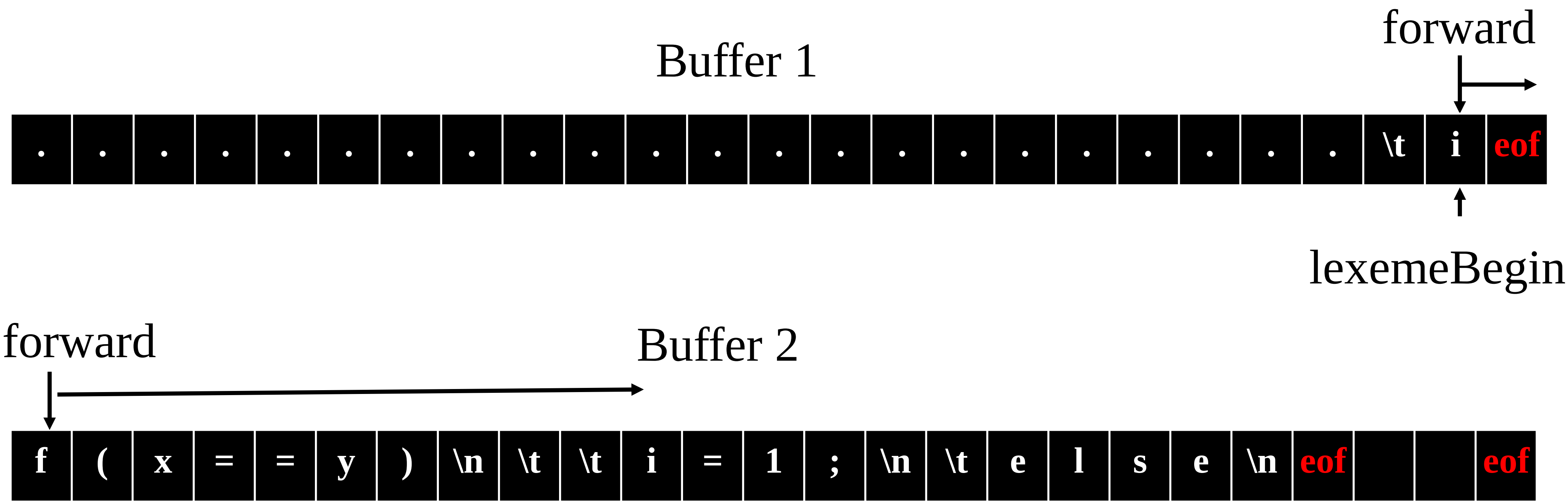
```
Typedef  struct
   {  TokenType  tokenval;
      char *stringval;
      int numval;
   } TokenRecord
```

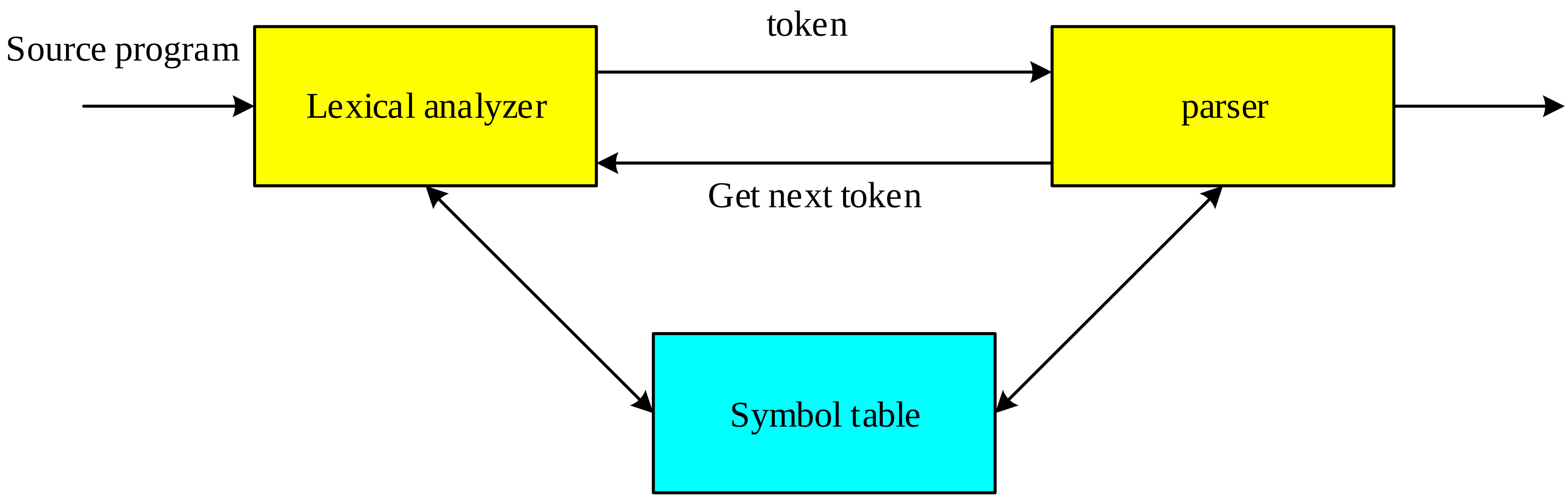
- A more common arrangement: the scanner return the token value only and place the other attributes in variables (such as in LEX/Flex and YACC/Bison) in symbol table.
- The string of input characters is kept in a **buffer** or provided by the system input facilities.

Scanning process



Scanning process: two buffers





What are responsibilities of each box ?

Lexical Analyzer in Perspective

- LEXICAL ANALYZER
  – Scan Input
  – Remove WS, NL, …
  – Identify Tokens
  – Create Symbol Table
  – Insert Tokens into ST
  – Generate Errors
  – Send Tokens to Parser

- PARSER
  – Perform Syntax Analysis
  – Actions Dictated by Token Order
  – Update Symbol Table Entries
  – Create Abstract Rep. of Source
  – Generate Errors
  – And More…. (We'll see later)

Issues in lexical analysis

- Separation of Lexical Analysis From Parsing Presents a **Simpler Conceptual Model**
  - From a **Software Engineering Perspective** Division Emphasizes
    - High **Cohesion** and Low **Coupling**
    - Implies Well Specified $\Rightarrow$ **Parallel** Implementation
- Separation Increases Compiler **Efficiency** (I/O Techniques to Enhance Lexical Analysis)
- Separation Promotes **Portability**.
  - This is critical today, when platforms (OSs and Hardware) are numerous and varied!

Design of a Lexical Analyzer

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser
- Describe which strings belong to each token (using patterns)
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: if, else, while, for, …,
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical analyzer = scanning + lexical analysis

Specification of tokens

Language Concepts :

A language, L, is simply any set of strings over a fixed alphabet.

Alphabet	Languages
{0,1}	{0, 10, 100, 1000, 001000…}
	{0, 1, 00, 11, 000, 111,…}
{a,b,c}	{abc, aabbcc, aaabbbccc,…}
{A, … ,Z}	{TEE, FORE, BALL,…}
	{FOR, WHILE, GOTO,…}
{A,…,Z,a,…,z,0,…9,	{ All legal PASCAL progs}
+,-,…,<,>,…}	{ All grammatically correct English sentences }

Special Languages: $\emptyset$ - EMPTY LANGUAGE

{ε} - contains ε string only

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - $|s|$ is the length of string s.

Terminology of Languages (cont.)

EXAMPLES AND OTHER CONCEPTS:

Suppose: S is the string banana

Prefix :	ban, banana
Suffix :	ana, banana
Substring :	nan, ban, ana, banana
Subsequence:	bnan, nn

Proper prefix, subfix, or substring *cannot* be S

Terminology of Languages (cont.)

- **Language**: a set of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - $s^n = s\,s\,s\,..\,s$ (n times) $s^0 = \varepsilon$

Operations on Languages

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
concatenation of L and M written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written L*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L* denotes "zero or more concatenations of L
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of L
Intersection of L and M written $L \cap M$	$L \cap M = \{s \mid s \text{ is in } L \text{ and } s \text{ is in } M\}$

Operations on Languages

$L = \{A, B, C, D\}$ $F = \{1, 2, 3\}$

$L \cup F = \{A, B, C, D, 1, 2, 3\}$

$LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$

$L^2 = \{AA, AB, AC, AD, BA, BB, BC, BD, CA, \ldots DD\}$

$L^4 = L^2 L^2 =$ {is the set of all four-letter strings}

$L^* = \{$ All possible strings of L plus $\varepsilon \}$

$L^+ = L^* - \varepsilon$

$L(L \cup F) =$ {is the set of strings beginning with a letter followed by a letter or digit}

$L(L \cup F)^* =$ {is the set of all strings of letters and digits beginning with a letter}

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions using a set of defining rules.
- Each regular expression *r* denotes a language *L(r)*.
- A language *L(r)* denoted by a regular expression *r* is called as a regular set.

Language & Regular Expressions

- A **Regular Expression** is a Set of Rules for Constructing Sequences of Symbols (Strings) From an Alphabet Σ

$$\text{Syntax: } r = \varepsilon \mid a \mid r + r \mid rr \mid r^{*} \mid (r),\ a \text{ in } \Sigma$$

- Atomic Regular Expressions
  - Epsilon: ε, $L(\varepsilon) = \{``\}$
  - Atomic: for every a in Σ, a, $L(a) = \{`a'\}$
- Compound Regular Expressions
  - Union: $r+s$, $L(r+s) = L(r) \cup L(s)$
  - Concatenation: rs, $L(rs) = L(r)L(s)$
  - Iteration: r^{*}, $L(r^{*})=L(r)^{*}$

Left-Associative

Regular Expressions

- Eg:
  - $0+1 \Rightarrow \{0,1\}$
  - $(0+1)(0+1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow$? $\{\varepsilon, 0, 00, 000, 0000, \ldots\}$
  - $(0+1)^* \Rightarrow$? all strings with 0 and 1, including the empty string
  - Keywords = ‘else’ + ‘if’ + ‘begin’ + . . .

Algebraic Properties of Regular Expressions

AXIOM	DESCRIPTION
$r + s = s + r$	+ is commutative
$r + (s + t) = (r + s) + t$	+ is associative
$(r s) t = r (s t)$	concatenation is associative
$r (s + t) = r s + r t$ $(s + t) r = s r + t r$	concatenation distributes over +
$\varepsilon r = r$ $r \varepsilon = r$	ε is the identity element for concatenation
$r^* = (r + \varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

Extended Regular Expressions

Regular Expressions:

digit = [0-9] = ‘0’+ ‘1’+ ‘2’+ ‘3’+ ‘4’+ ‘5’+’6’+ ‘7’+’8’+’9’

letter =[a-zA-Z] = ‘A’ + . . . + ‘Z’ + ‘a’ + . . . + ‘z’

$r^+ = r\,(r)^*$

$r? = r + \varepsilon$

$. = \Sigma$

Theorem:

Regular expressions has same expressive as extended regular expressions

Regular Definitions

- To write regular expressions for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use ***regular definitions***.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

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

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

Examples:

- Integer: a non-empty string of digits

  digit = [0-9]

  integer = digit digit* // digit^+

- Identifier: strings of letters or digits, starting with a letter

  letter = [a-zA-Z]

  identifier = letter (letter + digit)*

- Whitespace: non-empty sequence of blanks, newlines, tabs

  WS = ('\n' + '\t' + ' ') $^+$

Regular Definitions (cont.)

Eg: Phone Numbers:

86-(0)21-20685397, 86-(0)571-12345676

digit = [0-9]

Σ = digit $\cup$ { -,(,) }

Country = digit^2

Area = $\text{digit}^2 + \text{digit}^3$

Phone = digit^8

Phone_number = Country '-(0)' Area '-' Phone

Regular Definitions (cont.)

- Eg: Unsigned numbers in Pascal or C

  digit $\rightarrow$ [0-9]

  digits $\rightarrow$ digit $^+$

  opt-fraction $\rightarrow$ (. digits) ?

  opt-exponent $\rightarrow$ (E (+|-)? digits) ?

  unsigned-num $\rightarrow$ digits opt-fraction opt-exponent

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

<u>Assume Following Tokens:</u>

if, then, else, relop, id, num

What language construct are they used for ?

<u>Given Tokens, What are Patterns ?</u>

if $\rightarrow$ if

then $\rightarrow$ then

else $\rightarrow$ else

relop $\rightarrow$ < + <= + > + >= + = + <>

id $\rightarrow$ letter (letter | digit)*

num $\rightarrow$ digit $^{+}$ (. digit $^{+}$) ? (E(+ | -) ? digit $^{+}$) ?

Grammar:

stmt $\rightarrow$ |if *expr* then *stmt*

|if *expr* then *stmt* else *stmt*

| ε

expr $\rightarrow$ *term* relop *term* | *term*

term $\rightarrow$ id | num

Note:

Each token has a unique token identifier to define category of lexemes

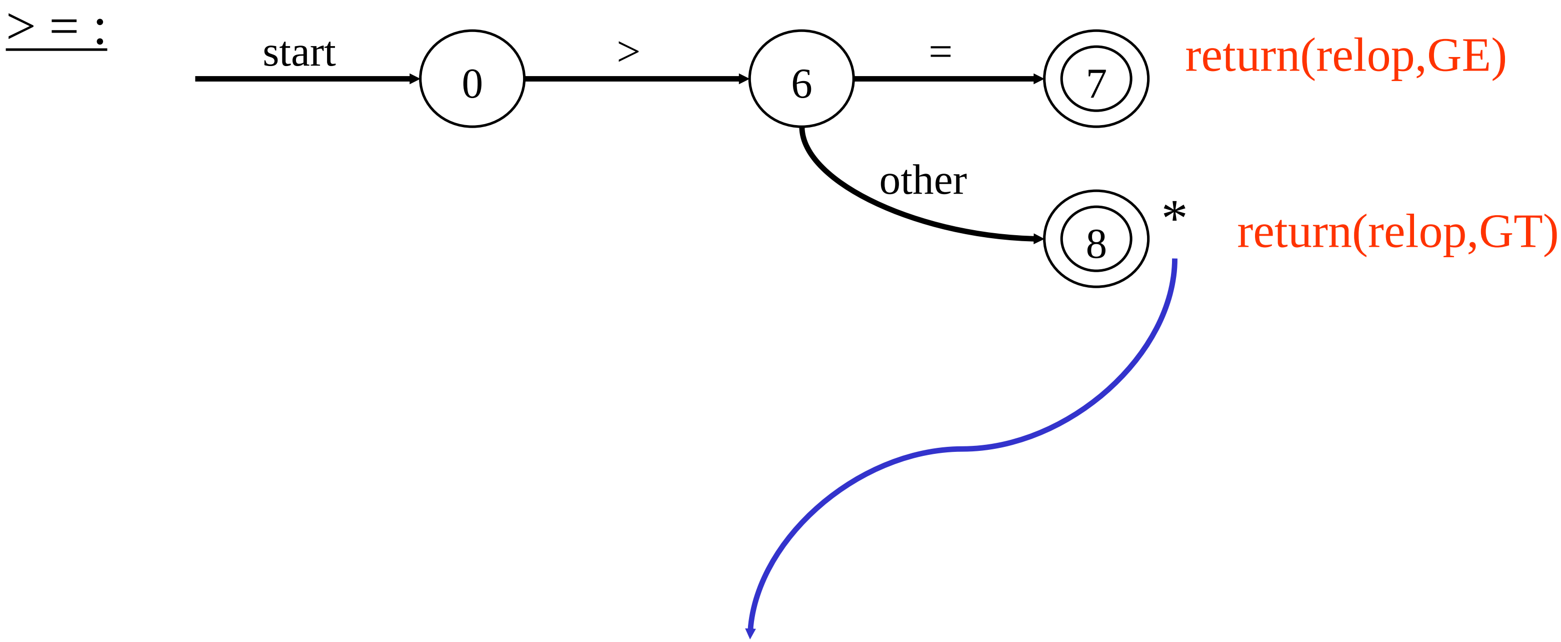
Regular Expression	Token	Attribute-Value
ws	-	-
`if`	if	-
`then`	then	-
`else`	else	-
id	id	pointer to table entry
num	num	value
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

ws= (‘\t’ + ’\n’ + ’ ’)$^{+}$

Constructing Transition Diagrams for Tokens

- Transition Diagrams (TD) are used to represent the tokens
- As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern
- Each TD has:
  - States : Represented by Circles
  - Actions : Represented by Arrows between states
  - Start State : Beginning of a pattern (Arrowhead)
  - Final State(s) : End of pattern (Concentric Circles)
- Each TD is Deterministic - No need to choose between 2 different actions !

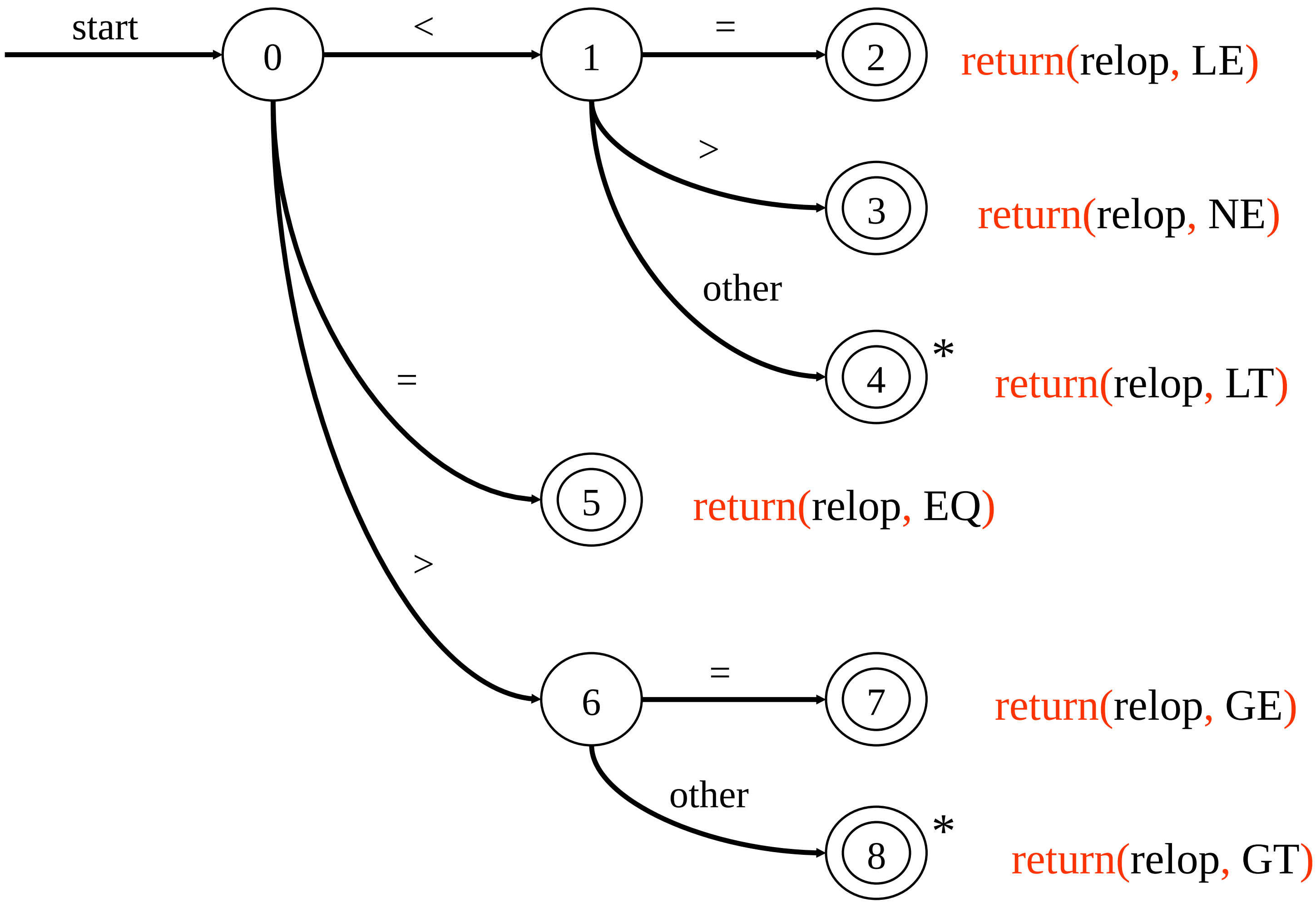
Example TDs



* means: We've accepted ">" and have read other char that must be unread.

Transition diagram for >=

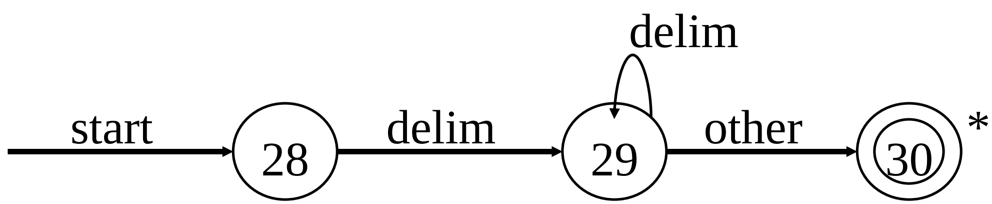
All RELOPs



Transition diagram for relop → < + <= + > + >= + = + <>

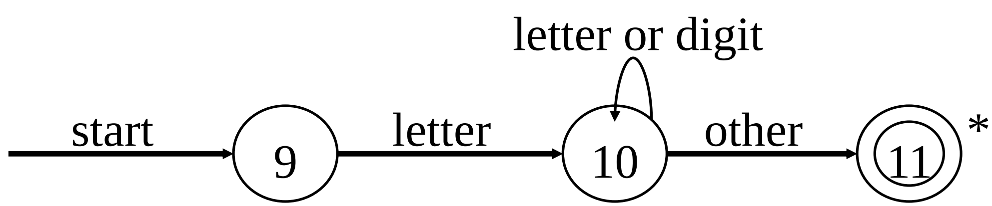
id and delim

<u>delim :</u>



Transition diagram for whitespace.

<u>id :</u>



return(get_token(), install_id())

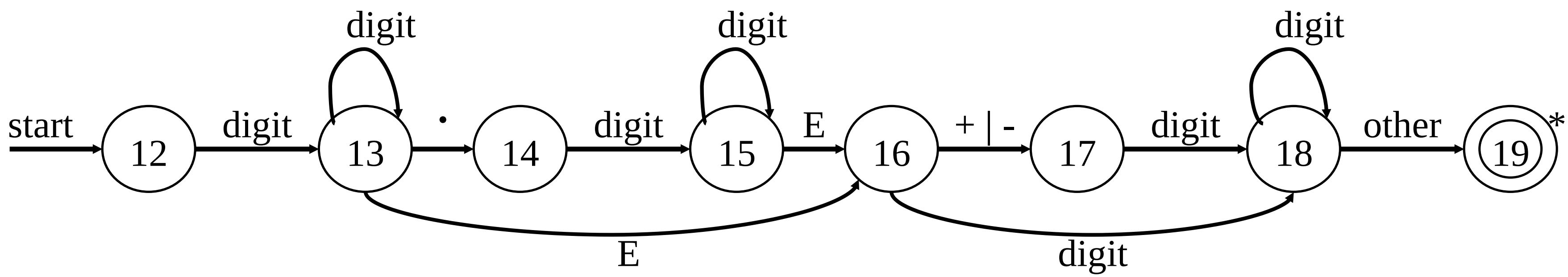
Either returns ptr or “0” if reserved

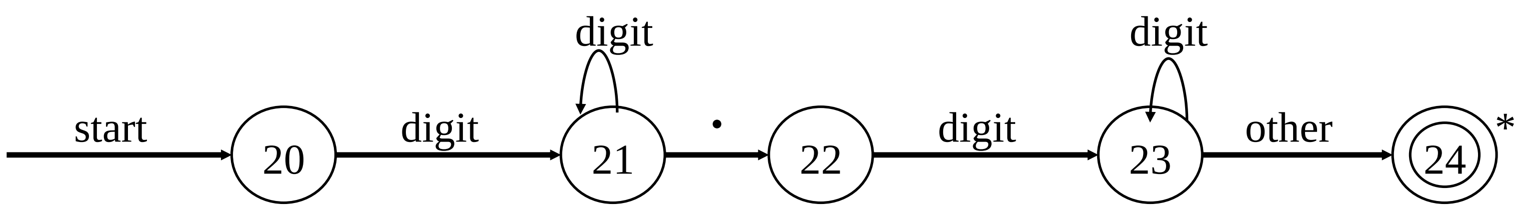
Transition diagram for identifiers and keywords.

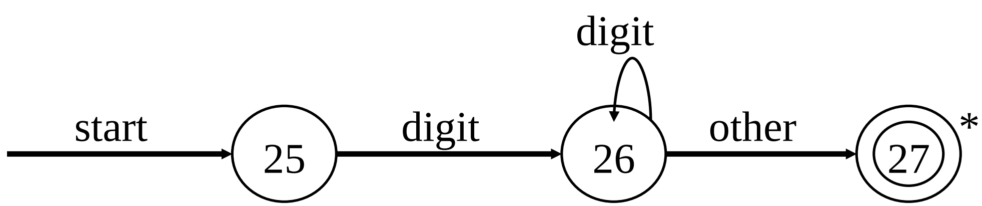
Key points

- When a token is recognized, one of the following must be done:
  - **If keyword: return Token of the keyword**
  - **If ID in symbol table: return entry of symbol table**
  - **If ID not in symbol table: install id and return the new entry of symbol table**
- Placing keywords in the symbol table is almost essential and is coded by hand, or placing keywords in other table called keywords/reserved-words table.

Unsigned number







return(num, install_num())

Ambiguities :

The lexeme for a given token must be the longest possible.

"greed"

Questions: Is ordering important for unsigned # ?

Why are there no TDs for `then`, `else`, `if` ?

Transition diagram for unsigned numbers in Pascal.

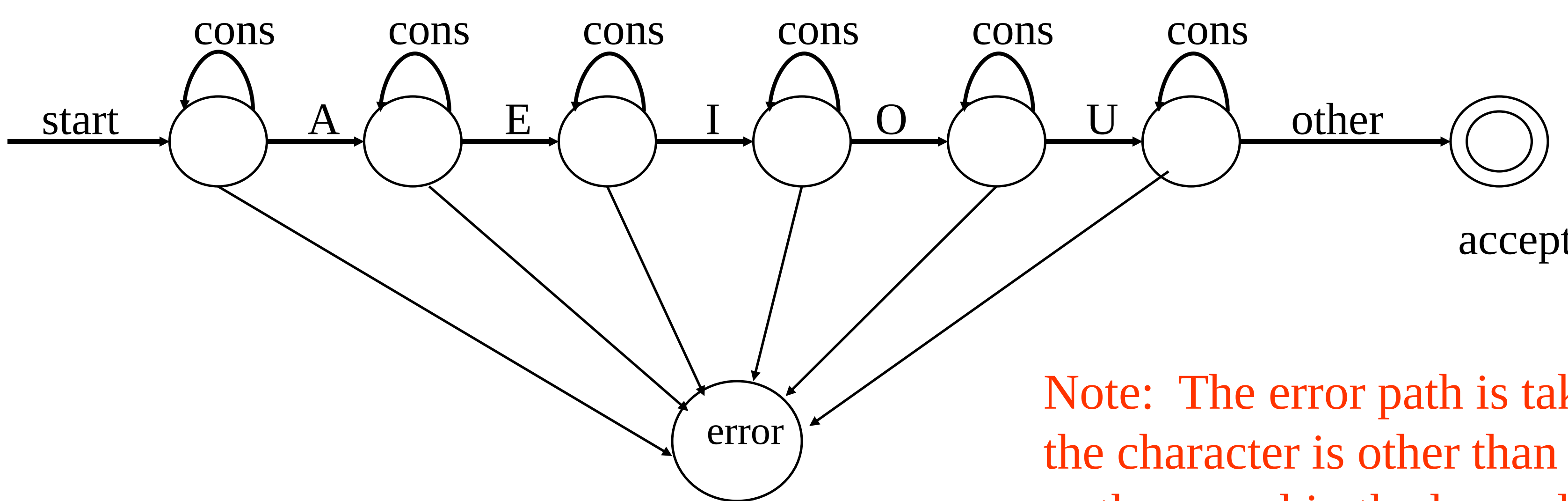
QUESTION :

What would the transition diagram (TD) for strings containing each vowel, in their strict lexicographical order, look like ?

Answer

cons $\rightarrow$ B + C + D + F + … + Z

string $\rightarrow$ cons* A cons* E cons* I cons* O cons* U cons*



Note: The error path is taken if the character is other than a cons or the vowel in the lex order.

Exercise

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

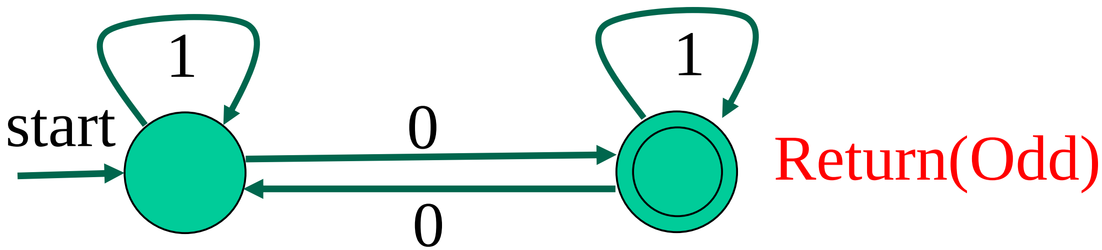
Write regular expressions for binary strings in which the number of 0:

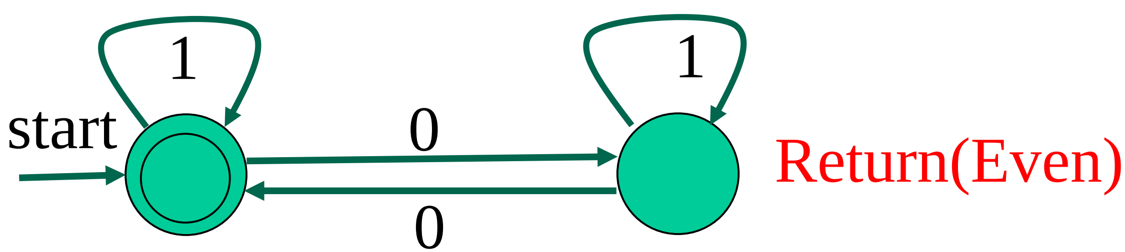
1. Odd =
2. Even =

Draw their Transition diagrams and return Token Odd and Even

Answer

1. Odd = $1^*01^*(01^*01^*)^*$
2. Even = $1^*(01^*01^*)^*$





What Else Does Lexical Analyzer Do?

All Keywords / Reserved words are matched as ids

- After the match, the symbol table or a special keyword table is consulted
- Keyword table contains string versions of all keywords and associated token values

if	257
then	258
begin	259
...	...

- When a match is found, the token is returned, along with its symbolic value, i.e., "then", 258
- If a match is not found, then it is assumed that an id has been discovered

Review

- Overall flow of a compiler
- Lexical analysis
  - ✓Token
  - ✓Regular language and regular expression
  - ✓Regular definition
  - ✓Transition Diagrams

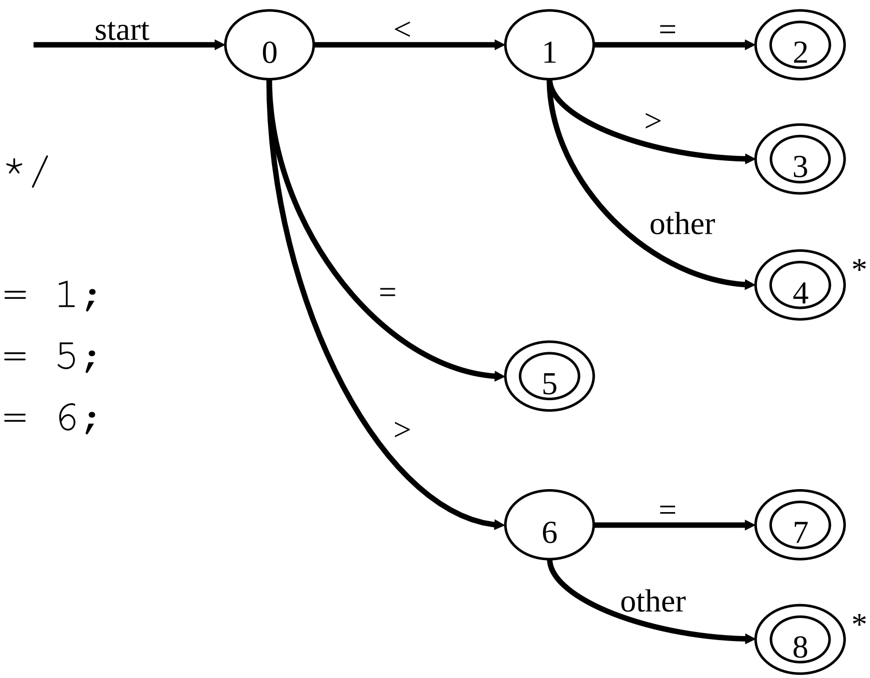
Implementing Transition Diagrams

FUNCTIONS USED

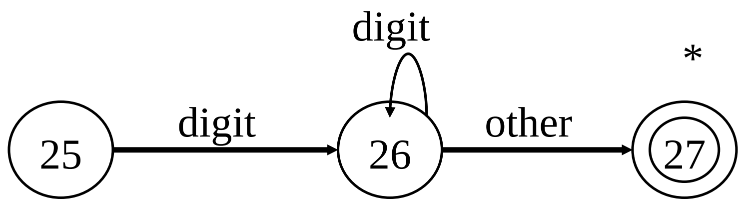
nextchar(), forward(), retract(), install_num(), install_id(), gettoken(), isdigit(), isletter(), recover()

```
lexeme_beginning = forward;
state = 0;
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c is lookahead character */
            if (c== blank || c==tab || c== newline) {
                state = 0;
                lexeme_beginning++;
              /* advance
                 beginning of lexeme */
            }
            else if (c == ‘<‘) state = 1;
            else if (c == ‘=‘) state = 5;
            else if (c == ‘>’) state = 6;
            else state = fail();
            break;
      … /* cases 1-8 here */
```

repeat until a “return” occurs



Implementing Transition Diagrams, II



advances `forward`

```
.............
case 25;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = fail();
          break;
case 26;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = 27;
          break;
case 27;  retract(1); lexical_value = install_num();
          return ( NUM );
.............
```

Case numbers correspond to transition diagram states !

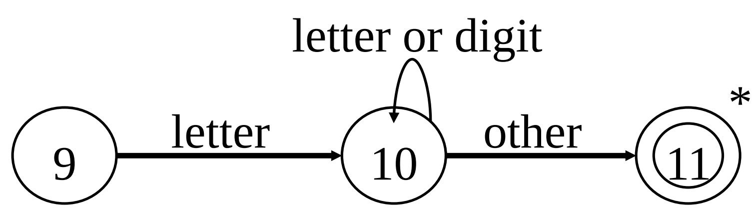
retracts `forward`

looks at the region `lexeme_beginning ... forward`

Implementing Transition Diagrams, III

```
..............
 case 9:   c = nextchar();
           if (isletter(c)) state = 10;
           else state = fail();
           break;
 case 10;  c = nextchar();
           if (isletter(c)) state = 10;
           else if (isdigit(c)) state = 10;
           else state = 11;
           break;
 case 11;  retract(1); lexical_value = install_id();
           return ( gettoken(lexical_value) );
..............
```

reads token
name from ST



When Failures Occur

```
Init fail()
{    start = state;
     forward = lexeme beginning;
     switch (start) {
        case 0:   start = 9;  break;
        case 9:   start = 12; break;
        case 12:  start = 20; break;
        case 20:  start = 25; break;
        case 25:  recover();  break;
        default:   /* lex error */
     }
     return start;
}
```

Switch to next transition diagram

C code to find next start state.

Regular expressions

Transition diagrams

Finite Automata

- A ***recognizer*** for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a ***finite automaton***.
- A finite automaton can be: ***deterministic(DFA)*** or ***non-deterministic (NFA)***
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

Finite Automata (cont.)

Finite Automata : A recognizer that takes an input string & determines whether it's a valid sentence of the language

Non-Deterministic : Has more than one alternative action for the same input symbol.

Deterministic : Has at most one action for a given input symbol.

Both types are used to recognize regular expressions.

Finite Automata (cont.)

- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automata are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: **Regular Expression → NFA → DFA** (two steps: first to NFA, then to DFA)
  - Algorithm2: **Regular Expression → DFA** (directly convert a regular expression into a DFA)

NFAs & DFAs

Non-Deterministic Finite Automata (NFAs) easily represent regular expression, but are somewhat less precise.

Deterministic Finite Automata (DFAs) require more complexity to represent regular expressions, but offer more precision.

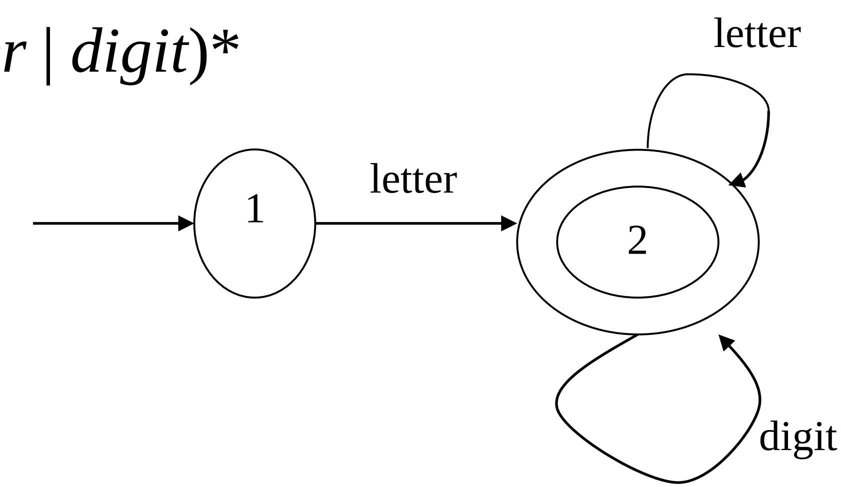
We'll review both

- A strong relationship between finite automata and regular expression
- ***Transition***: record a change from one state to another upon a match of the character or characters by which they are labeled.
- ***start state***: the recognition process begins

  drawing an unlabeled arrowed line to it coming "from nowhere"
- ***accepting states***: represent the end of the recognition process.

  drawing a double-line border around the state in the diagram

- *EX.: Identifier ->letter* (*letter* | *digit*)*



Non-Deterministic Finite Automata

An NFA is a mathematical model that consists of :

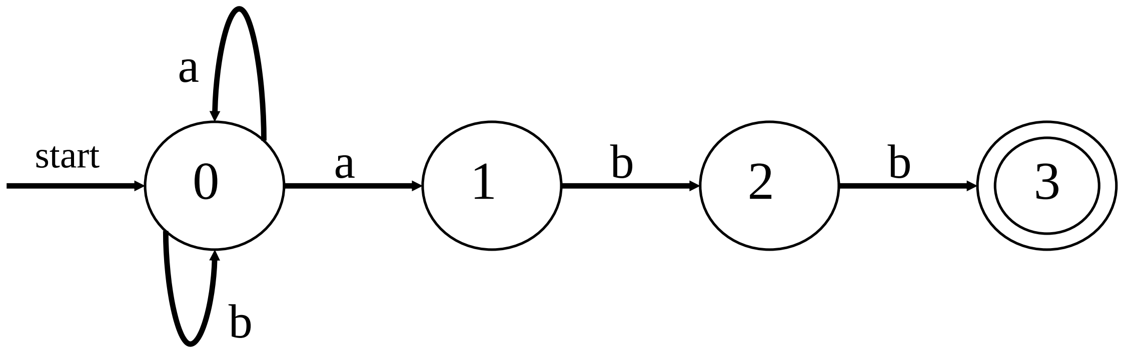
- S, a **finite** set of states
- Σ, the symbols of the input alphabet
- *move*, a transition function.
    - *move*(state, symbol) $\rightarrow$ set of states
    - *move* : $S \times \Sigma \cup \{\varepsilon\} \rightarrow \text{Pow}(S)$
- A state, $s_0 \in S$, the start state
- $F \subseteq S$, a set of final or accepting states.

NFA (cont.)

- **ε- transitions** are allowed in NFAs. In other words, we can move from one state to another one **without consuming any symbol.**
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (cont.)

- EX. Regular expression: (a+b)*abb



- The moves of string “aabb”:

Moves: $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Representing NFAs

Transition Diagrams : Number states (circles), arcs, final states, …

Transition Tables: More suitable to representation within a computer

We’ll see examples of both !

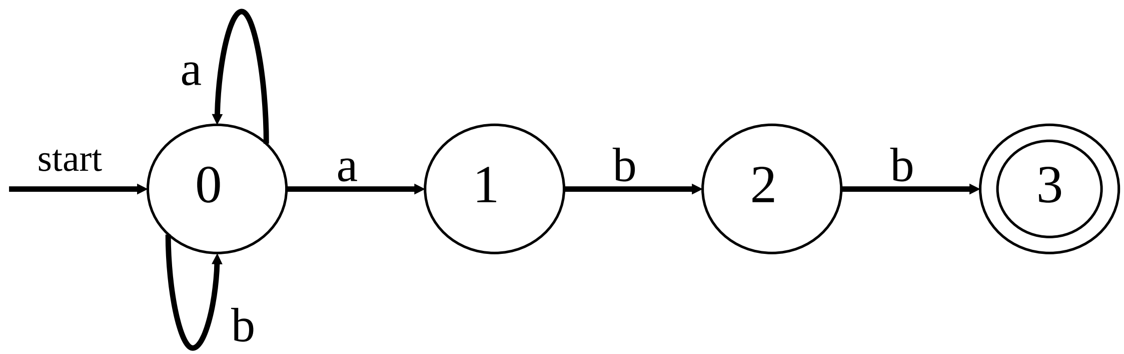
Example NFA

$S = \{ 0, 1, 2, 3 \}$

$s_0 = 0$

$F = \{ 3 \}$

$\Sigma = \{ a, b \}$



Transition graph of the NFA

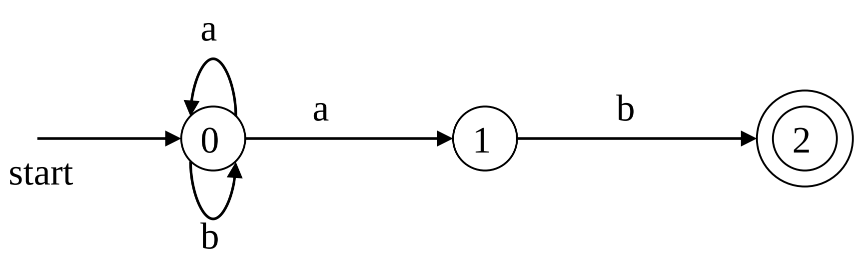
Transition table for the finite automaton

	input	
state	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

NFA (exercise)

The language recognized by this NFA is (a+b)*ab, figure out the transition graph or transition table for NFA.

NFA (exercise answer)



Transition graph of the NFA

0 is the start state s_0

{2} is the set of final states F

$\Sigma = \{a,b\}$

$S = \{0,1,2\}$

Transition Function:

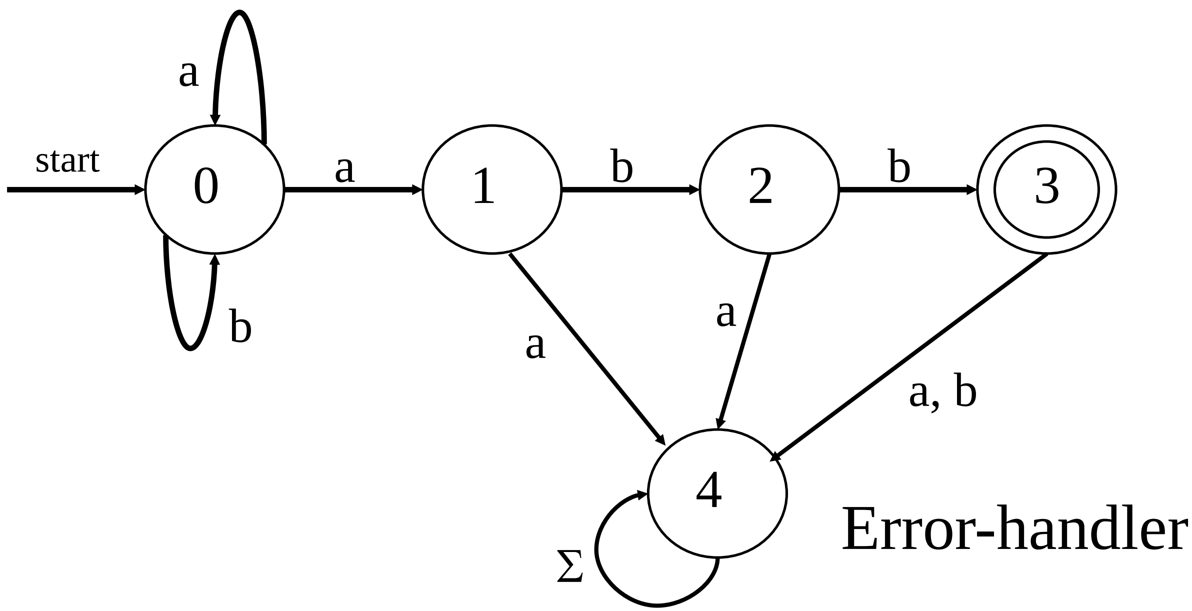
	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

Transition table for the finite automaton

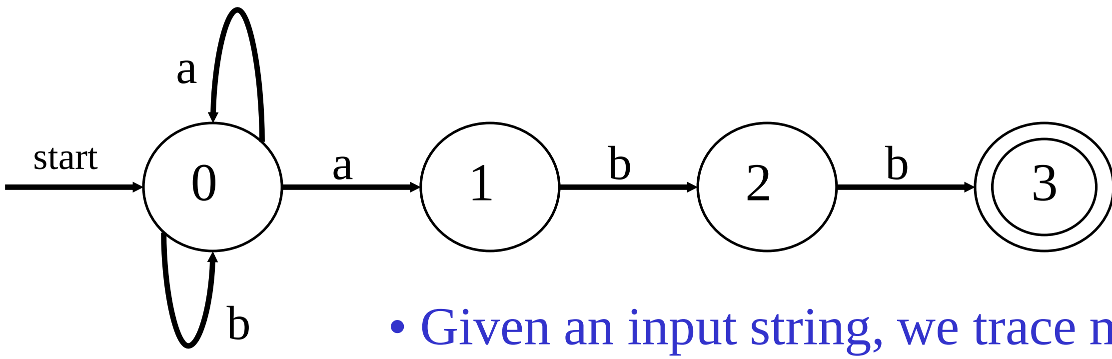
The language recognized by this NFA is $(a+b)^*ab$

Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a "death" state, and transitioning all previously undefined transition to this death state.



How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE: Input: ababb

move(0, a) = 1

move(1, b) = 2

move(2, a) = ? (undefined)

REJECT !

-OR-

move(0, a) = 0

move(0, b) = 0

move(0, a) = 1

move(1, b) = 2

move(2, b) = 3

ACCEPT !

NFA- Regular Expressions & Compilation

Problems with NFAs for Regular Expressions:

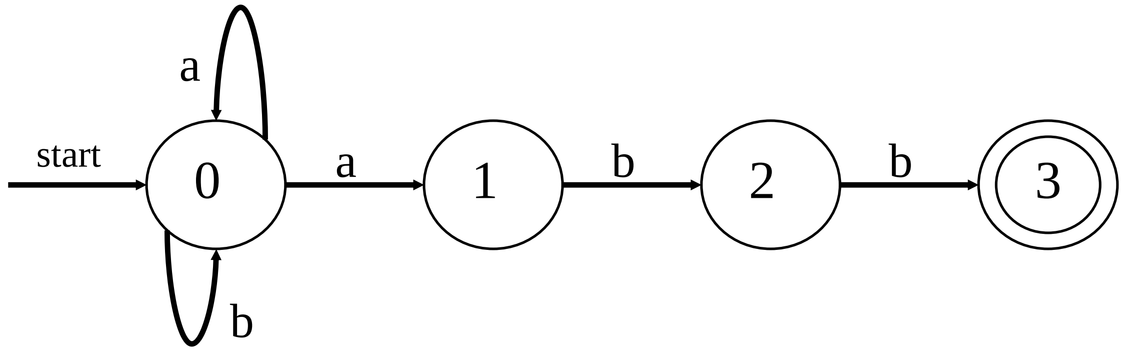
1. Valid input might not be accepted on some runs
2. NFA may behave differently on the same input

Relationship of NFAs to Compilation:

1. Regular expression is "pattern" for a "token"
2. Regular expression "recognized" by NFA
3. Tokens are building blocks for lexical analysis
4. Lexical analyzer can be described by a collection of NFAs. Each NFA is for a language of a token.

Other issues

Not all paths may result in acceptance.



ababb is accepted along path : $0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

BUT… it is <u>not accepted</u> along the valid path:

$0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$ ➡ DFA

Deterministic finite automation (DFA)

A DFA is an NFA with the following restrictions:

- ε moves are not allowed
- For every state $s \in S$, there is one and only one (at most) path from s for every input symbol $a \in \Sigma$.

  - *move*, a transition function.

    move(state, symbol) $\rightarrow$ one state

    $move : S \times \Sigma \cup \{\varepsilon\} \rightarrow S$

Implementing a DFA

•Let us assume that the end of a string is marked with a special symbol (say eof). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c); //state transition
   c ← nextchar; //read next character
end;
if s is in F then return "yes"
else return "no"
```

Simulating a DFA.

Implementing a NFA

```
S ← ε-closure({s0})          /*{ set all of states can be accessible from
                                  s0 by ε-transitions }*/
c ← nextchar
while (c != eof) {
        S ← ε-closure(move(S,c)) /* { set of all states can be
                                       accessible from a state s in S
        c ← nextchar                   by a transition on c } */
if (S∩F != Φ) then              //{ if S contains an accepting state }
    return "yes"
else return "no"
```

- This algorithm is not efficient. Why?

Converting a NFA into a DFA

- given an arbitrary NFA, construct an equivalent DFA (i.e., one that accepts precisely the same strings)
- need:

1、eliminating ε-transitions

an **ε-closure**: the set of all states reachable by ε-transitions from a state or states.

2、multiple transitions from a state on a single input character.

keeping track of the set of states that are reachable by matching a single character.

Subset construction

- Both these processes lead us to consider **sets of states** instead of **single state**. Thus, it is not surprising that the DFA we construct has as its states *sets of states* of the original NFA.
- The algorithm is called the **subset construction**

Conversion : NFA → DFA

- Algorithm Constructs a Transition Table for DFA from NFA
- Each state in DFA corresponds to a SET of states of the NFA
- Why does this occur ?
    - ε moves
    - non-determinism

    Both require us to characterize multiple situations that occur for accepting the same string.

    (Recall : Same input can have multiple paths in NFA)
- Key Issue : Reconciling AMBIGUITY !

Algorithm Concepts （cont.）

NFA $N = (S, \Sigma, s_0, F, MOVE)$

ε-Closure(s) : $s \in S$

: set of states in S that are reachable from s via ε-moves of N that originate from s.

No input is consumed

ε-Closure(T) : $T \subseteq S$, union of ε-Closure $_{t \in T}(t)$

: NFA states reachable from all $t \in T$ on ε-moves only.

move(T,a) : $T \subseteq S$, $a \in \Sigma$, union of move $_{t \in T}(t, a)$

: Set of states to which there is a transition on input a from some $t \in T$

These 3 operations are utilized by algorithms / techniques to facilitate the conversion process.

ε-Closure(T)

push all states in T onto stack;

initialize ε-closure(T) to T;

while (stack is not empty)

pop t, the top element from the stack;

for (each state u with 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

Computation of ε-closure.

Algorithm for subset construction

put ε-closure($\{s_0\}$) as an unmarked state into the set of DFA (DStates)

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

while (there is one unmarked S_1 in DStates) do

mark S_1

for (each input symbol a)

$S_2 \leftarrow \varepsilon$-closure(move($S_1$,a))

set of states to which there is a transition on a from a state s in S_1

if (S_2 is not in DStates) then

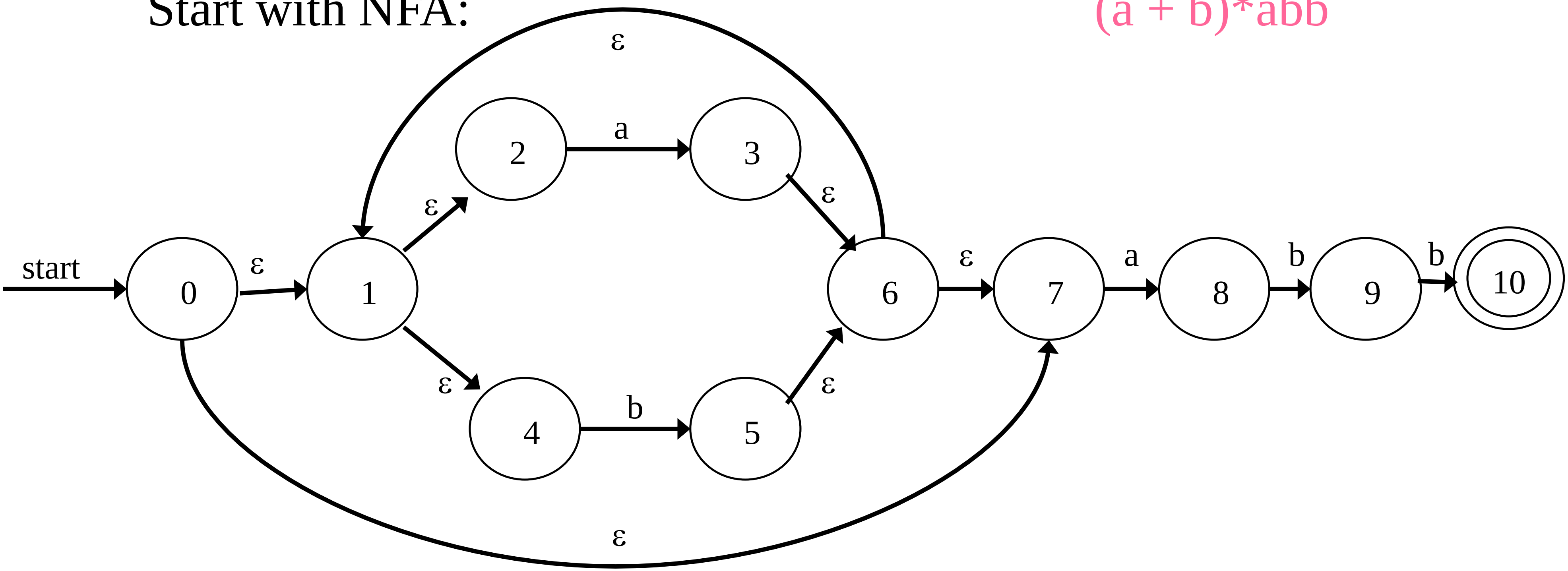
add S_2 into DStates as an unmarked state

transfunc[S_1,a] $\leftarrow S_2$

- the start state of DFA is ε-closure($\{s_0\}$)
- a state S in DStates is an accepting state of DFA if a state in S is an accepting state of NFA

Converting NFA to DFA – 1st Look

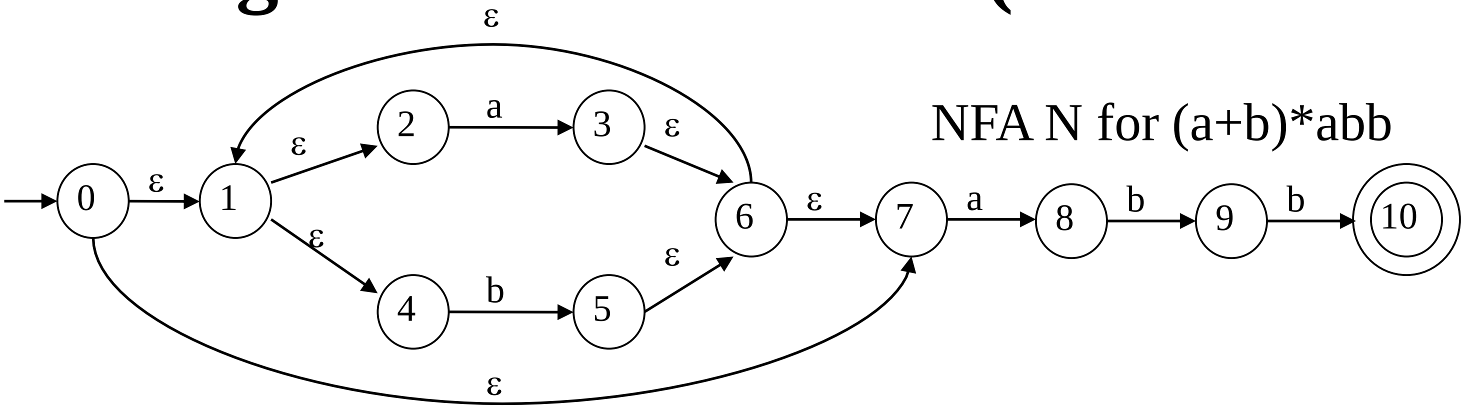
Start with NFA: (a + b)*abb



From State 0, Where can we move without consuming any input ?

This forms a new state: 0,1,2,4,7 What transitions are defined for this new state ?

Converting a NFA into a DFA (calculate ε-closure)



NFA N for (a+b)*abb

ε-closure({0}) = {0,1,2,4,7}=S_0 S_0 into DS as an unmarked state

$\Downarrow$ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] $\Leftarrow$ S_1 transfunc[S_0,b] $\Leftarrow$ S_2

$\Downarrow$ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5,9}) = {1,2,4,5,6,7,9} = S_3

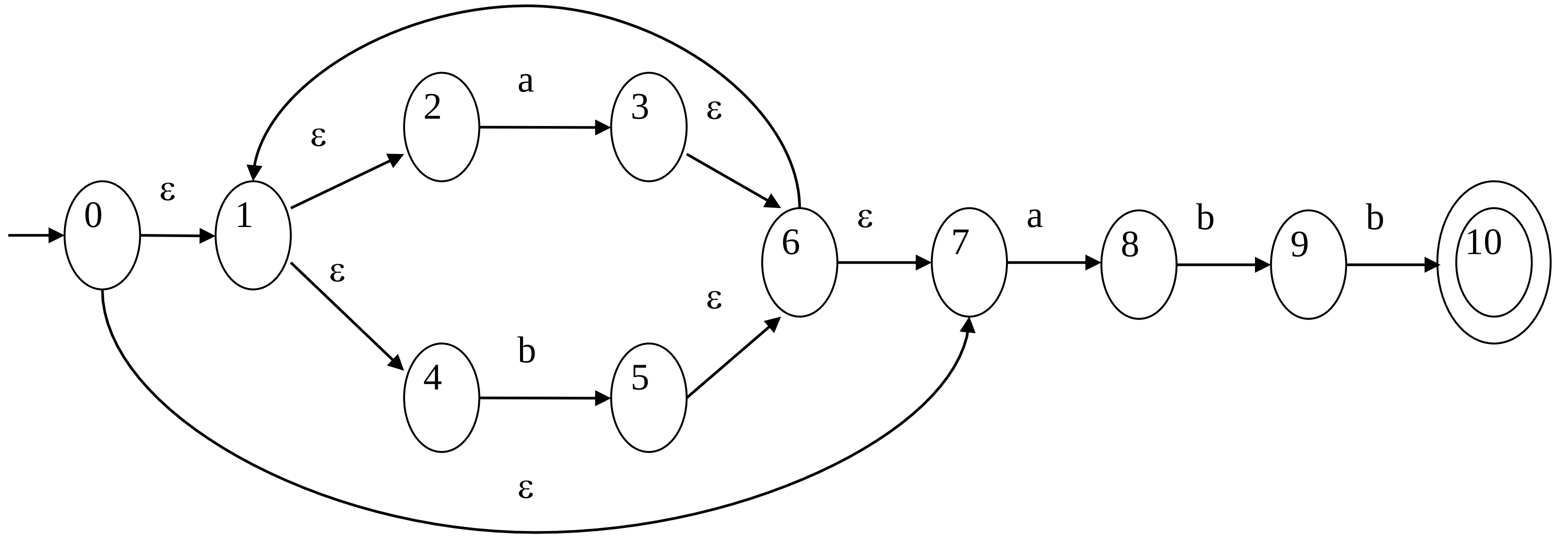
transfunc[S_1,a] $\Leftarrow$ S_1 transfunc[S_1,b] $\Leftarrow$ S_3

$\Downarrow$ mark S_2

ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} =S_2

transfunc[S_2,a] $\Leftarrow$ S_1 transfunc[S_2,b] $\Leftarrow$ S_2



$\Downarrow$ mark S_3

ε-closure(move(S_3,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_3,b)) = ε-closure({5,10}) = {1,2,4,5,6,7,10} = S_4

transfunc[S_3,a] $\leftarrow$ S_1 transfunc[S_3,b] $\leftarrow$ S_4

$\Downarrow$ mark S_4

ε-closure(move(S_4,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} =S_1

ε-closure(move(S_4,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

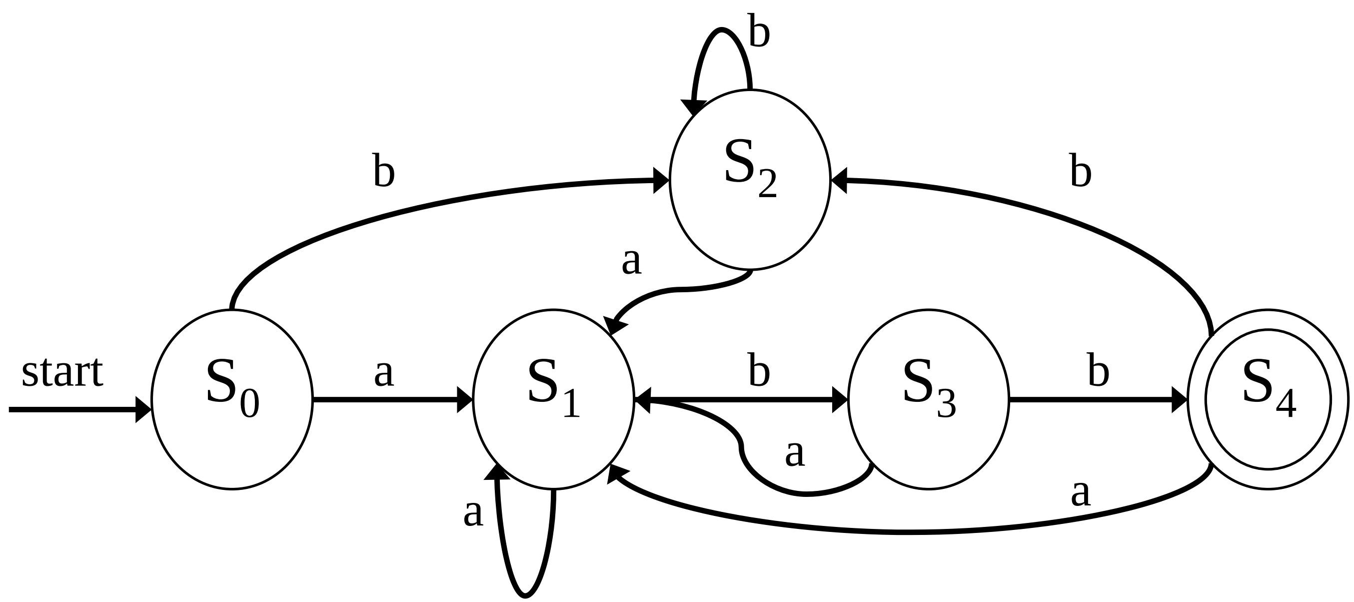
transfunc[S_4,a] $\leftarrow$ S_1 transfunc[S_4,b] $\leftarrow$ S_2

Conversion Example – continued (4)

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol a	b
S_0	S_1	S_2
S_1	S_1	S_3
S_2	S_1	S_2
S_3	S_1	S_4
S_4	S_1	S_2

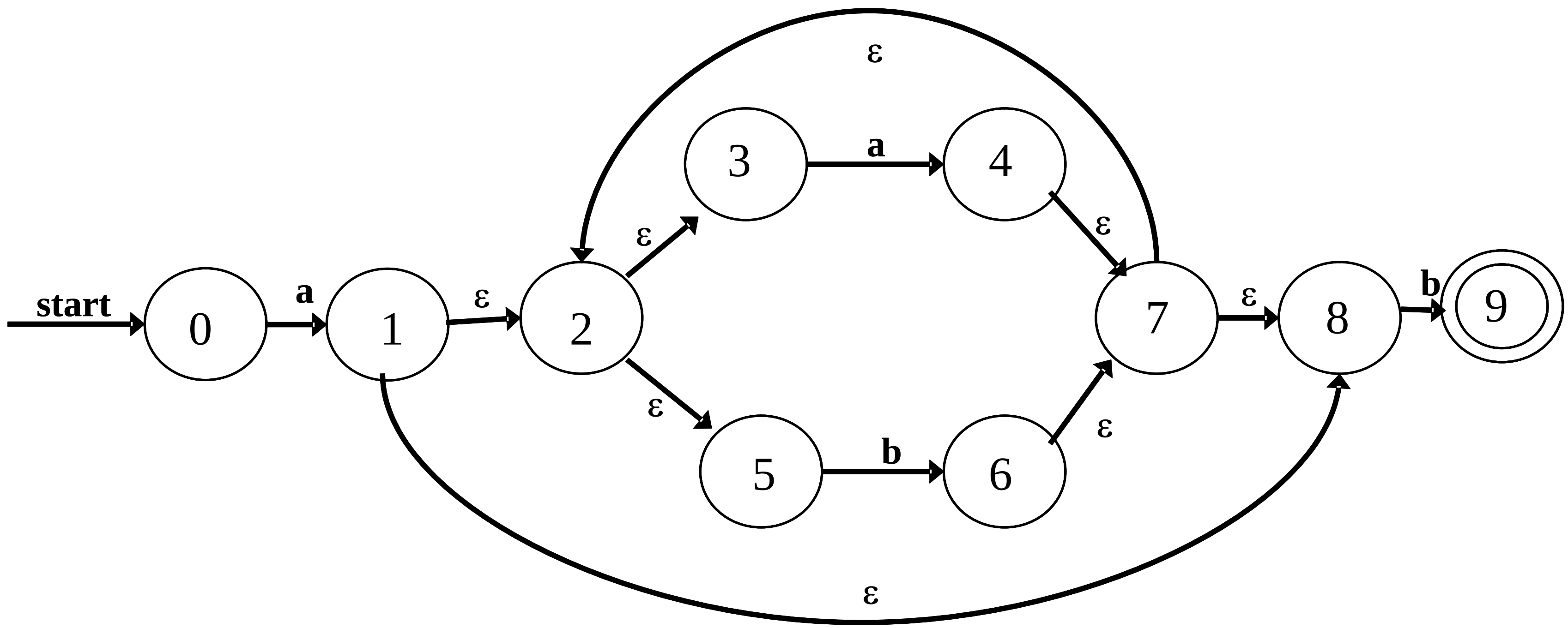
Transition table Dtran for DFA.



Result of applying the subset construction

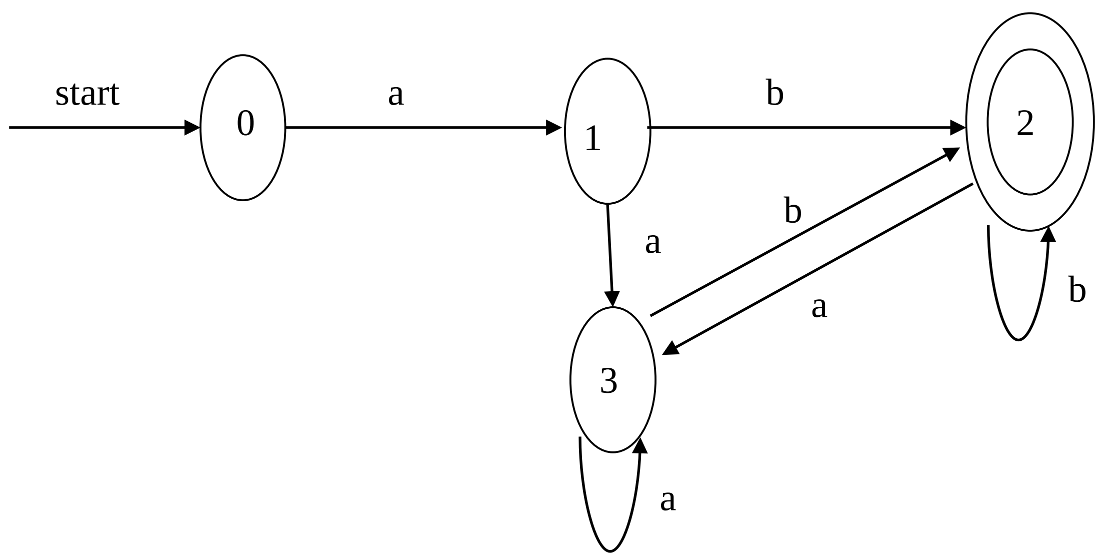
DFA (exercise in class)

The language recognized by this NFA is $a(a+b)^{*}b$, figure out a transition graph of DFA.



DFA (exercise answer)

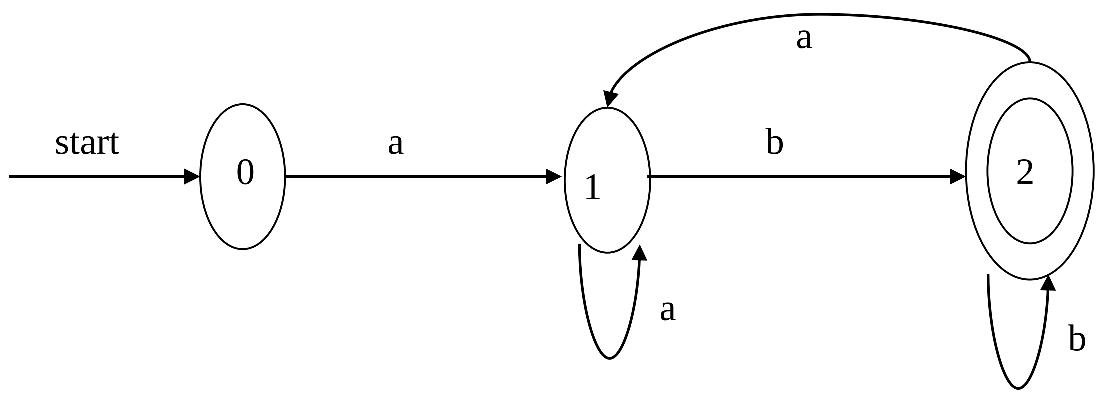
Dstates	Input Symbol a	b
S_0	S_1	
S_1	S_3	S_2
S_2	S_3	S_2
S_3	S_3	S_2



DFA accepting a (a+b) * b

DFA (exercise answer)

Dstates	Input Symbol a	b
S_0	S_1	
S_1	S_1	S_2
S_2	S_1	S_2

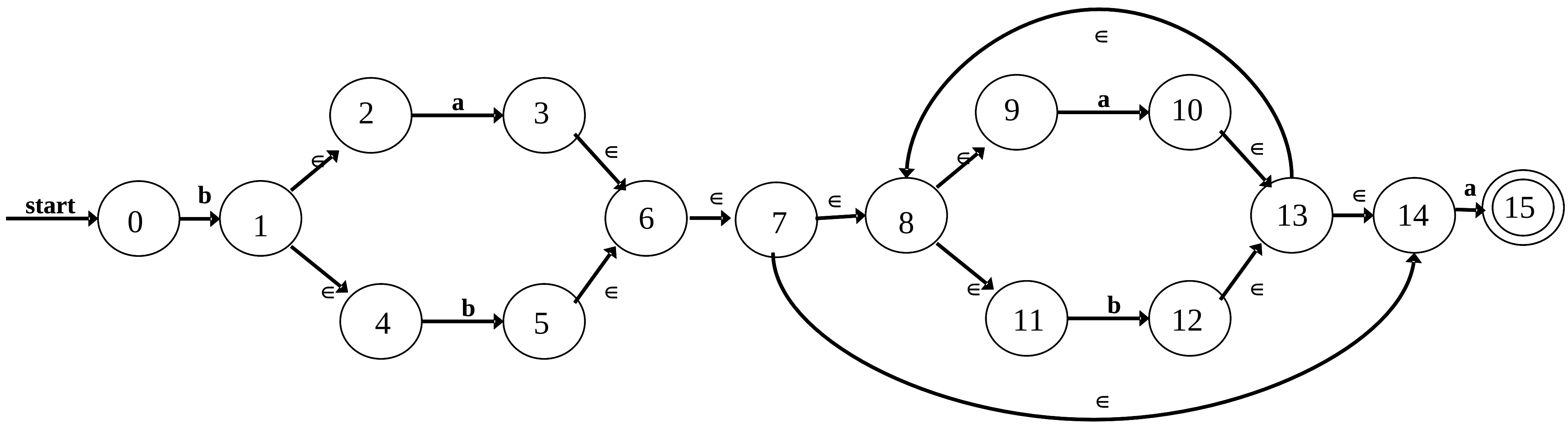


Minimal DFA accepting a $(a+b)^*$ b

Another (exercise)

- Using subset construction to construct DFA from NFA.

$$b(a+b)^{+}a$$



Regular Expression to NFA

We now focus on transforming an Reg. Expr. to an NFA

This construction allows us to take:

- Regular Expressions (which describe tokens)
- To an NFA (to characterize language)
- To a DFA (which can be "computerized")
  - Minimizing DFA

The construction process is component-wise

Builds NFA from components of the regular expression in a special order with particular techniques.

NOTE: Construction is "syntax-directed" translation, i.e., syntax of regular expression is determining factor for NFA construction and structure.

Thompson’s Construction (cont.)

- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thompson’s Construction is simple and systematic method. It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols). To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.

DFA2RE cf. Kleene's algorithm in wikipedia

Construction Algorithm : R.E. $\rightarrow$ NFA

Construction Process :

1st : Identify subexpressions of the regular expression

ε

Σ symbols

$r + s$

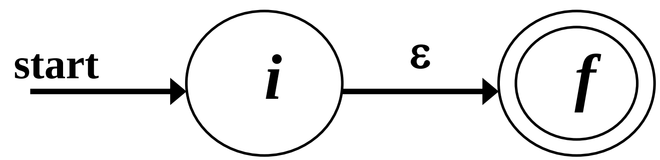
rs

r^*

2nd : Characterize "pieces" of NFA for each subexpression

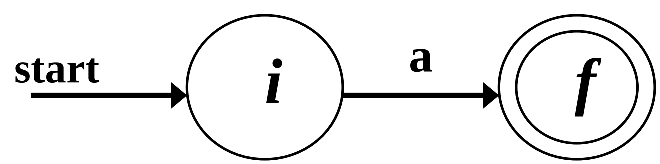
Piecing Together NFAs

1. For **ε** in the regular expression, construct NFA



L(ε)

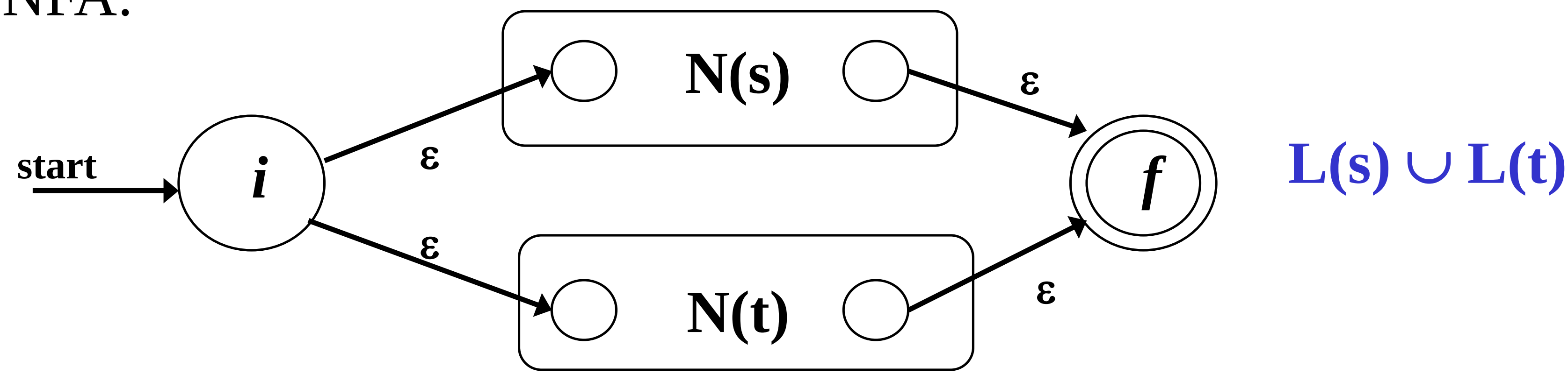
2. For a $\in \Sigma$ in the regular expression, construct NFA



L(a)

Piecing Together NFAs – continued(1)

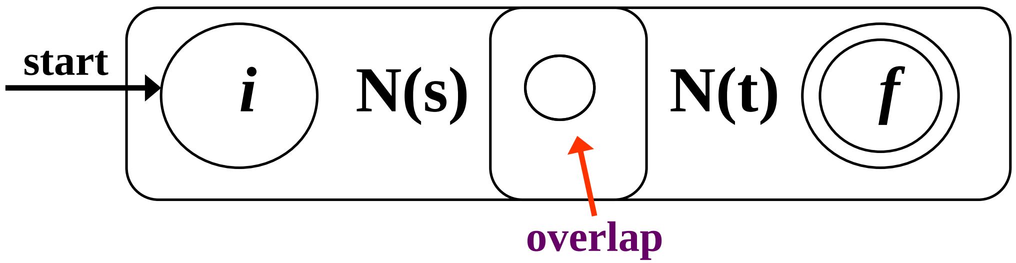
3.(a) If s, t are regular expressions, N(s), N(t) their NFAs s+t has NFA:



where *i* and *f* are new start / final states, and **ε** -moves are introduced from *i* to the old start states of N(s) and N(t) as well as from all of their final states to *f*.

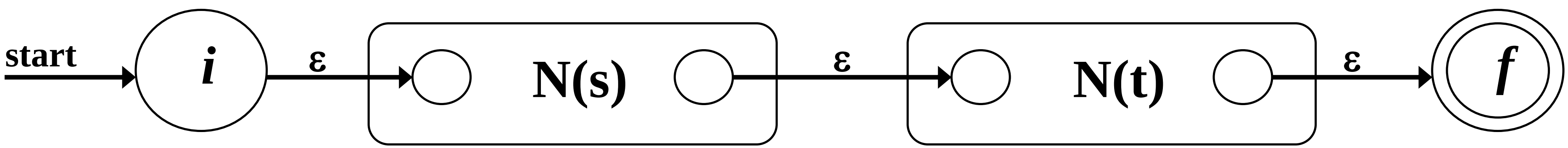
Piecing Together NFAs – continued(2)

3.(b) If s, t are regular expressions, N(s), N(t) their NFAs st (concatenation) has NFA:



L(s) L(t)

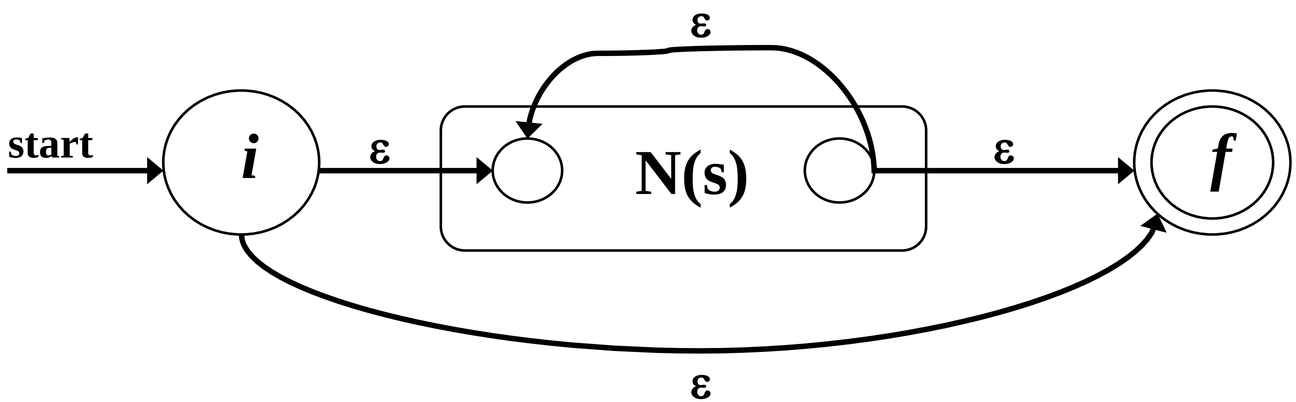
Alternative:



where *i* is the start state of N(s) (or new under the alternative) and *f* is the final state of N(t) (or new). Overlap maps final states of N(s) to start state of N(t).

Piecing Together NFAs – continued(3)

3.(c) If s is a regular expressions, N(s) its NFA, s* (Kleene star) has NFA:



where : *i* is new start state and *f* is new final state

ε -move *i* to *f* (to accept null string)

ε -moves *i* to old start, old final(s) to *f*

ε-move old final to old start (why?)

Review

- NFA & DFA
- Implementations of NFA &DFA
- NFA2DFA: Subset construction
- RE2NFA: Thompson's Construction

Properties of Construction

Let r be a regular expression, with NFA N(r), then

1. N(r) has #of states $\leq$ 2*(#symbols + #operators) of r
2. N(r) has exactly one start and one accepting state
3. Each state of N(r) has at most one outgoing edge $a \in \Sigma$ or at most two outgoing $\boldsymbol{\varepsilon}$'s
4. BE CAREFUL to assign unique names to all states !

Final Notes : R.E. to NFA Construction

- So, an NFA may be simulated by algorithm, when NFA is constructed using Previous techniques
- Algorithm run time is proportional to |N| * |x| where |N| is the number of states and |x| is the length of input
- Alternatively, we can construct DFA from NFA and use the resulting Dtran to recognize input:

	space required	time to simulate
NFA	$O(\|r\|)$	$O(\|r\|*\|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

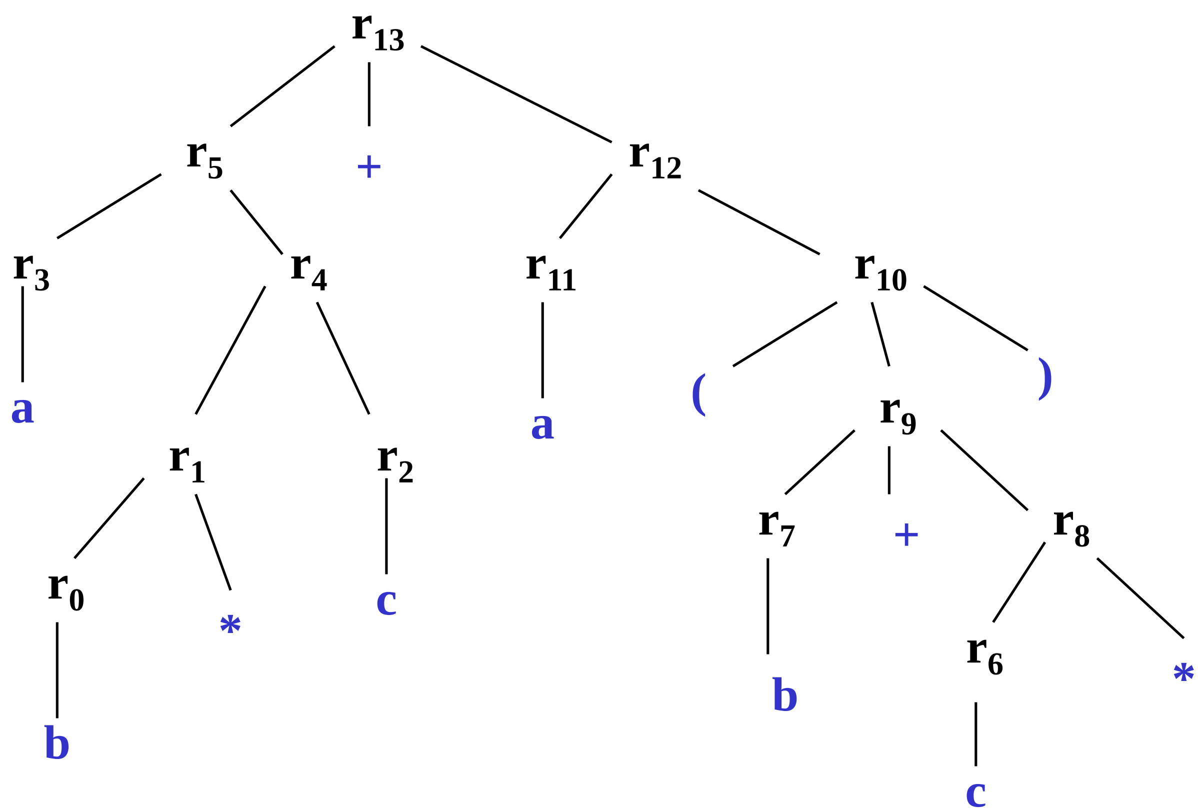
where |r| is the length of the regular expression.

Which one is better?

R.E. → NFA (example)

Example - **(ab*c) + (a(b+c*))**

Parse Tree for this regular expression:

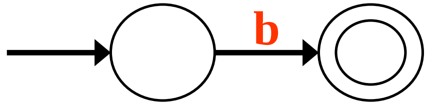


What is the NFA? Let's construct it !

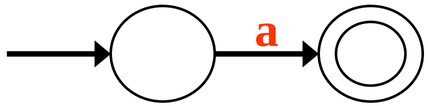
R.E. → NFA (example)– Construction(1)

$(ab^*c) + (a(b+c^*))$

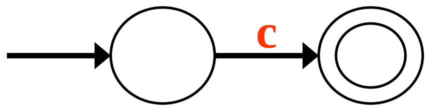
r_0:



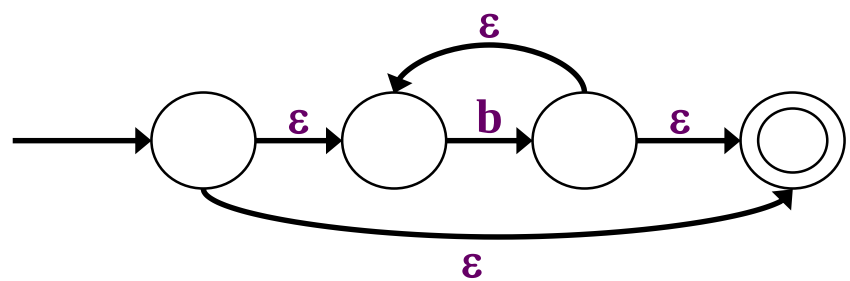
r_3:



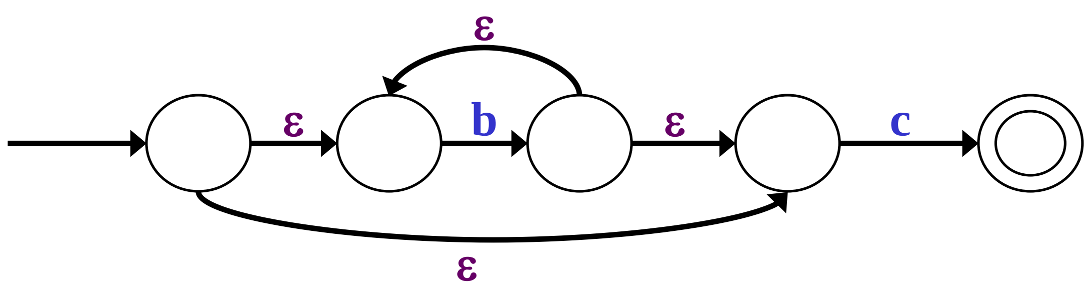
r_2:



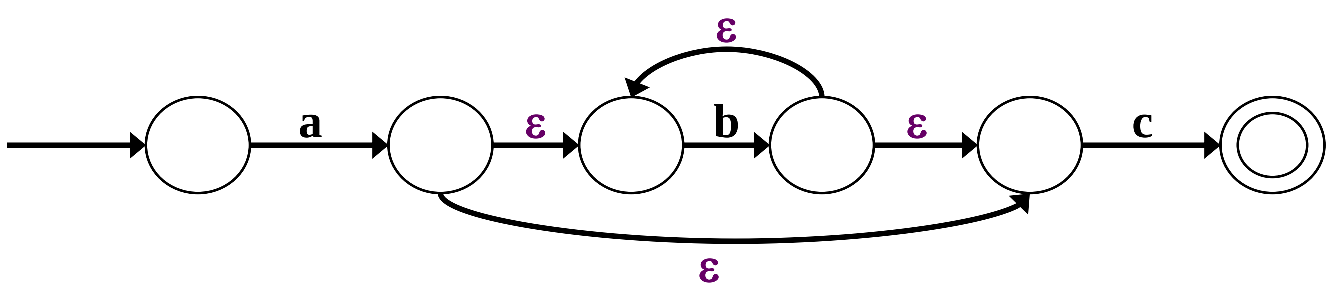
r_1:



$r_4 : r_1\ r_2$



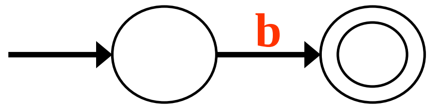
$r_5 : r_3\ r_4$



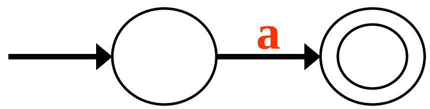
R.E. → NFA (example)– Construction(2)

$(ab^*c) + (a(b+c^*))$

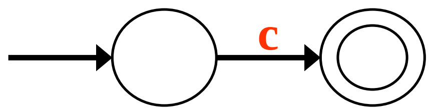
r_7:



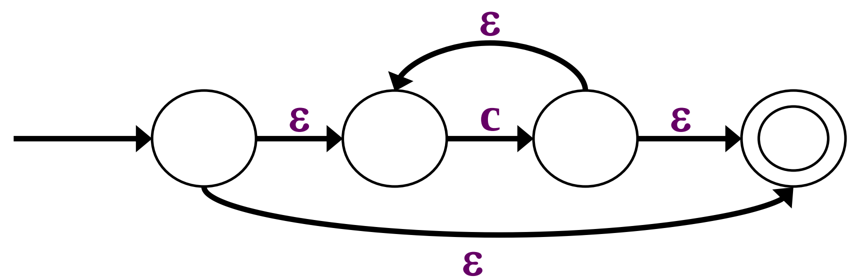
r_{11}:



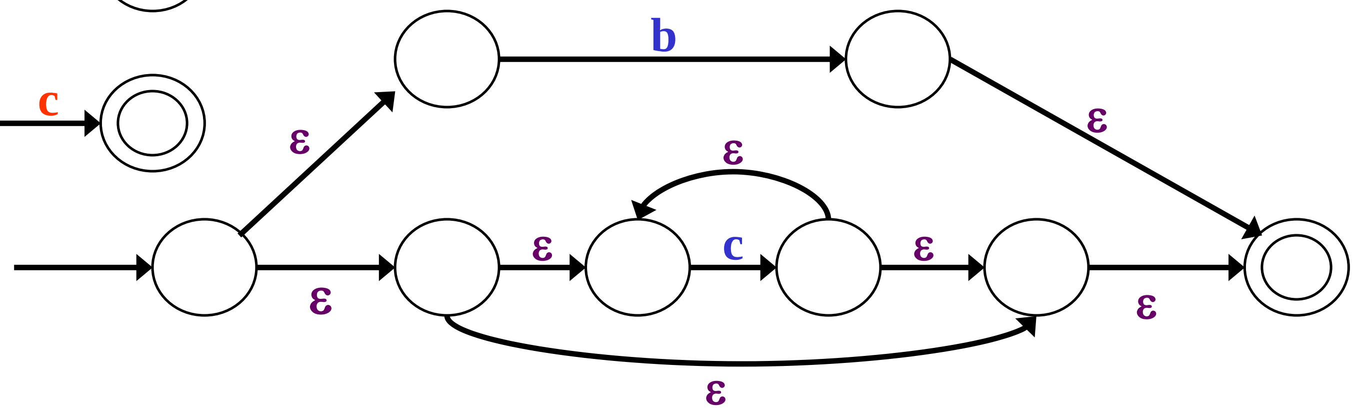
r_6:



r_8:

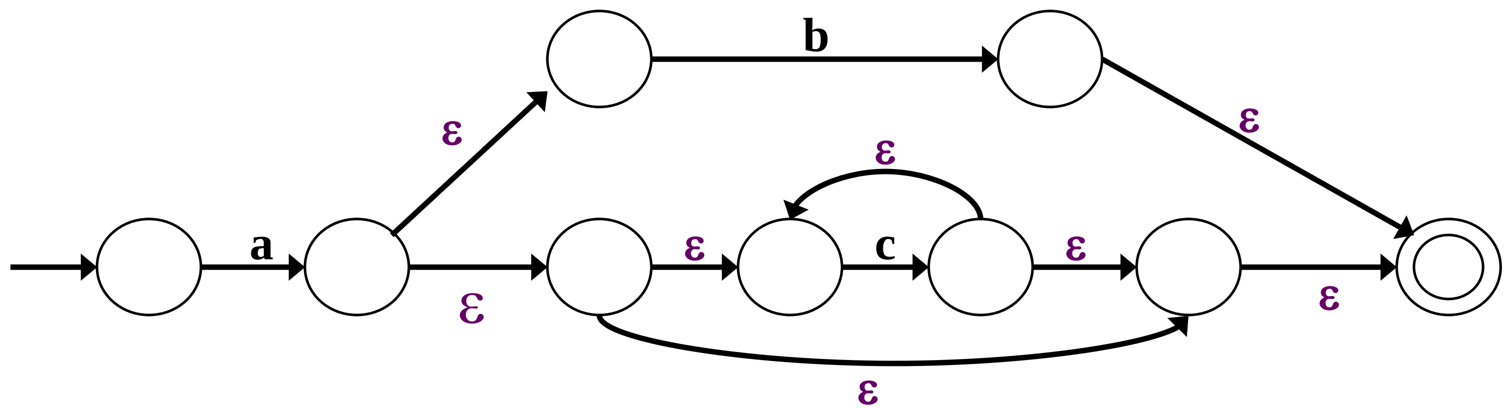


$r_9 : r_7 + r_8$



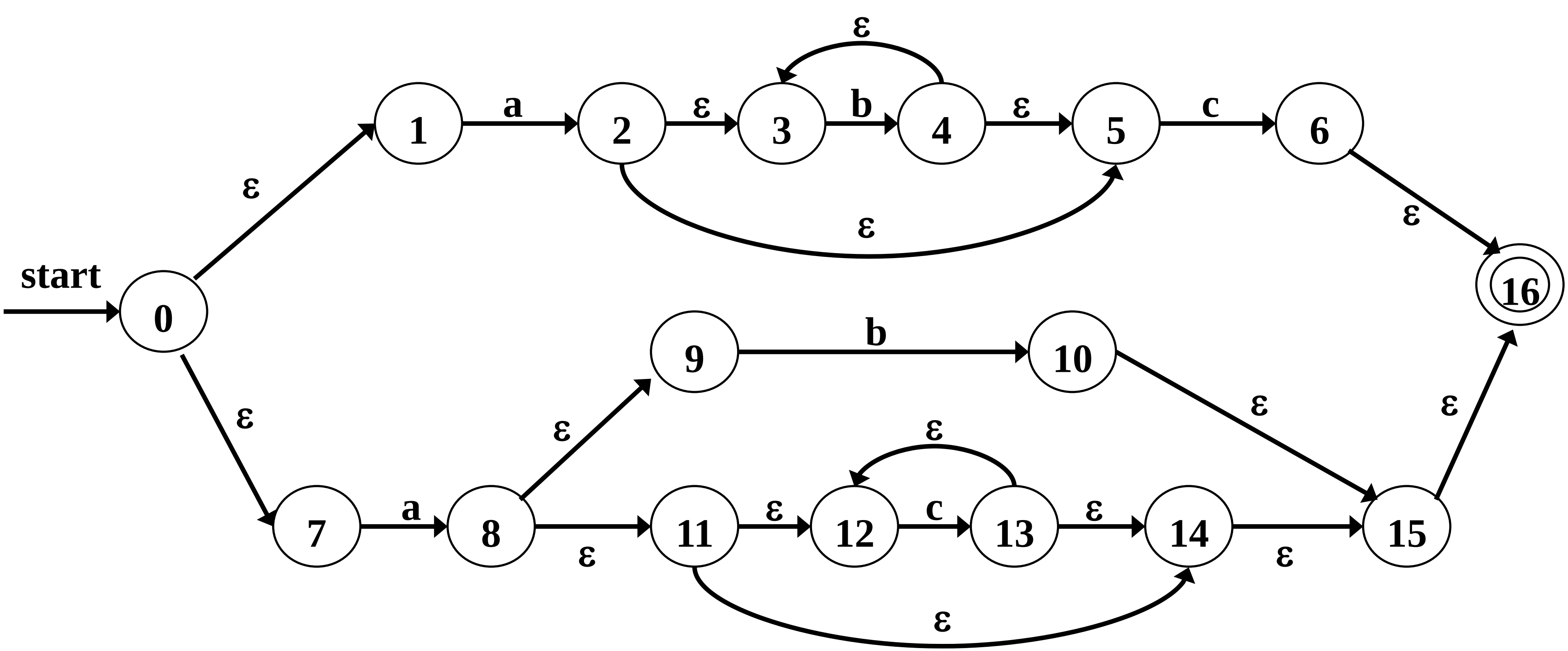
$r_{10} : r_9$

$r_{12} : r_{11} \; r_{10}$



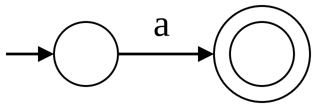
R.E. → NFA (example)– Final Step

$$r_{13} : r_5 + r_{12}$$

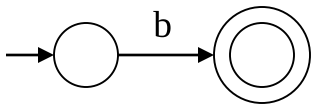


Quiz: Thompson's Construction - $(a+b)^* a$

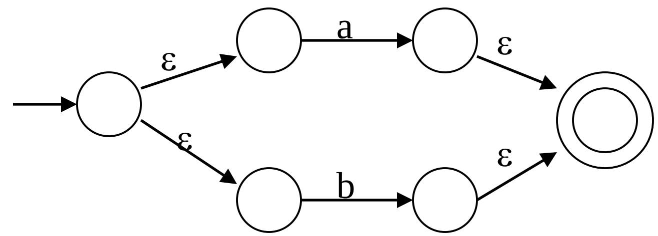
a:



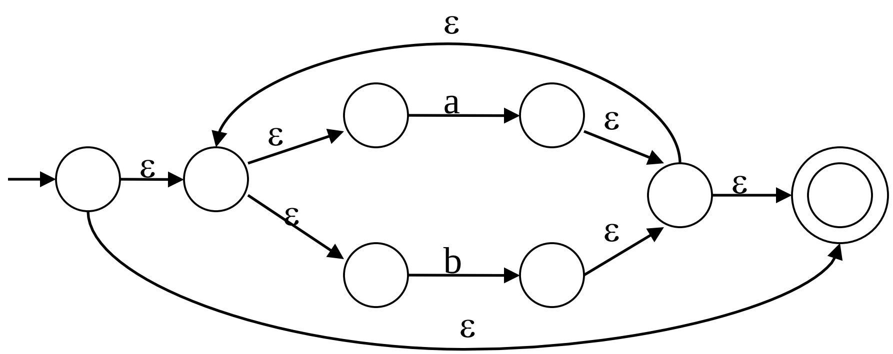
b:



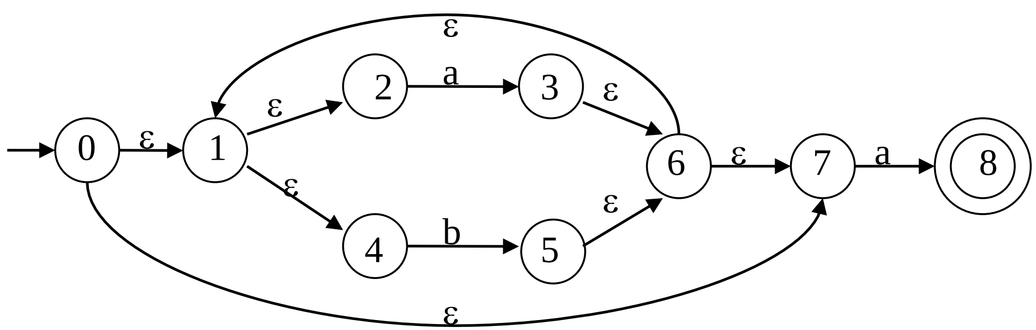
(a + b)



$(a+b)^*$



$(a+b)^* a$



Pulling Together Concepts

- **Designing Lexical Analyzer Generator**

  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer

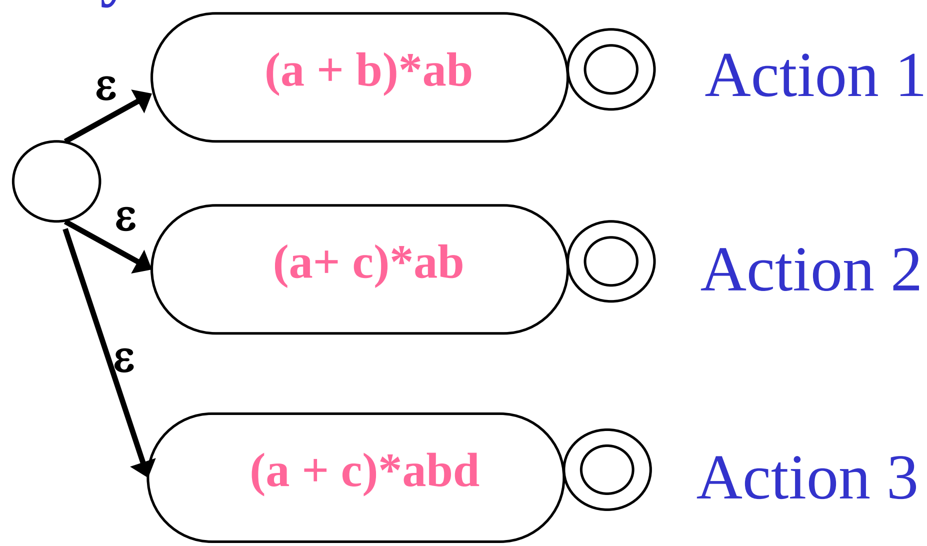
- **Recall Lex Structure**

  Pattern Action

  Pattern Action

  … …

e.g.

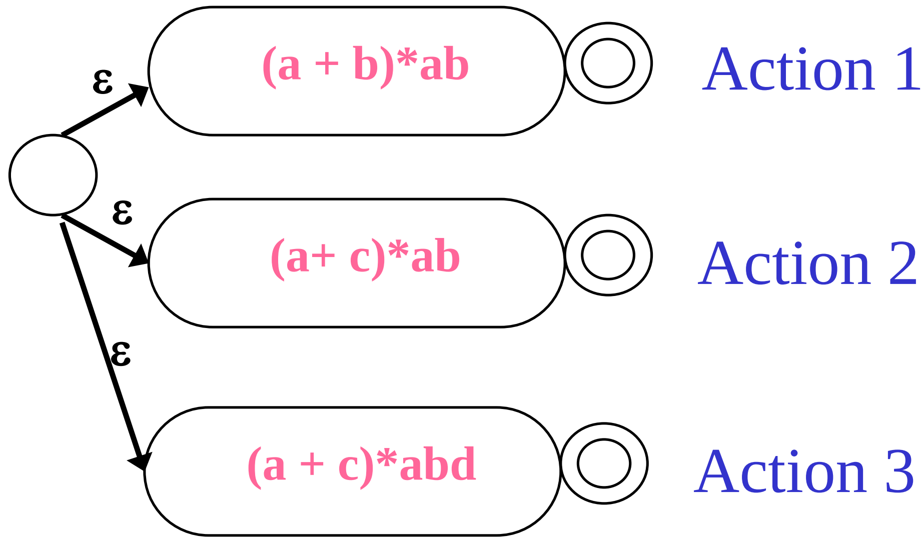


- Each pattern recognizes lexemes

Recognizer!

- Each pattern described by regular expression

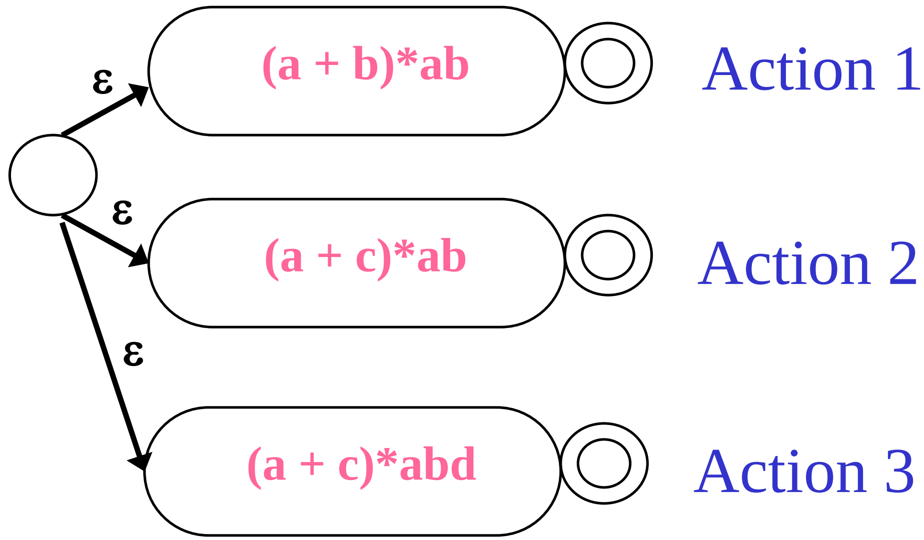
Pulling Together Concepts



- Consider: ababaabde…
  abab -> Action 1, aabd ->Action3

- Consider: aaab, which action ? 1 vs 2
  Role: perform the action whose pattern is listed first

Pulling Together Concepts



- Transform the above the NFA into a DFA

- Consider: aaab, which action ?
  Role: perform action of the first pattern whose accepting state is represented in the accepting state of the DFA

Lookahead

- IF(i,j) = 3 vs. IF(expr) THEN … in Fortran
- Keyword IF is not preserved
- How to determine IF is a keyword or a name of array
  - Lookahead: r1/r2, e.g., **IF /\(.* \) {Letters}**
  - **/ =** ε in NFA/DFA, Move lexemeBegin to the next position of /

Quiz

$(a + b)^* a$

Draw NFA

Convert NFA to DFA

Converting Regular Expressions Directly to DFAs

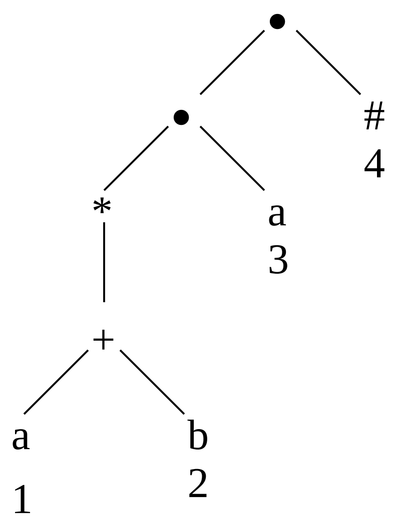
- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

  r ➔ r # **augmented regular expression**

- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #, exclude **ε**) will be numbered (position numbers).

Regular Expression ➔ DFA (cont.)

$(a+b)^* a$ ➔ $(a+b)^* a \#$ augmented regular expression



Syntax tree of $(a+b)^* a \#$

✓ each symbol is numbered (positions)
✓ each symbol is at a leave

✓ inner nodes are operators

followpos

Then we define the function **followpos** for the positions (positions assigned to leaves).

followpos(i) -- is the set of positions which can follow the position i in the strings generated by the augmented regular expression.

For example, $(\underset{1}{a} + \underset{2}{b})^* \underset{3}{a} \underset{4}{\#}$

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

followpos is just defined for leaves, it is not defined for inner nodes.

firstpos, lastpos, nullable

- To evaluate followpos, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree.

- **firstpos(n)** -- the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n.

- **lastpos(n)** -- the set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n.

- **nullable(n)** -- *true* if the empty string is a member of strings generated by the sub-expression rooted by n.
  false otherwise.

How to evaluate firstpos, lastpos, nullable

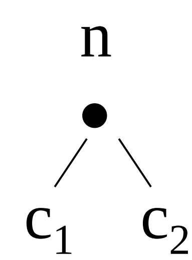
n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	**true**	Φ	Φ
leaf labeled with position i	**false**	**{i}**	**{i}**
$+$ c_1 c_2	**nullable(c_1) or nullable(c_2)**	**firstpos(c_1) $\cup$ firstpos(c_2)**	**lastpos(c_1) $\cup$ lastpos(c_2)**
$\bullet$ c_1 c_2	**nullable(c_1) and nullable(c_2)**	**if (nullable(c_1))** **firstpos(c_1) $\cup$ firstpos(c_2)** **else firstpos(c_1)**	**if (nullable(c_2))** **lastpos(c_1) $\cup$ lastpos(c_2)** **else lastpos(c_2)**
$*$ c_1	**true**	**firstpos(c_1)**	**lastpos(c_1)**

Rules for computing nullable and firstpos.

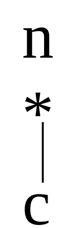
How to evaluate followpos

- Two-rules define the function followpos:

1. If **n** is **concatenation-node** with left child c_1 and right child c_2, and **i** is a position in **lastpos(c_1)**, then all positions in **firstpos(c_2)** are in **followpos(i)**.

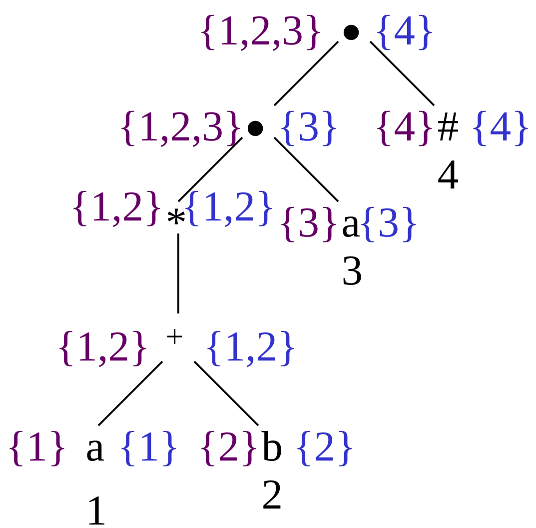


2. If **n** is a **star-node**, and **i** is a position in **lastpos(n)/lastpos(c),** then all positions in **firstpos(n)/firstpos(c)** are in **followpos(i)**.



- If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Example -- (a + b) * a



green – firstpos
blue – lastpos

Then we can calculate followpos

followpos(1) = ? {1,2,3}
followpos(2) = ? {1,2,3}
followpos(3) = ? {4}
followpos(4) = ? {}

- After we calculate follow positions, we are ready to create DFA for the regular expression.

Algorithm (RE → DFA)

- Create the syntax tree of (r) #
- Calculate the functions: firstpos, lastpos, nullable, followpos
- Put firstpos(root) into the states of DFA as an unmarked state.
- *while (there is an unmarked state S in the states of DFA) do*
  - *mark **S***
  - ***for each** input symbol **a do***
    - *let $s_1,...,s_n$ are positions in **S** and symbols in those positions are **a***
    - ***S'** ← followpos(s_1) $\cup$ ... $\cup$ followpos(s_n)*
    - ***move(S,a) ← S'***
    - ***if** (**S'** is not empty and not in the states of DFA)*
      - *put **S'** into the states of DFA as an unmarked state.*

- *the start state of DFA is firstpos(root)*
- *the accepting states of DFA are all states containing the position of #*

Example -- $(a + b)^* \, a \, \#$

$\quad\quad\quad\quad\quad\quad\;\; 1 \quad\;\; 2 \quad\quad\; 3 \;\; 4$

followpos(1)={1,2,3} followpos(2)={1,2,3} followpos(3)={4} followpos(4)={}

S_0=firstpos(root)={1,2,3}

$\Downarrow$ mark S_0

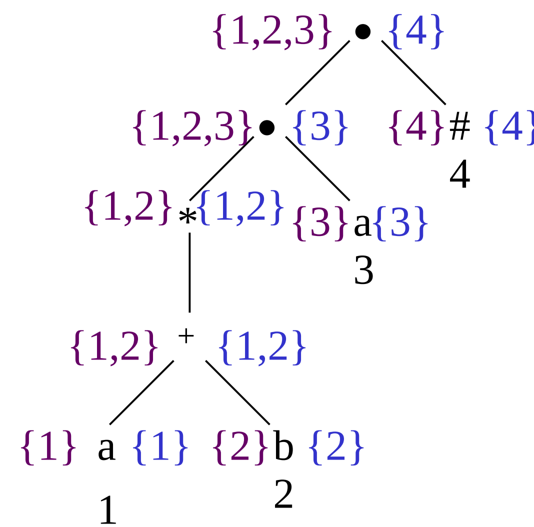
a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_1 move(S_0,a)=S_1

b: followpos(2)={1,2,3}=S_0 move(S_0,b)=S_0

$\Downarrow$ mark S_1

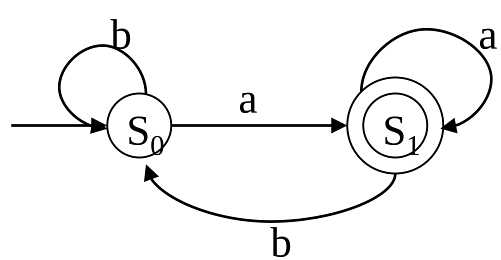
a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_1 move(S_1,a)=S_1

b: followpos(2)={1,2,3}=S_0 move(S_1,b)=S_0



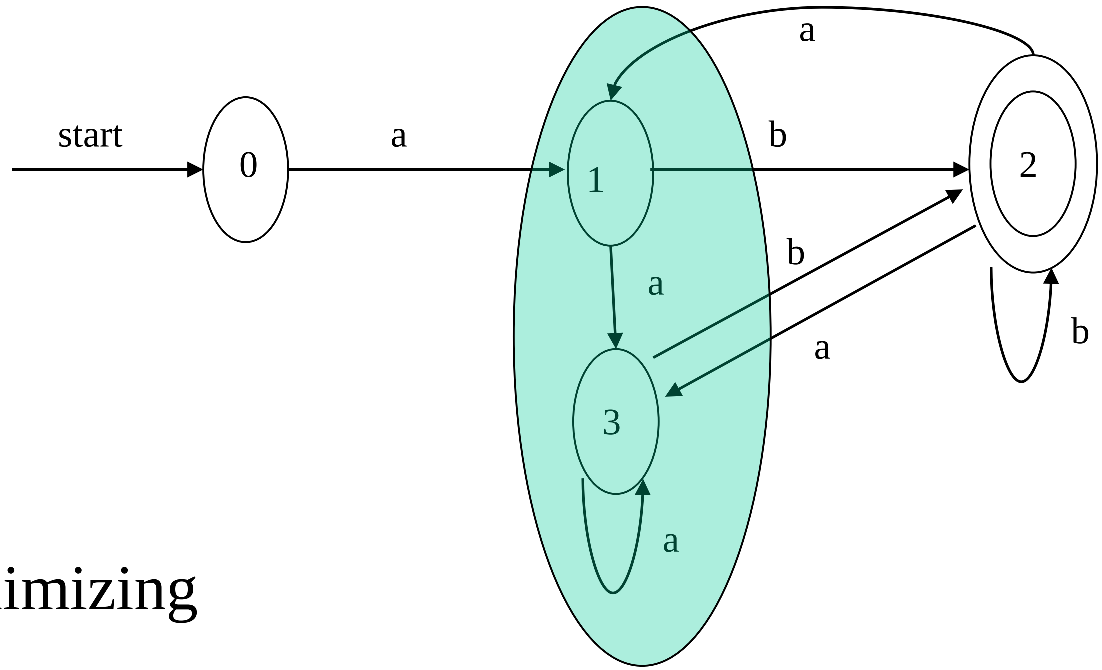
start state: S_0

accepting states: {S_1}

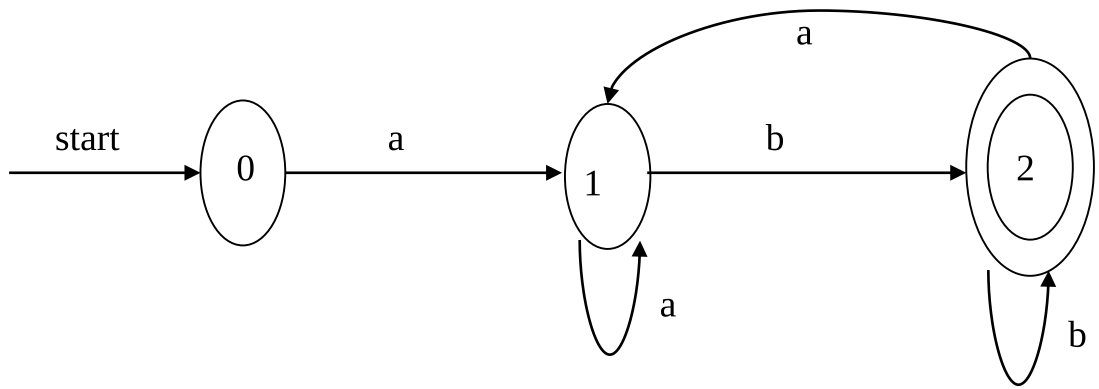


	a	b
S_0	S_1	S_0
S_1	S_1	S_0

DFA



Minimizing



DFA accepting a (a+b) * b

DFA minimization

- Each DFA has a unique minimal DFA (except that states can be given different names)

- Distinguishing extension for x and y, exists z such that

  exactly one of the two strings *xz* and *yz* belongs to *L*
- Equivalent relation: x ~ y

  there is no distinguishing extension for x and y
- Myhill–Nerode theorem

  L is regular iff $\sim_L$ has a finite number of equivalence classes

  # of minimal DFA = # of equivalence classes in $\sim_L$

Minimizing the Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states

- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group if for all input symbols a, states s_1 and s_2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

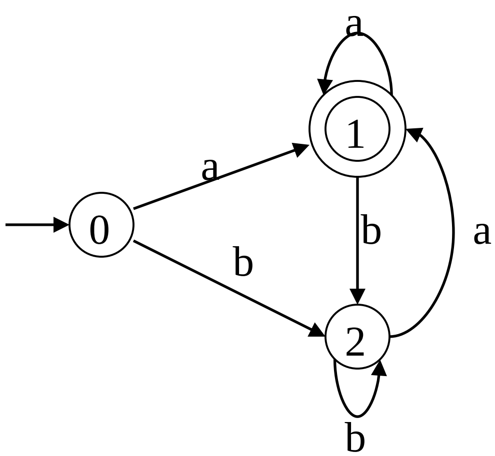
Hopcroft, John (1971), "An n log n algorithm for minimizing states in a finite automaton", Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971), New York: Academic Press, pp. 189–196.

Minimizing the Number of States of DFA (cont.)

1. Construct initial partition Π of S with two groups: accepting/ non-accepting.
2. (Construct Π_{new})For each group G of Π **do begin**
   ① Partition G into subgroups such that two states s,t of G are in the same subgroup if for all symbols a states s,t have transitions on a to states of the same group of Π.

   ② Replace G in Π_{new} by the set of all these subgroups.
3. Compare Π_{new} and Π. If equal, $\Pi_{final} := \Pi$ then proceed to 4, else set $\Pi := \Pi_{new}$ and goto 2.
4. Aggregate states belonging in the groups of Π_{final}

Minimizing DFA - Example



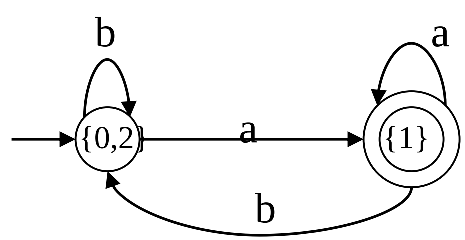
$G_1 = \{1\}$
$G_2 = \{0,2\}$

G_2 cannot be partitioned because

move(0,a)=1 move(0,b)=2
move(2,a)=1 move(2,b)=2

a	b
0->1	0->2
2->1	2->2

So, the minimized DFA (with minimum states)



Some Other Issues in Lexical Analyzer

- The lexical analyzer has to recognize the longest possible string.
  - Ex: identifier `newval` -- `n` `ne` `new` `newv` `newva` `newval`
- What is the end of a token? Is there any character which marks the end of a token?
  - It is normally not defined.
  - If the number of characters in a token is fixed, the characters cannot be in an identifier can mark the end of token.
  - We may need a lookhead
    - In Prolog: p :- X is 1. p :- X is 1.5.
    - The dot followed by a white space character can mark the end of a number.
    - But if that is not the case, the dot must be treated as a part of the number.

Some Other Issues in Lexical Analyzer (cont.)

- Skipping comments
  - Normally we don’t return a comment as a token.
  - We skip a comment, and return the next token (which is not a comment) to the parser.
  - So, the comments are only processed by the lexical analyzer, and they don’t complicate the syntax of the language.

Some Other Issues in Lexical Analyzer (cont.)

- Symbol table interface
  - symbol table holds information about tokens (at least lexeme of identifiers)
  - how to implement the symbol table, and what kind of operations.
    - hash table – open addressing, chaining
    - putting into the hash table, finding the position of a token from its lexeme.

- Positions of the tokens in the file (for the error handling).

Using Flex/Lex

Program Structure:

```
%{
Declarations
%}
Definitions  /*regular expressions */
%%
Translation rules  /*Token-action pairs*/
%%
Auxiliary procedures
```

Name the file e.g. lexer.l

Then, "`flex lexer.l or lex lexer.l`" produces the file "`lex.yy.c`" (a C-program), compile by

```
gcc -lfl lex.yy.c
```

FLEX https://github.com/westes/flex/releases

FLEX AND BISON IN C++: http://www.jonathanbeard.io/tutorials/FlexBisonC++

Example

C declarations

```
%{
        /* definitions of all constants
        LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ... */
%}
```

declarations

```
......
letter [A-Za-z]
digit [0-9]
id     {letter}({letter}|{digit})*
......
%%
```

Rules

```
if      { return(IF);}
then    { return(THEN);}
[()]    { return * yytext} /* yytext = lexemeBegin */
{id}    { yylval = install_id(); return(ID); }
......
%%
```

Auxiliary

```
install_id()
{       /* procedure to install the lexeme to the ST */
```

Example

```
%{ int num_lines = 0, num_chars = 0; %}

%%

\n    {++num_lines; ++num_chars;}
.     {++num_chars;}

%%

main( argc, argv )
int argc; char **argv;

    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
       yyin = fopen( argv[0], "r" );
    else  yyin = stdin;
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );    }
```

Another Example

```
%{ #include <stdio.h> %}
WS     [ /t/n]*

%%

[0123456789]+                  printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]*           printf("WORD\n");
{WS}                           /* do nothing */
.                              printf("UNKNOWN\n");
%%

main( argc, argv )
int argc; char **argv;
    { ++argv, --argc;
      if ( argc > 0 ) yyin = fopen( argv[0], "r" );
          else  yyin = stdin;
    yylex();     }
```

Concluding Remarks

Focused on Lexical Analysis Process, Including

- **Regular Expressions**
- **Finite Automaton （NFA, DFA）**
- **Conversion (RE ⟹ NFA,NFA ⟹ DFA, RE ⟹ DFA)**
- **Flex/Lex**
- **Interplay among all these various aspects of lexical analysis**

Looking Ahead:

The next step in the compilation process is Parsing:

- **Top-down vs. Bottom-up**

-- Relationship to Language Theory

Homework

- Construct DFA for the following regular expression in two ways:
  1. $a(a+b)^*(b+\varepsilon)$
  2. $a(a+b)^*a(a+b)a$
- The first way is to construct NFA by using Thompson’s algorithm, then to construct DFA from NFA by using subset construction algorithm .
- The second way is to construct DFA from regular expression directly.
- Minimizing the states of DFA.