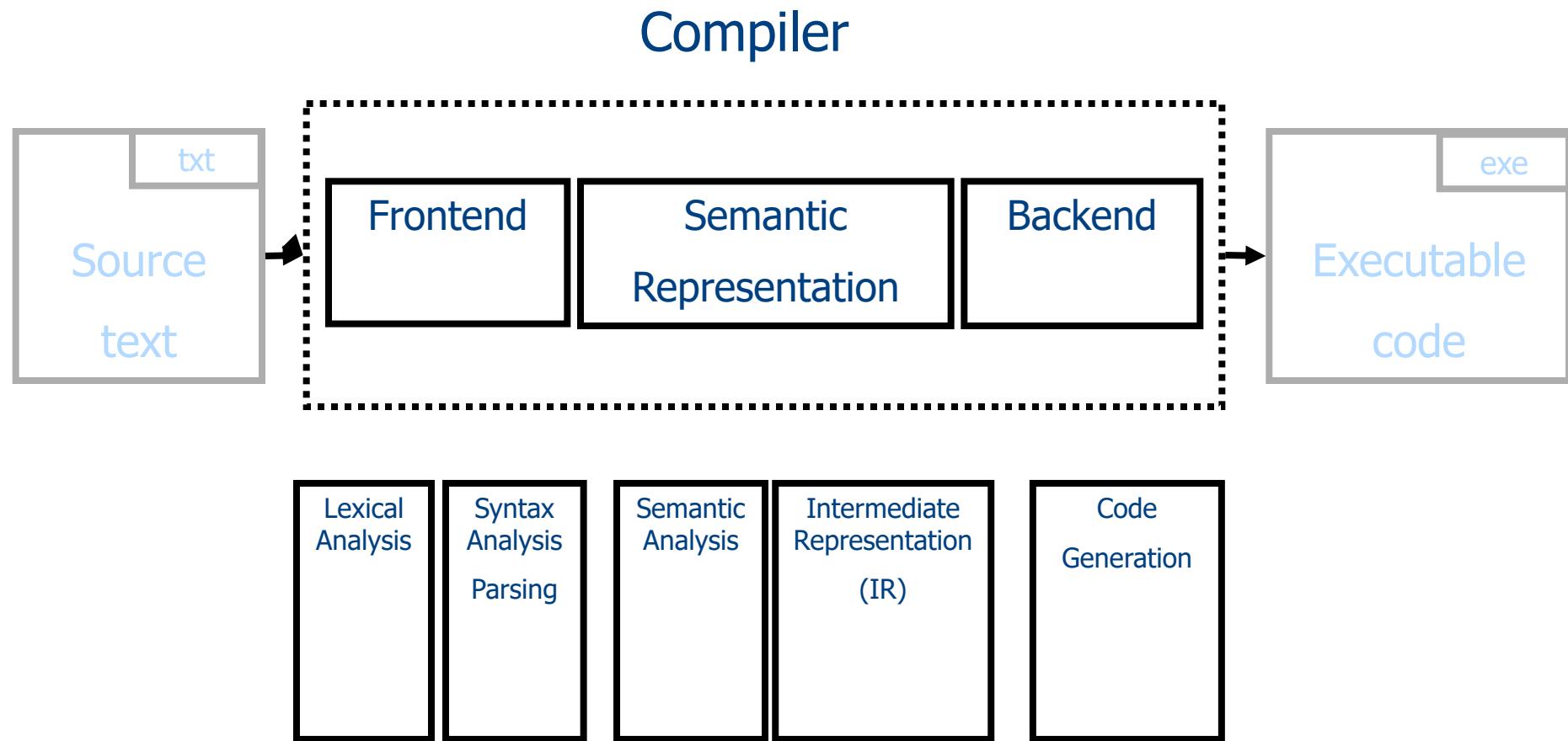


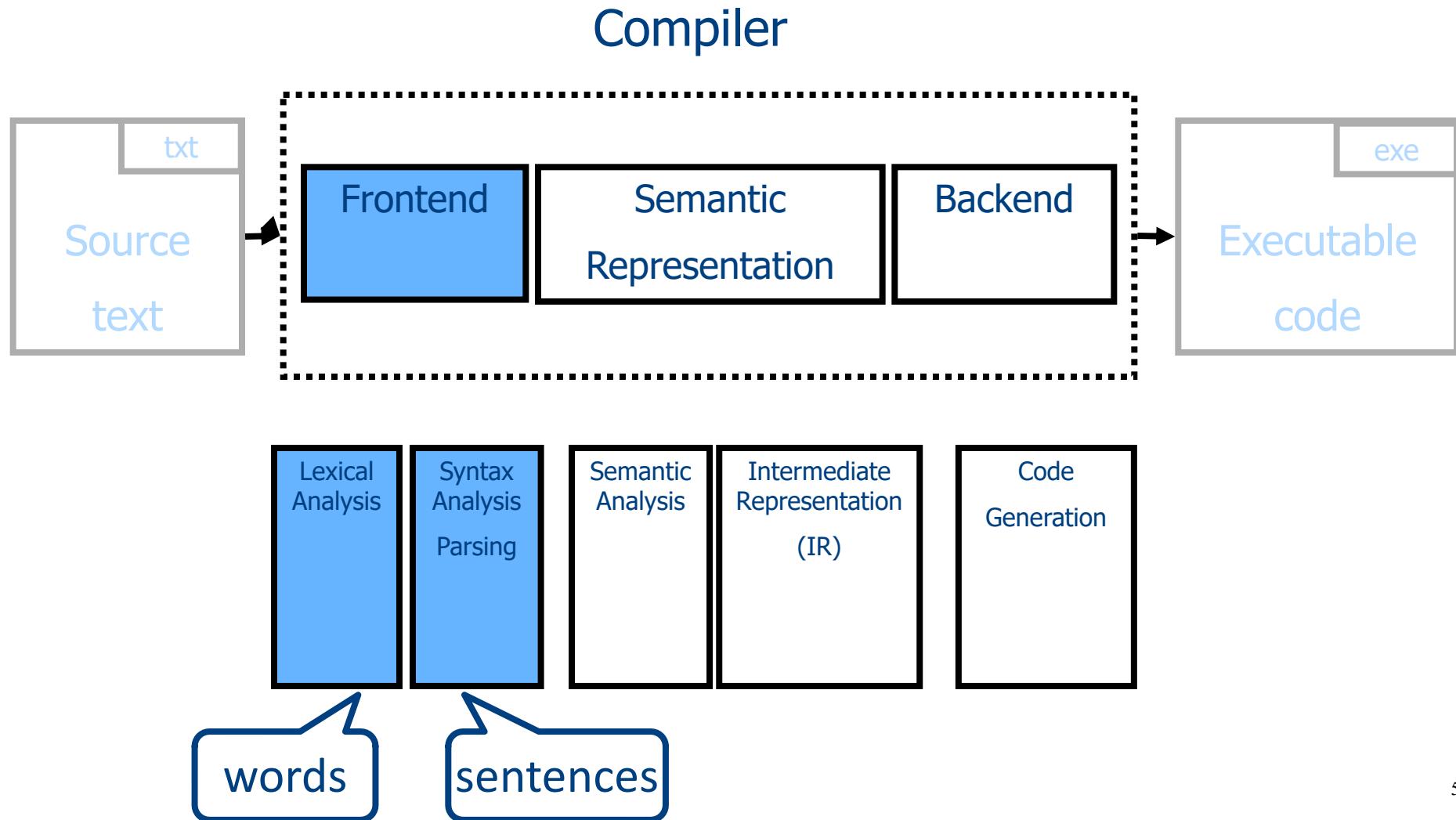
# Lexical Analysis

Modern Compiler Design: Chapter 2.1

# Conceptual Structure of a Compiler



# Conceptual Structure of a Compiler



# What does Lexical Analysis do?

- Language: fully parenthesized expressions

Expr → Num | LP Expr Op Expr RP

Num → Dig | Dig Num

Dig → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

LP → '('

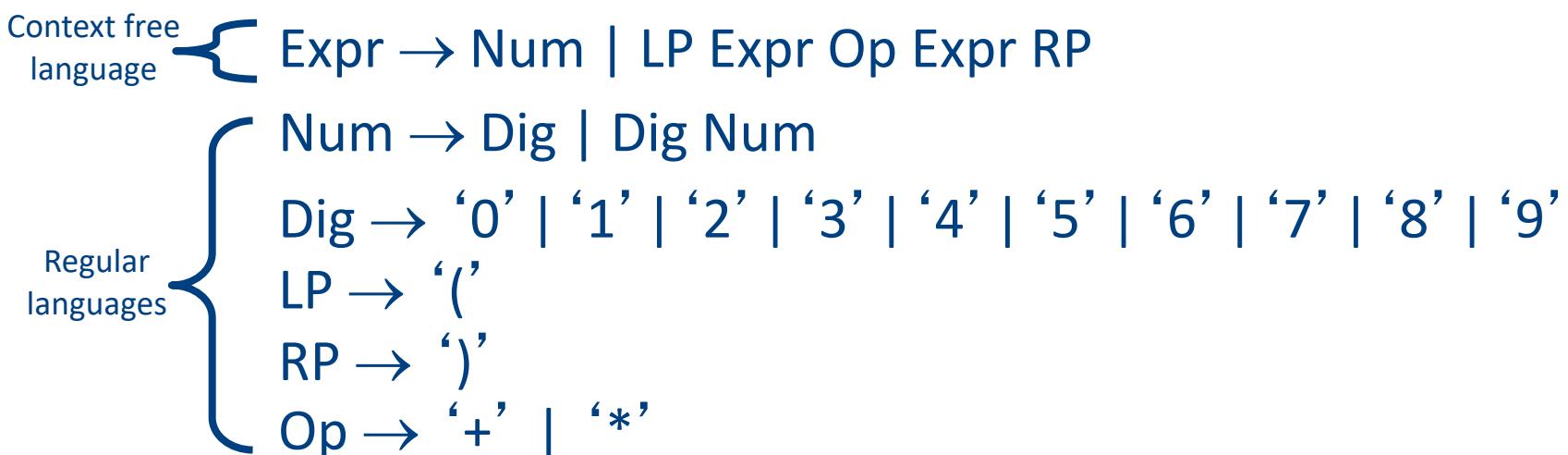
RP → ')'

Op → '+' | '\*'

(    (    23    +    7    )    \*    19    )

# What does Lexical Analysis do?

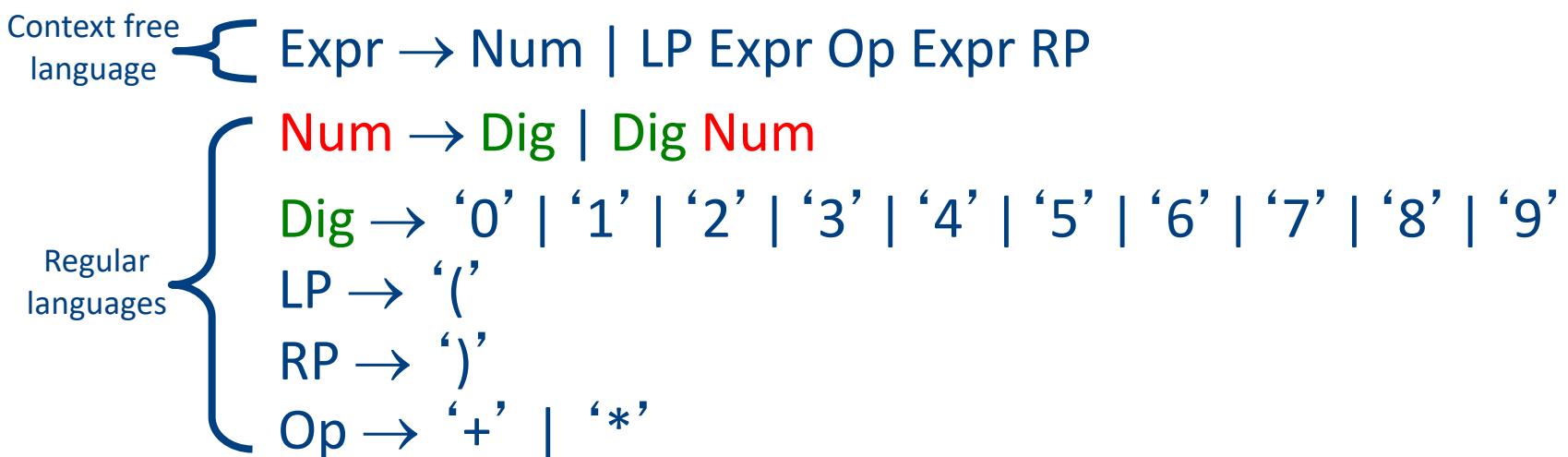
- Language: fully parenthesized expressions



(    (    23    +    7    )    \*    19    )

# What does Lexical Analysis do?

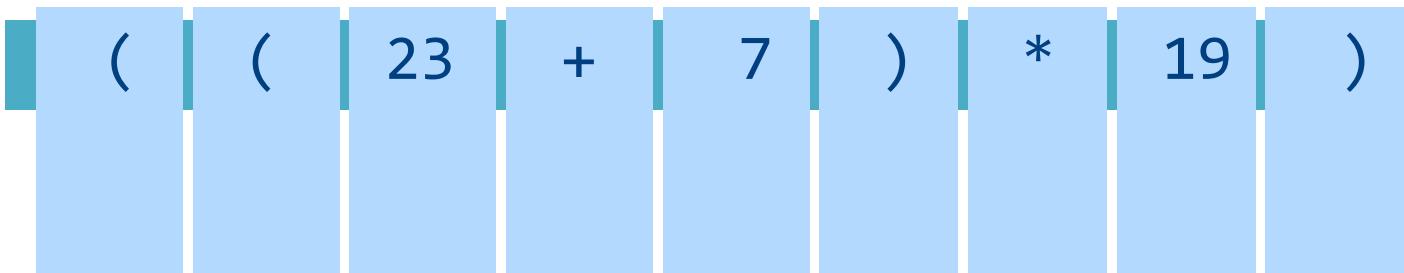
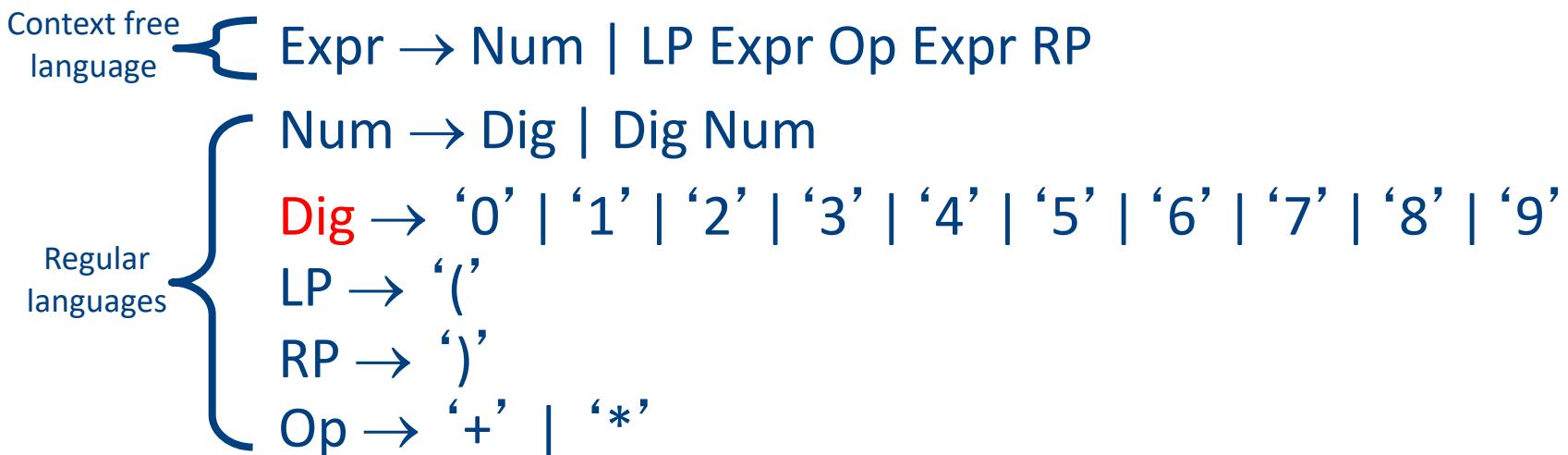
- Language: fully parenthesized expressions



( ( 23 + 7 ) \* 19 )

# What does Lexical Analysis do?

- Language: fully parenthesized expressions

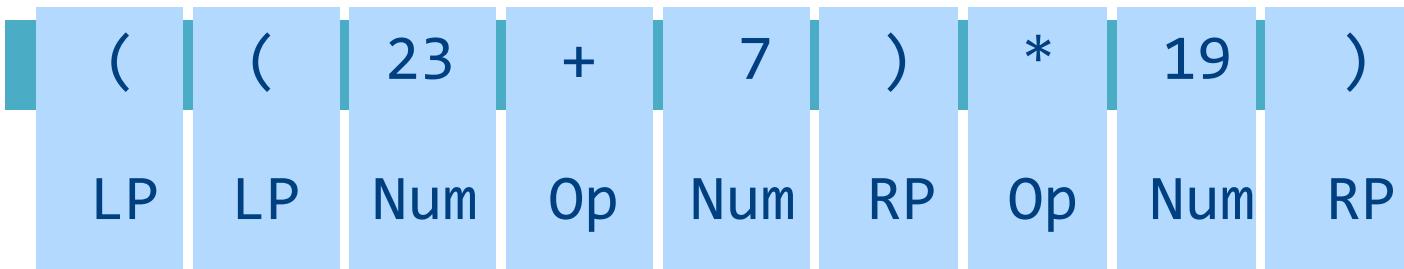


# What does Lexical Analysis do?

- Language: fully parenthesized expressions

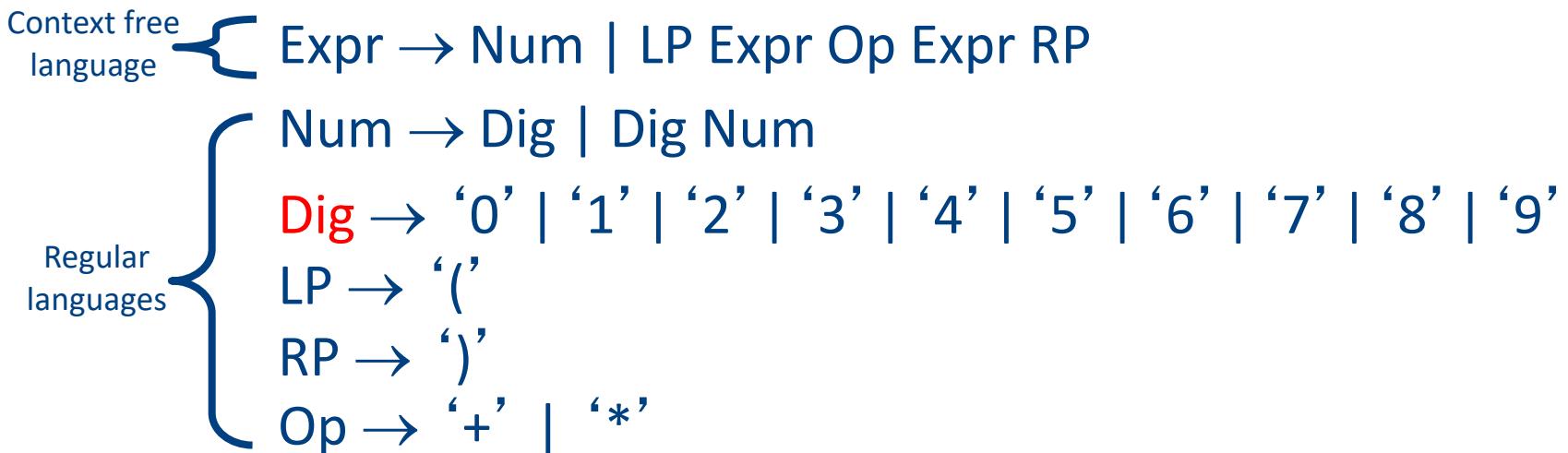
Context free language       $\left\{ \begin{array}{l} \text{Expr} \rightarrow \text{Num} \mid \text{LP Expr Op Expr RP} \\ \\ \text{Num} \rightarrow \text{Dig} \mid \text{Dig Num} \\ \text{Dig} \rightarrow '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \\ \text{LP} \rightarrow '(' \\ \text{RP} \rightarrow ')' \\ \text{Op} \rightarrow '+' \mid '*' \end{array} \right.$

Regular languages



# What does Lexical Analysis do?

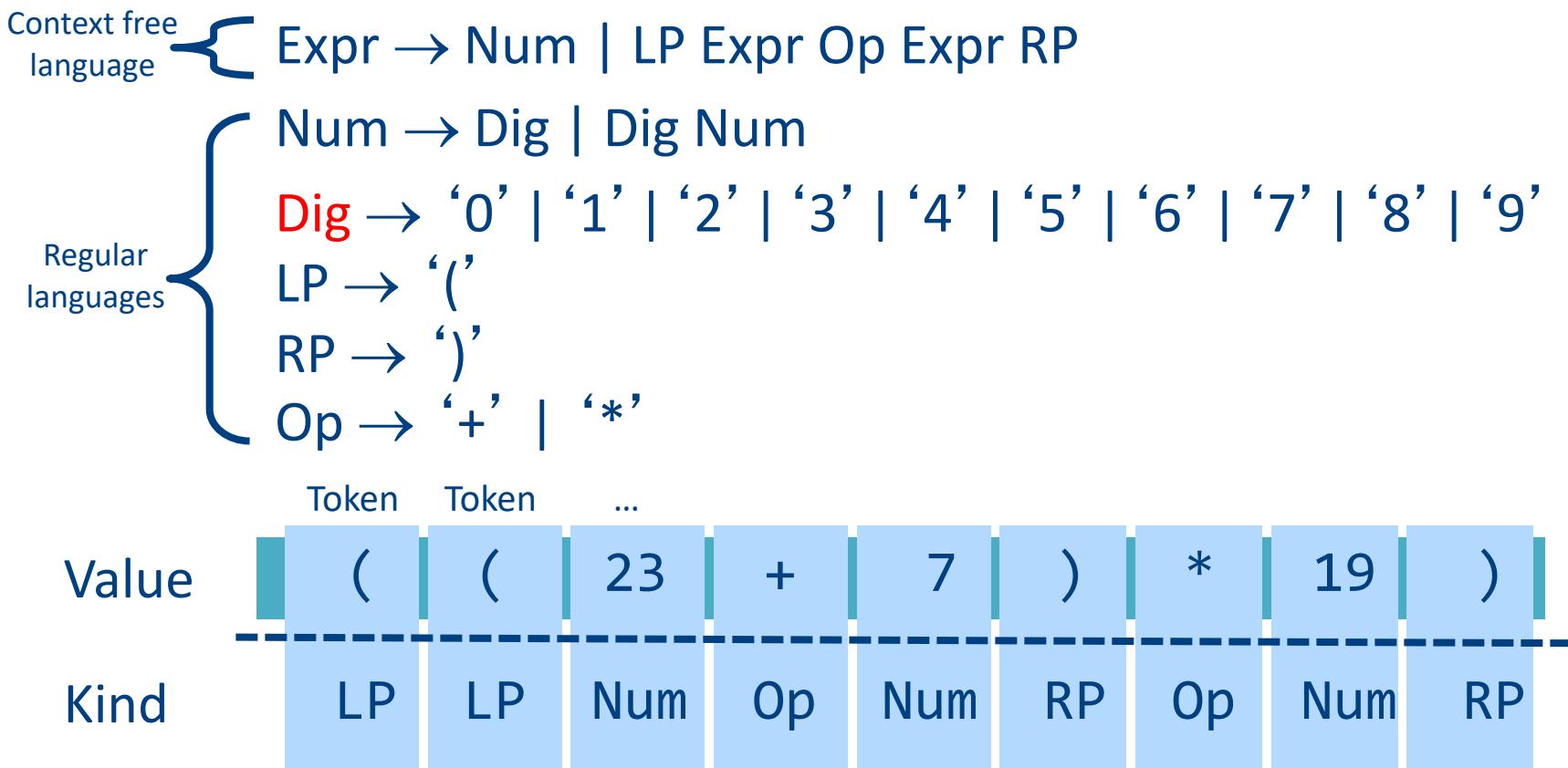
- Language: fully parenthesized expressions



Value	(	(	23	+	7	)	*	19	)
Kind	LP	LP	Num	Op	Num	RP	Op	Num	RP

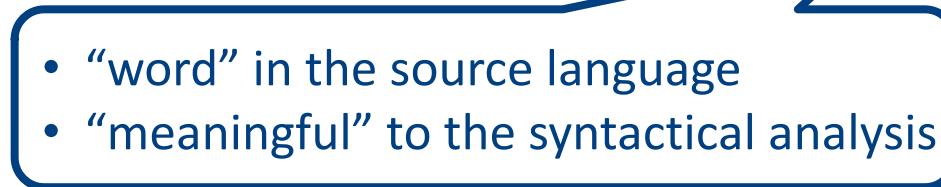
# What does Lexical Analysis do?

- Language: fully parenthesized expressions



# What does Lexical Analysis do?

- Partitions the input into stream of **tokens**
  - Numbers
  - Identifiers
  - Keywords
  - Punctuation
- Usually represented as (kind, value) pairs
  - (Num, 23)
  - (Op, '\*')



- “word” in the source language
- “meaningful” to the syntactical analysis

# From scanning to parsing

*program text*

$((23 + 7) * x)$

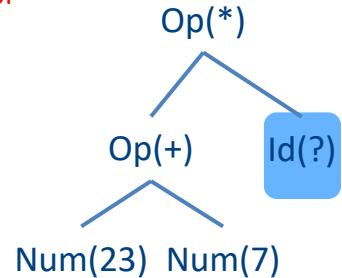
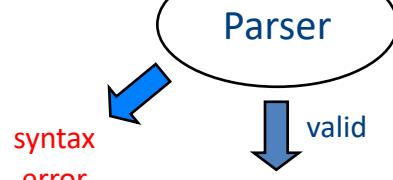


*token stream*

(	(	23	+	7	)	*	?	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:

$\text{Expr} \rightarrow \dots \mid \text{Id}$   
 $\text{Id} \rightarrow 'a' \mid \dots \mid 'z'$



*Abstract Syntax Tree*

# Why Lexical Analysis?

- Well, not strictly necessary, but ...
  - Regular languages  $\subseteq$  Context-Free languages
- Simplifies the syntax analysis (parsing)
  - And language definition
- Modularity
- Reusability
- Efficiency

# Lecture goals

- Understand role & place of lexical analysis
- Lexical analysis theory
- Using program generating tools

# Lecture Outline

- ✓ Role & place of lexical analysis
- What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

# What is a token? (Intuitively)

- A “word” in the source language
  - Anything that should appear in the input to syntax analysis
    - Identifiers
    - Values
    - Language keywords
- Usually, represented as a pair of (kind, value)

# Example Tokens

Type	Examples
ID	foo, n_14, last
NUM	73, 00, 517, 082
REAL	66.1, .5, 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

# Example Non Tokens

Type	Examples
comment	<code>/* ignored */</code>
preprocessor directive	<code>#include &lt;foo.h&gt;</code>
	<code>#define NUMS 5.6</code>
macro	<code>NUMS</code>
whitespace	<code>\t, \n, \b, ‘ ‘</code>

# Some basic terminology

- **Lexeme** (aka symbol) - a series of letters separated from the rest of the program according to a convention (space, semi-column, comma, etc.)
- **Pattern** - a rule specifying a set of strings.  
Example: “an identifier is a string that starts with a letter and continues with letters and digits”
  - (Usually) a regular expression
- **Token** - a pair of (pattern, attributes)

# Example

```
void match0(char *s) /* find a zero */  
{  
    if (!strncmp(s, "0.0", 3))  
        return 0.0 ;  
}
```

```
VOID ID(match0) LPAREN CHAR DEREF ID(s) RPAREN  
LBRACE  
IF LPAREN NOT ID(strncmp) LPAREN ID(s) COMMA STRING(0.0)  
COMMA NUM(3) RPAREN RPAREN  
RETURN REAL(0.0) SEMI  
RBRACE  
EOF
```

# Example Non Tokens

Type	Examples
comment	<code>/* ignored */</code>
preprocessor directive	<code>#include &lt;foo.h&gt;</code>
	<code>#define NUMS 5.6</code>
macro	<code>NUMS</code>
whitespace	<code>\t, \n, \b, ‘ ‘</code>

- Lexemes that are recognized but get consumed rather than transmitted to parser
  - if
  - i/\*comment\*/f

# Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

# How can we define tokens?

- Keywords – easy!
  - if, then, else, for, while, ...
- Identifiers?
- Numerical Values?
- Strings?
- Characterize unbounded sets of values using a bounded description?

# Regular languages

- Formal languages
  - $\Sigma$  = finite set of letters
  - Word = sequence of letter
  - Language = set of words
- Regular languages defined equivalently by
  - Regular expressions
  - Finite-state automata

# Common format for reg-exp

Basic Patterns	Matching
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
^x	Any character except x
Repetition Operators	
R?	An R or nothing (=optionally an R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R
Composition Operators	
R1R2	An R1 followed by R2
R1 R2	Either an R1 or R2
Grouping	
(R)	R itself

# Examples

- $ab^*|cd? =$
- $(a|b)^* =$
- $(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)^* =$

# Escape characters

- What is the expression for one or more + symbols?
  - $(+)^+$  won't work
  - $(\backslash +)^+$  will
- backslash \ before an operator turns it to standard character
  - $\backslash^*$ ,  $\backslash?$ ,  $\backslash+$ ,  $a\backslash(b\backslash +\backslash^*)$ ,  $(a\backslash(b\backslash +\backslash^*))^+$ , ...
- backslash double quotes surrounds text
  - “ $a(b^+*)$ ”, “ $a(b^+*)^+$

# End of Lecture 1



# Shorthands

- Use names for expressions
  - letter = a | b | ... | z | A | B | ... | Z
  - letter\_ = letter | \_
  - digit = 0 | 1 | 2 | ... | 9
  - id = letter\_ (letter\_ | digit)\*
- Use hyphen to denote a range
  - letter = a-z | A-Z
  - digit = 0-9

# Examples

- if = if
- then = then
- relop = < | > | <= | >= | = | <>
- digit = 0-9
- digits = digit+

# Example

- A number is

```
number = ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
        ( ε | \. ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
          ( ε | E ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
        )
)
```

- Using shorthands it can be written as

```
number = digits (ε | \.digits (ε | E (ε|+|-) digits ) )
```

# Exercise 1 - Question

- Language of rational numbers in decimal representation (no leading, ending zeros)
  - 0
  - 123.757
  - .933333
  - Not 007
  - Not 0.30

# Exercise 1 - Answer

- Language of rational numbers in decimal representation (no leading, ending zeros)
  - $\text{Digit} = 1|2|\dots|9$
  - $\text{Digit0} = 0|\text{Digit}$
  - $\text{Num} = \text{Digit Digit0}^*$
  - $\text{Frac} = \text{Digit0}^* \text{Digit}$
  - $\text{Pos} = \text{Num} \mid \backslash.\text{Frac} \mid 0\backslash.\text{Frac} \mid \text{Num}\backslash.\text{Frac}$
  - $\text{PosNeg} = (\epsilon|{-})\text{Pos}$
  - $\text{R} = 0 \mid \text{PosNeg}$

# Exercise 2 - Question

- Equal number of opening and closing parenthesis:  $[^n]^n = [], [[]], [[[]], \dots$

# Exercise 2 - Answer

- Equal number of opening and closing parenthesis:  $[^n]^n = [], [[]], [[[]], \dots$
- Not regular
- Context-free
- Grammar:  $S ::= [] \mid [S]$

# Challenge: Ambiguity

- If = `if`
- Id = `Letter (Letter | Digit)*`
- “`if`” is a valid identifiers... what should it be?
- “`iffy`” is also a valid identifier
- Solution
  - Longest matching token
  - Break ties using order of definitions...
    - Keywords should appear before identifiers

# Creating a lexical analyzer

- Given a list of token definitions (pattern name, regex), write a program such that
  - Input: String to be analyzed
  - Output: List of tokens
- How do we build an analyzer?

# Building a Scanner – Take I

- Input: String
- Output: Sequence of tokens

# Building a Scanner – Take I

```
Token nextToken()
{
    char c ;
    loop: c = getchar();
    switch (c){
        case ` `: goto loop ;
        case `;`: return SemiColumn;
        case `+`:
            c = getchar() ;
            switch (c) {
                case `+`: return PlusPlus ;
                case '=': return PlusEqual;
                default: ungetc(c); return Plus;
            };
        case `<`: ...
        case `w` : ...
    }
}
```

# There must be a better way!



# A better way

- **Automatically generate** a scanner
- Define tokens using regular expressions
- Use **finite-state automata** for detection

# A better way

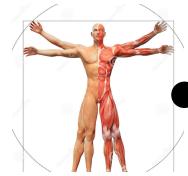


# Reg-exp vs. automata

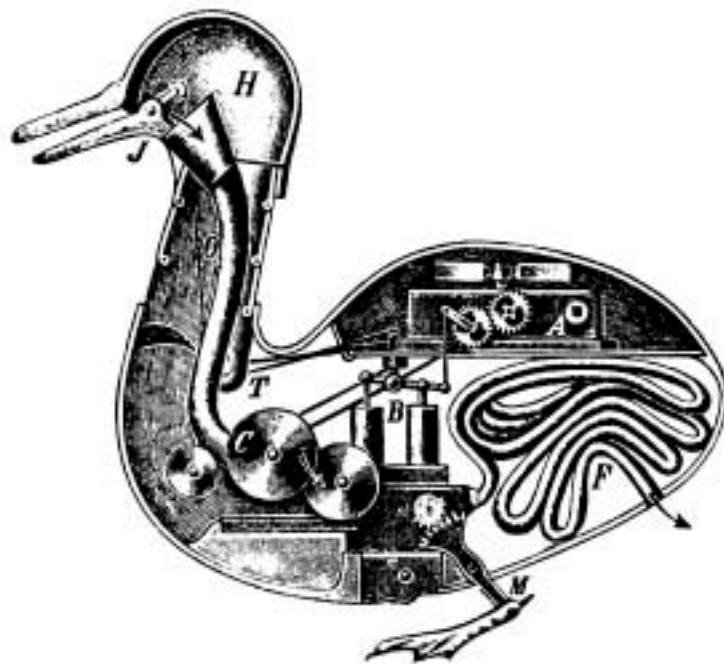
- Regular expressions are declarative
  - Good for humans
  - Not “executable”
- Automata are operative
  - Define an *algorithm* for deciding whether a given word is in a regular language
  - Not a natural notation for humans

# Overview

- Define tokens using regular expressions
- Construct a nondeterministic finite-state automaton (NFA) from regular expression
- Determinize the NFA into a deterministic finite-state automaton (DFA)
- DFA can be directly used to identify tokens



# Automata theory: a bird's-eye view

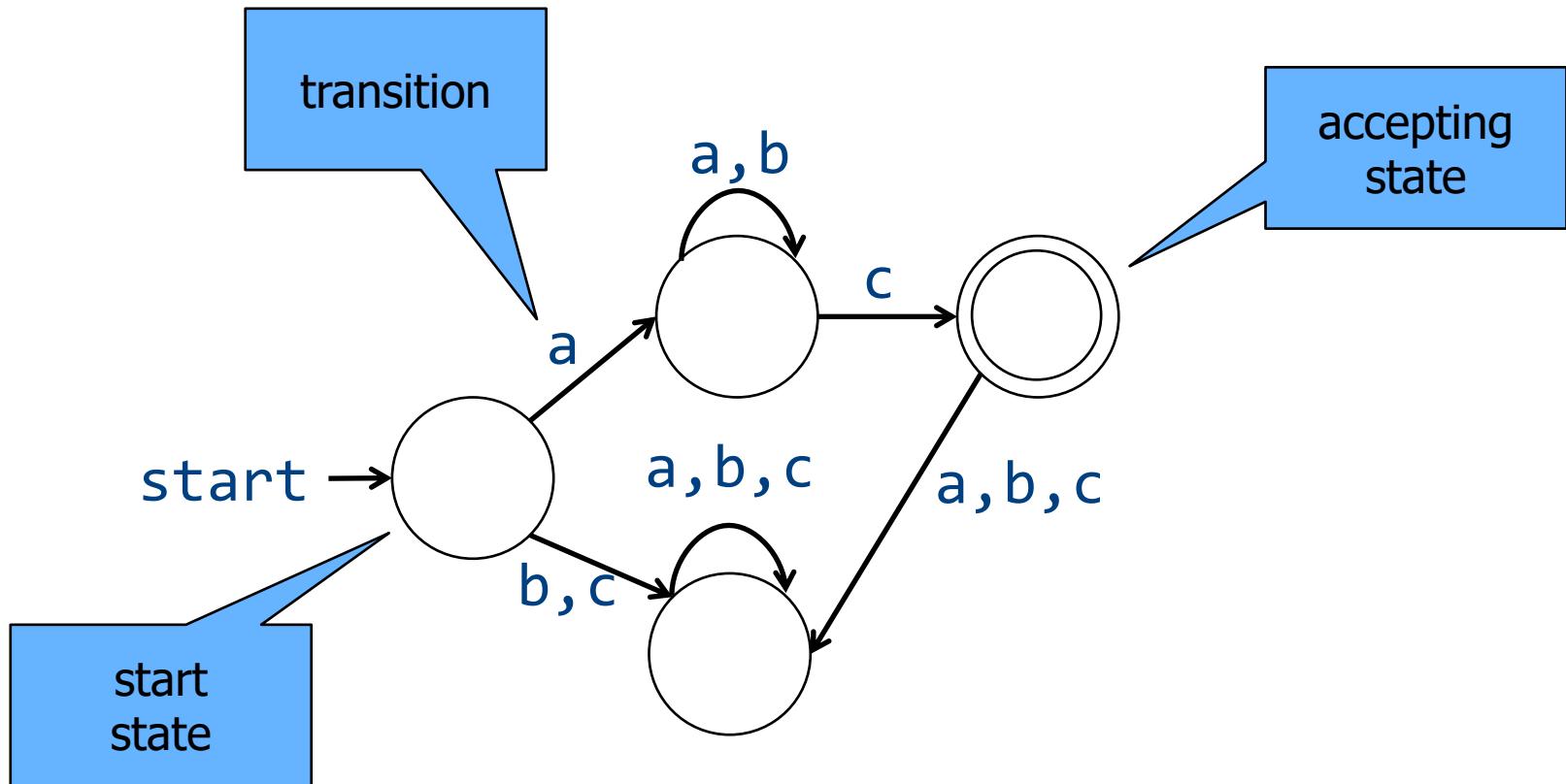


# Deterministic Automata (DFA)

- $M = (\Sigma, Q, \delta, q_0, F)$ 
  - $\Sigma$  - alphabet
  - $Q$  – finite set of state
  - $q_0 \in Q$  – initial state
  - $F \subseteq Q$  – final states
  - $\delta : Q \times \Sigma \rightarrow Q$  - transition function
- For a word  $w$ ,  $M$  reach some state  $x$ 
  - $M$  accepts  $w$  if  $x \in F$

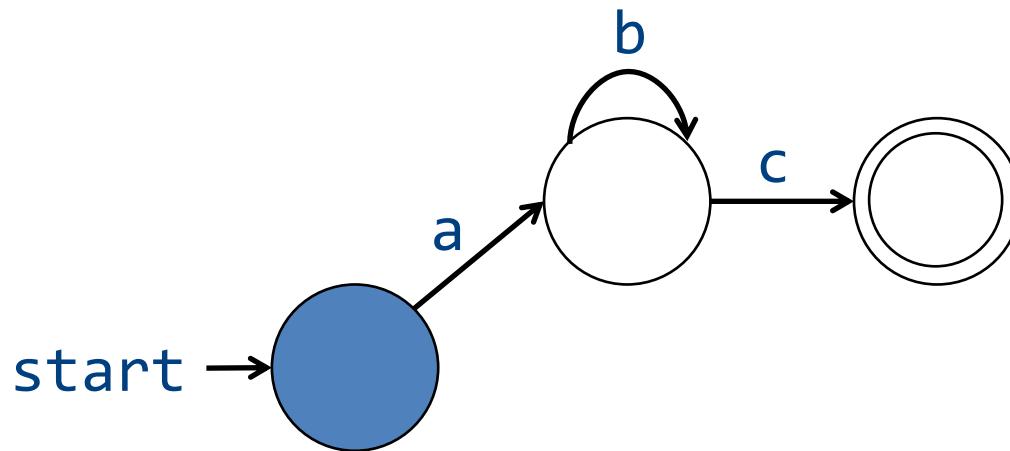
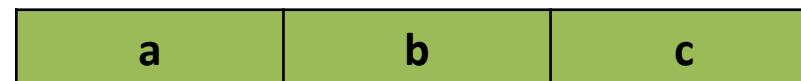
# DFA in pictures

- An automaton is defined by states and transitions



# Accepting Words

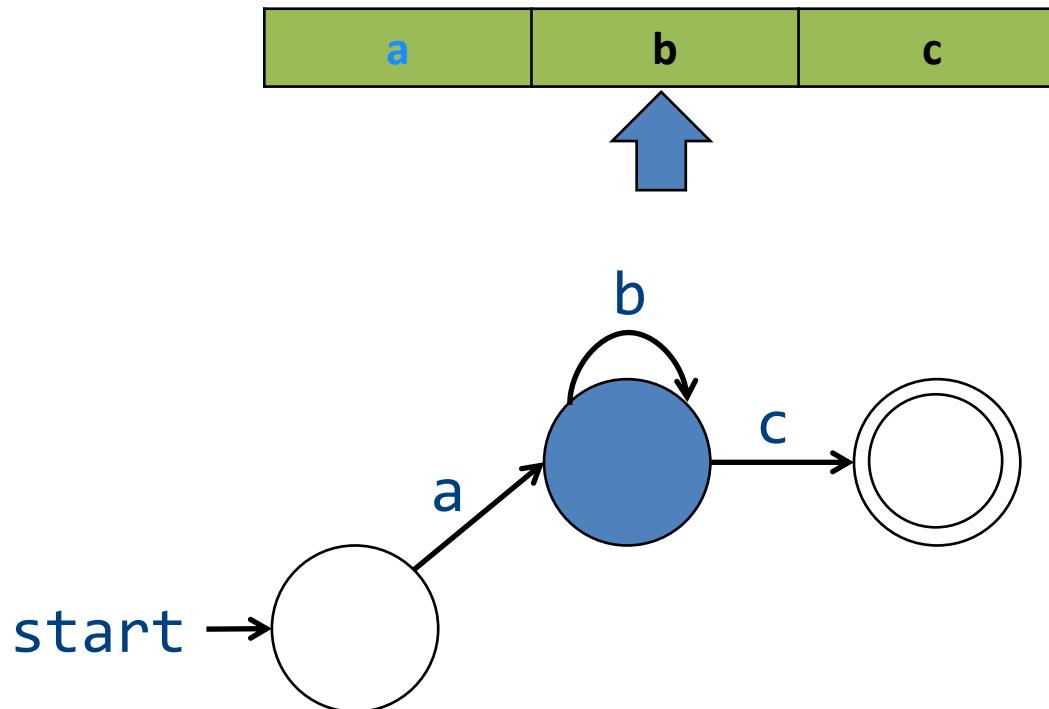
- Words are read left-to-right



- Missing transition = non-acceptance
  - “Stuck state”

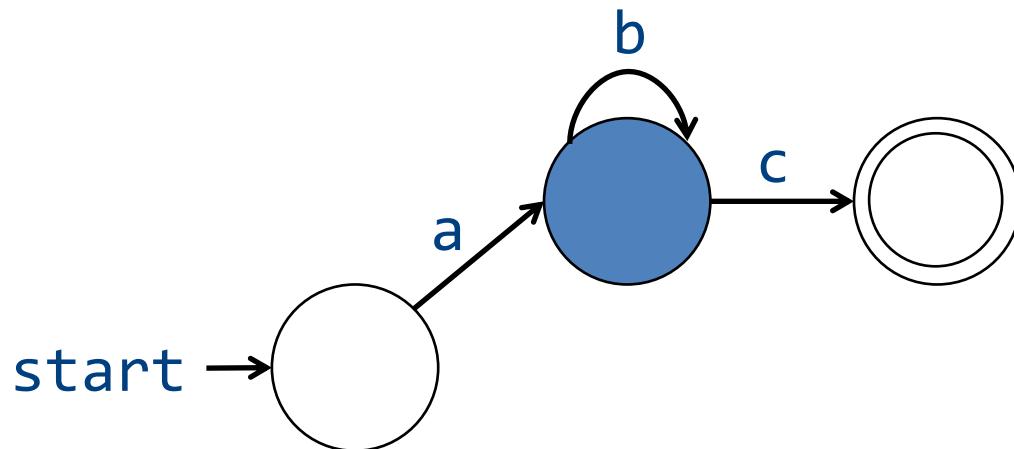
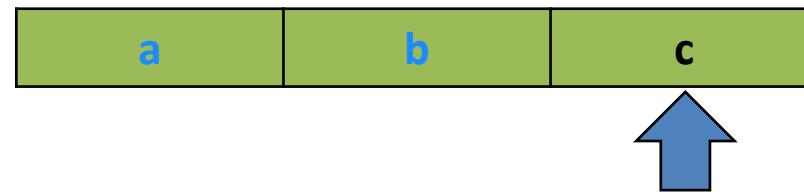
# Accepting Words

- Words are read left-to-right



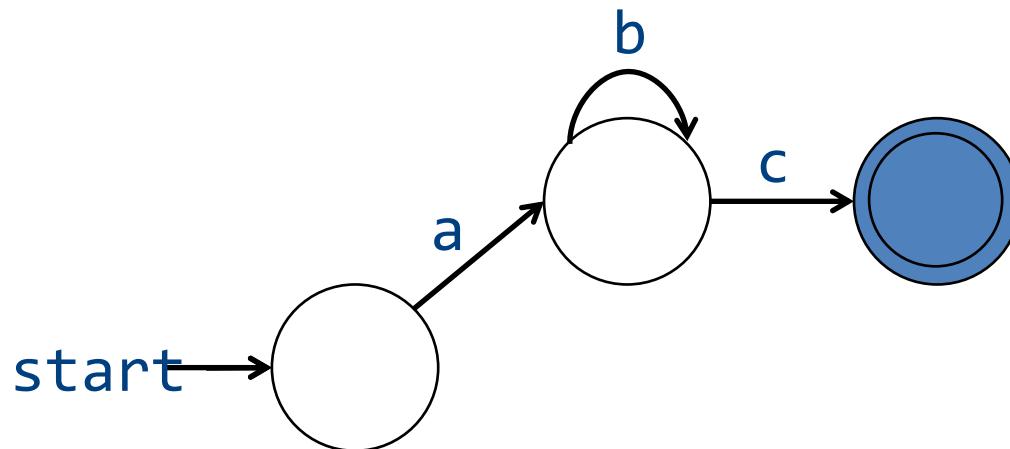
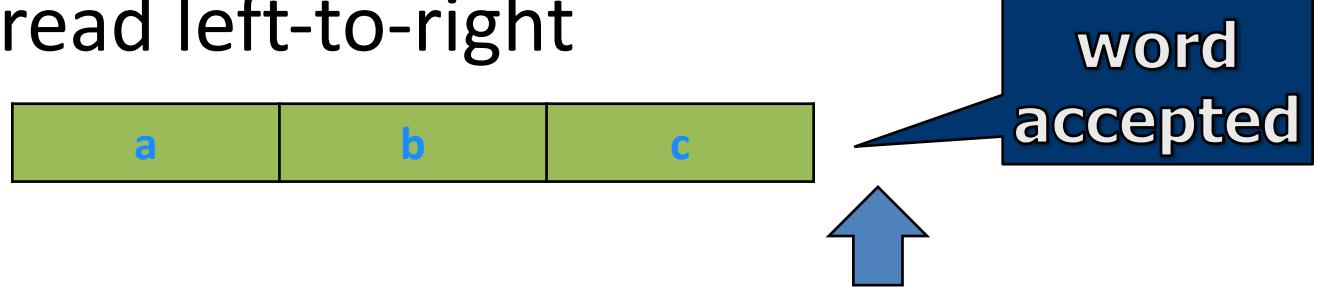
# Accepting Words

- Words are read left-to-right



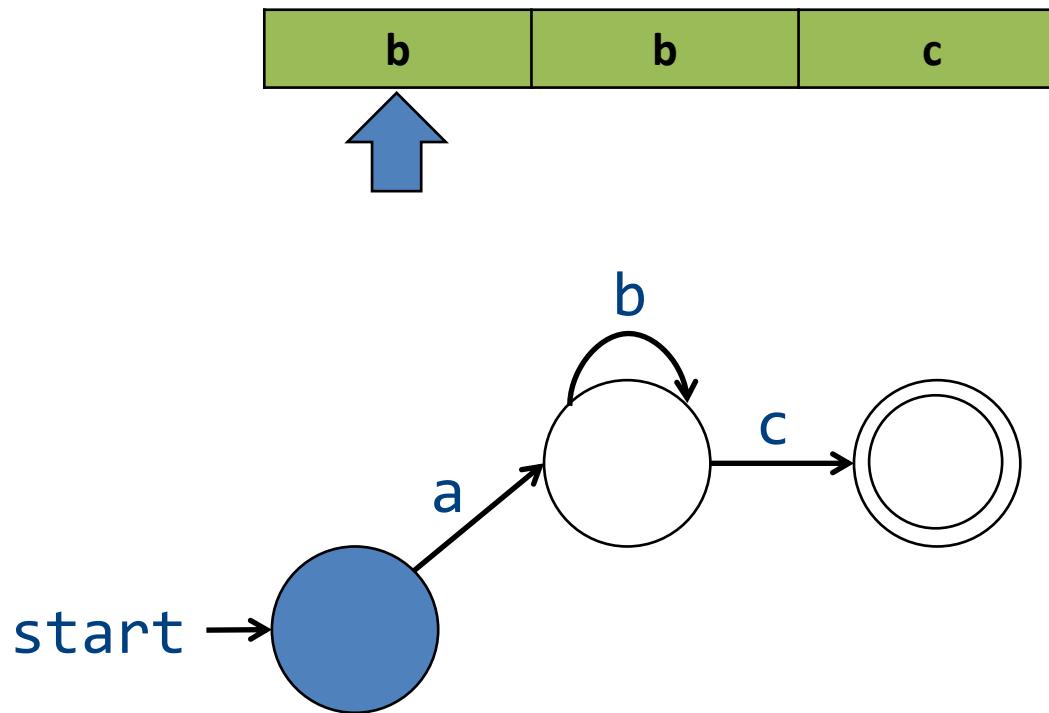
# Accepting Words

- Words are read left-to-right



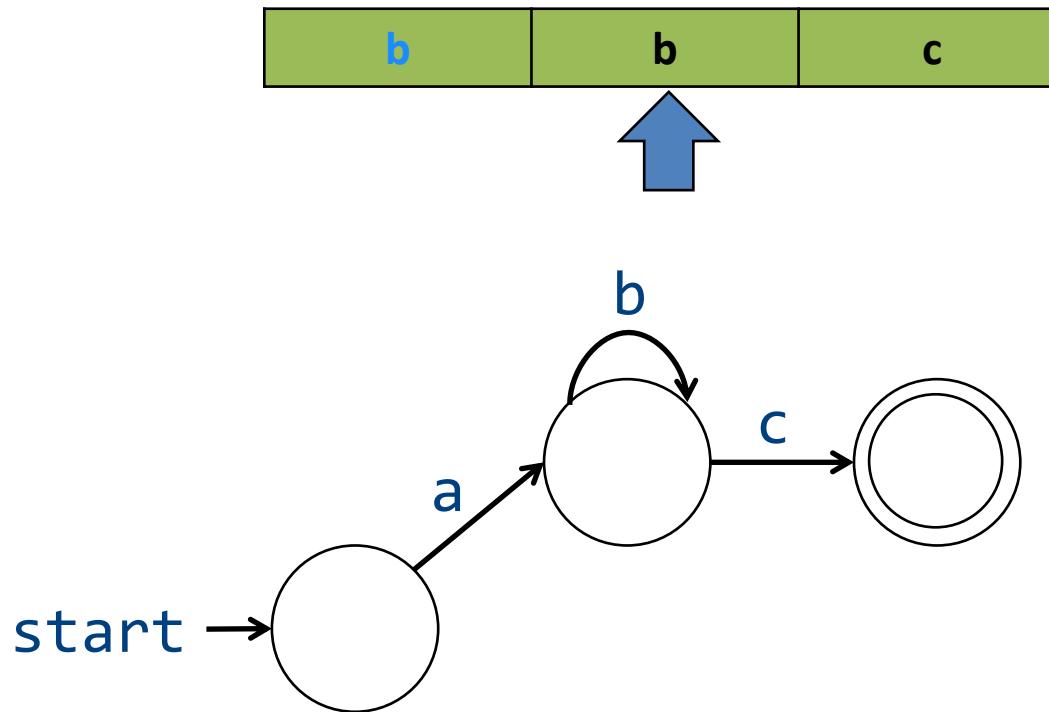
# Rejecting Words

- Words are read left-to-right



# Rejecting Words

- Missing transition means non-acceptance

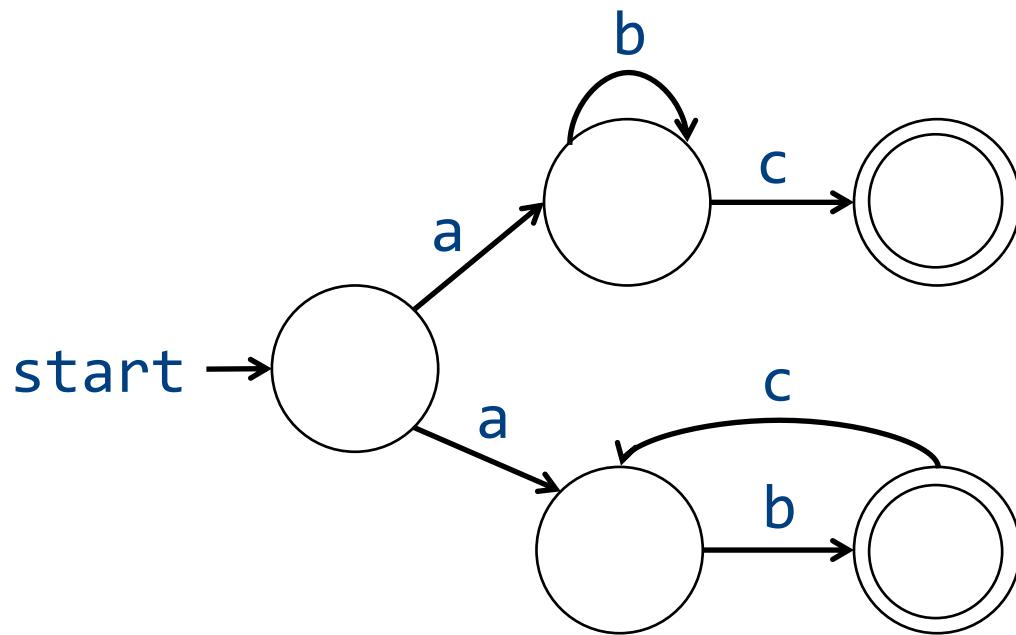


# Non-deterministic Automata (NFA)

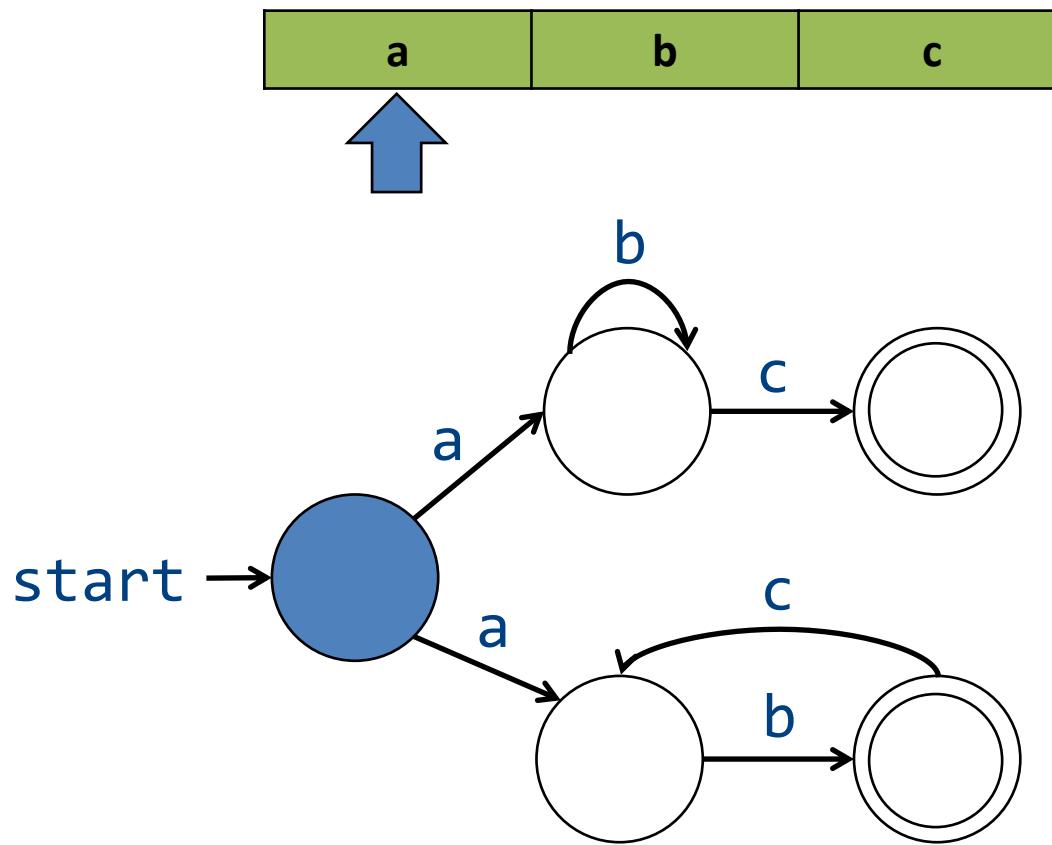
- $M = (\Sigma, Q, \delta, q_0, F)$ 
  - $\Sigma$  - alphabet
  - $Q$  – finite set of state
  - $q_0 \in Q$  – initial state
  - $F \subseteq Q$  – final states
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$  - transition function
    - DFA:  $\delta : Q \times \Sigma \rightarrow Q$
- For a word  $w$ ,  $M$  can reach a number of states  $X$ 
  - $M$  accepts  $w$  if  $X \cap F \neq \emptyset$ 
    - Possible:  $X = \emptyset$
- Possible  $\epsilon$ -transitions

# NFA

- Allow multiple transitions from given state labeled by same letter

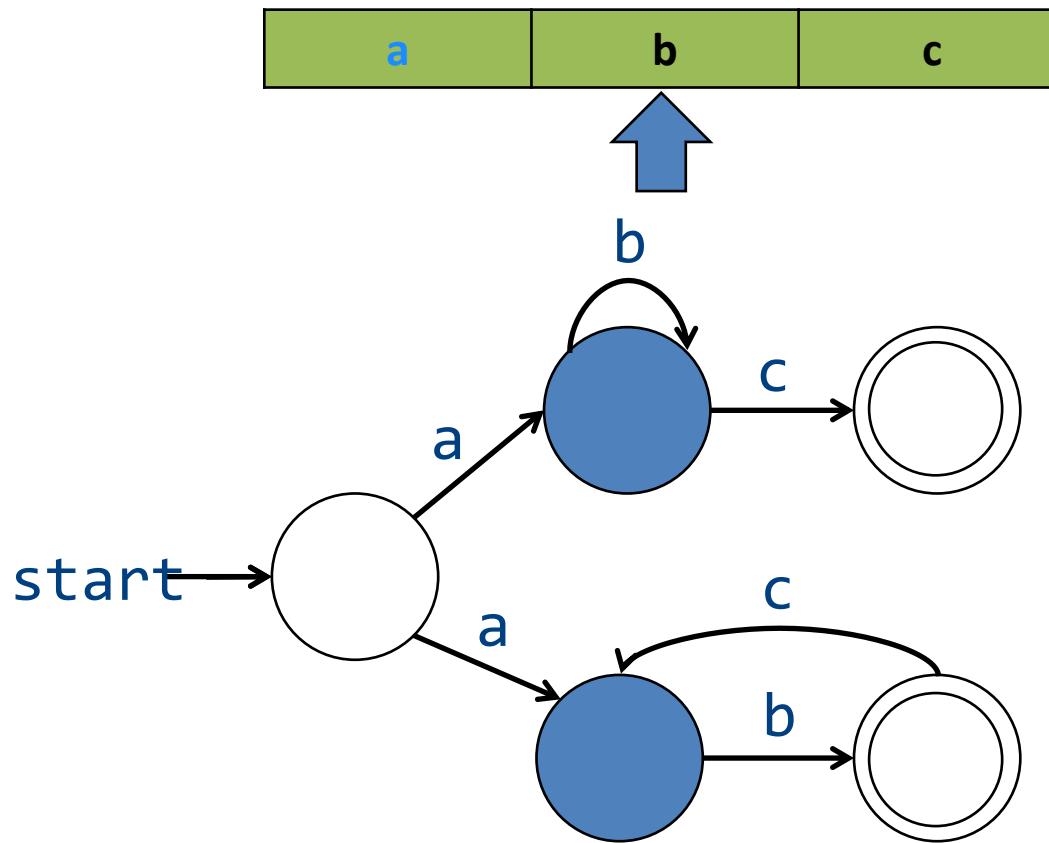


# Accepting words

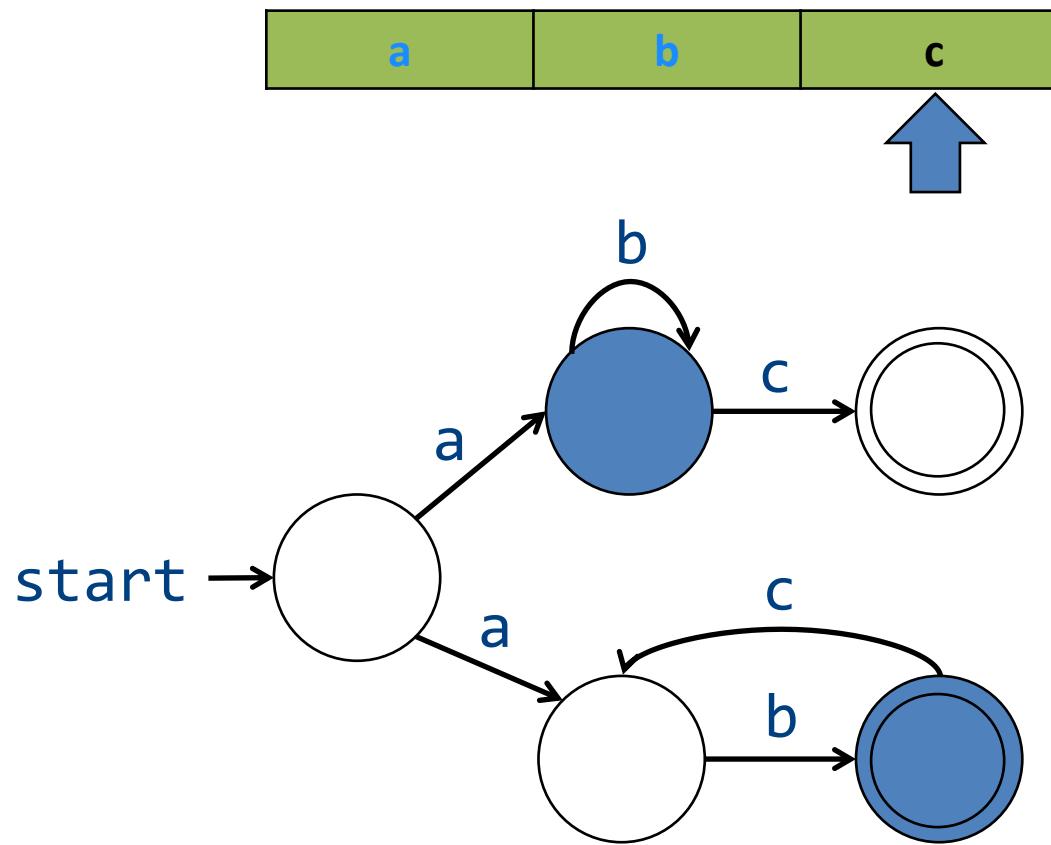


# Accepting words

- Maintain set of states

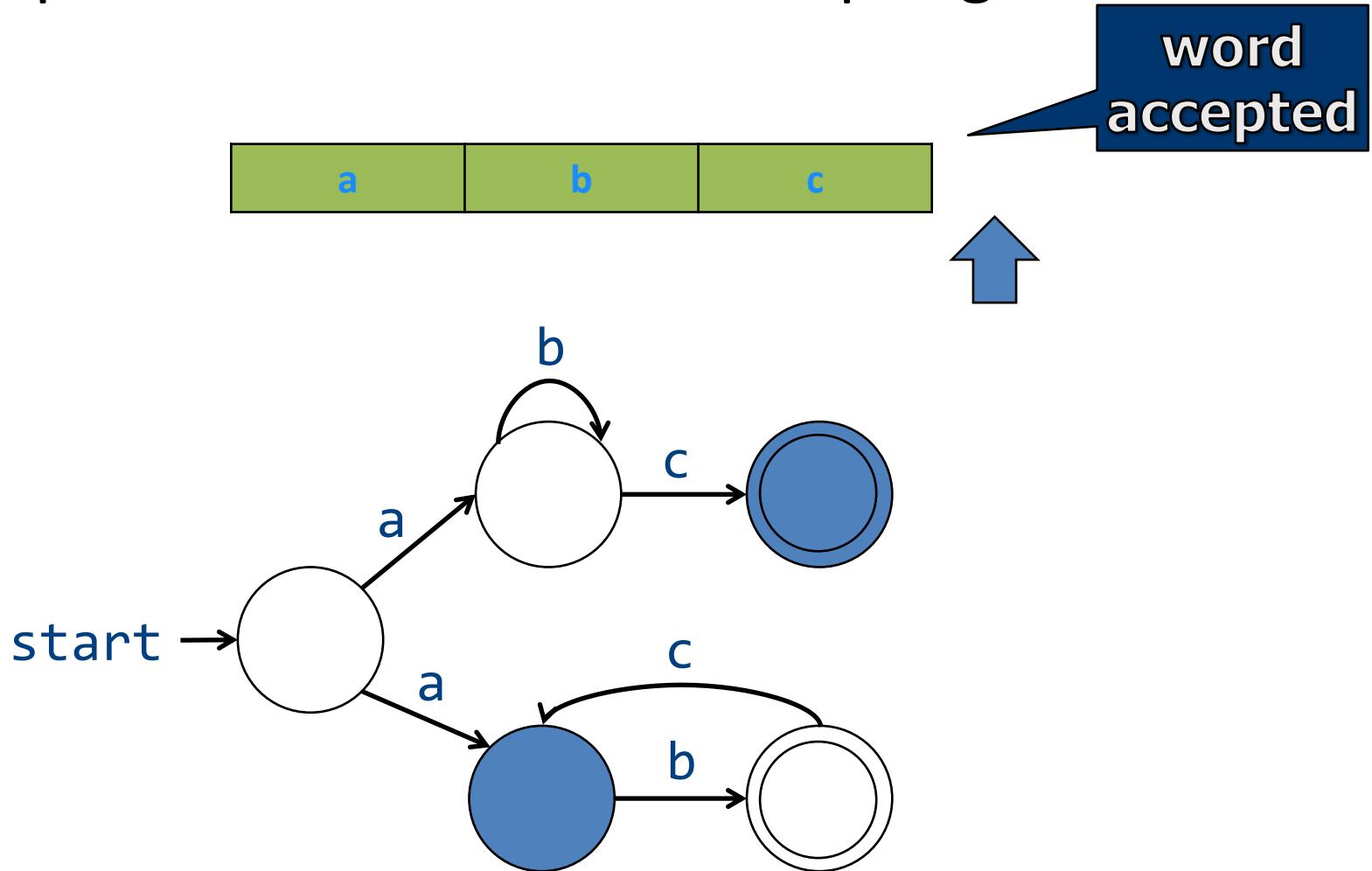


# Accepting words



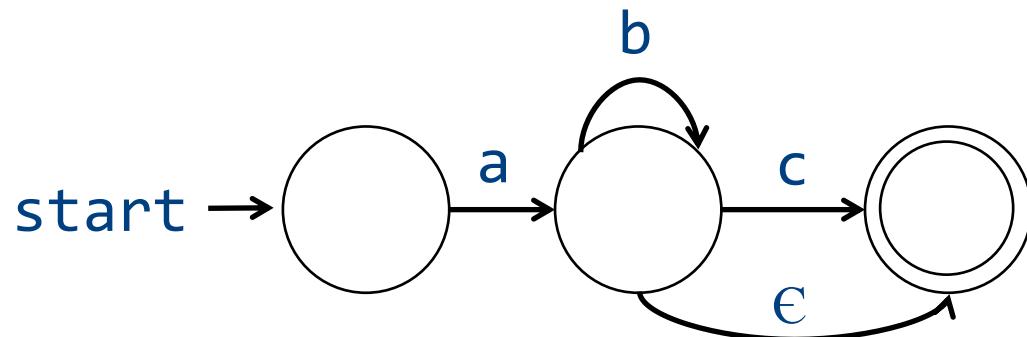
# Accepting words

- Accept word if reached an accepting state

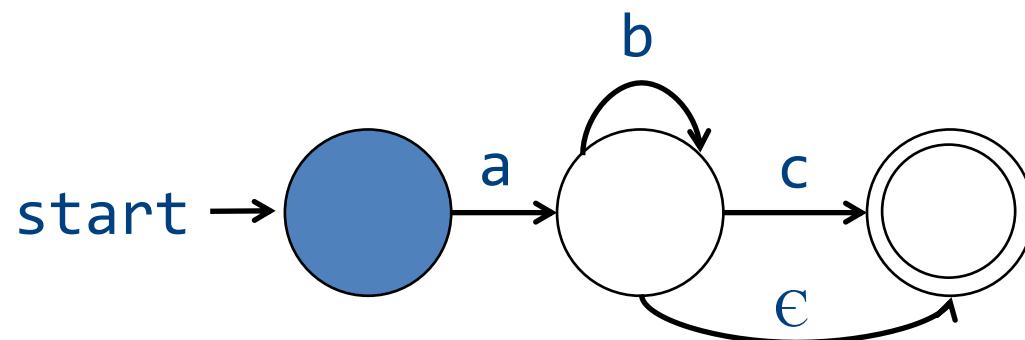
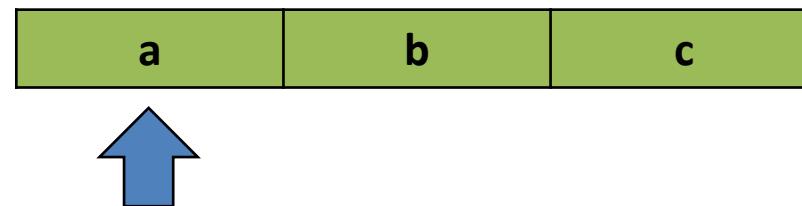


# NFA+ $\epsilon$ automata

- $\epsilon$  transitions can “fire” without reading the input

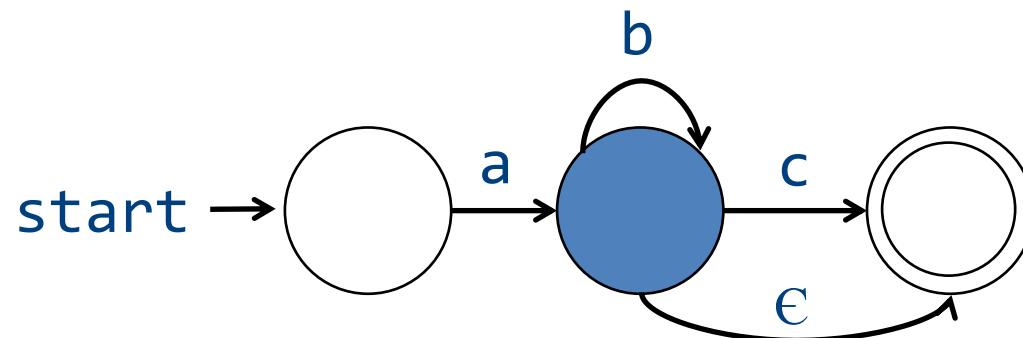
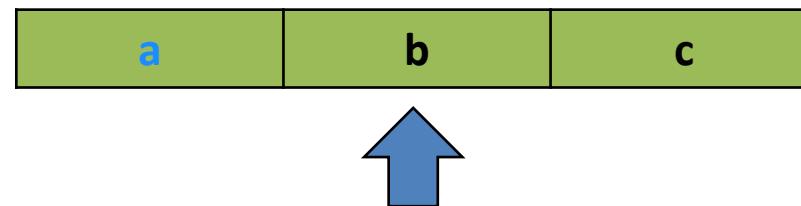


# NFA+ $\epsilon$ run example

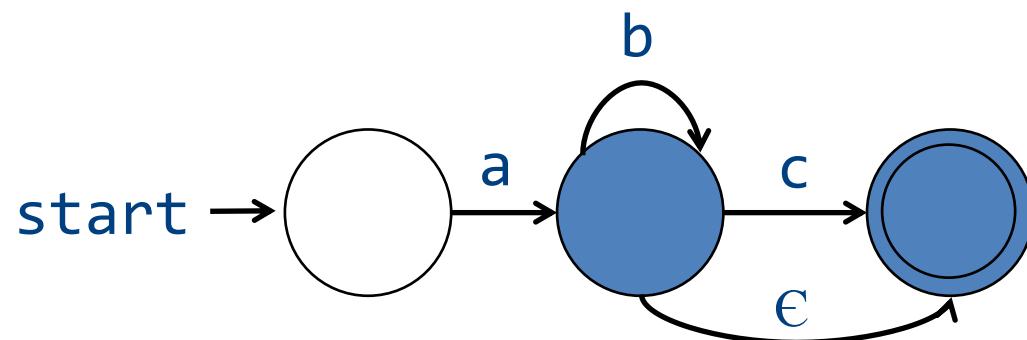
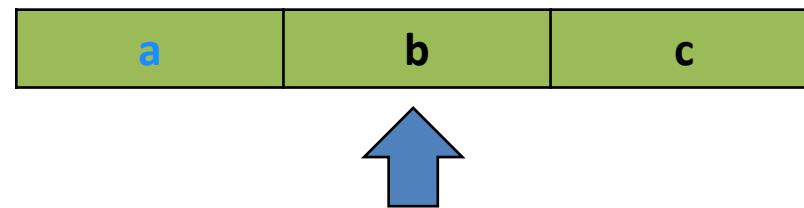


# NFA+ $\epsilon$ run example

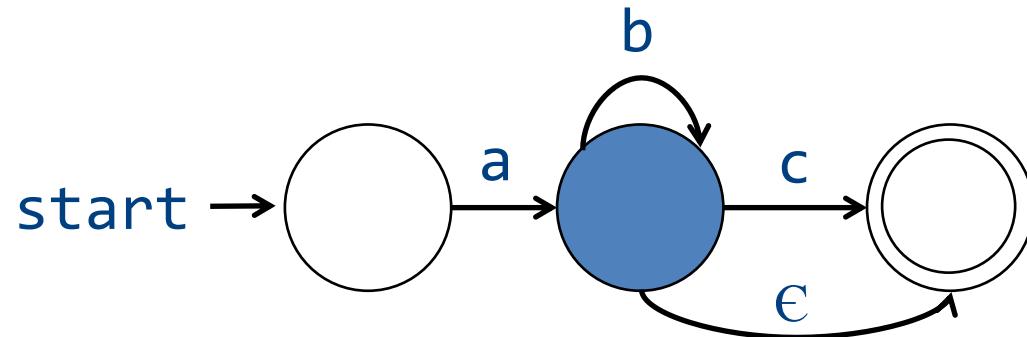
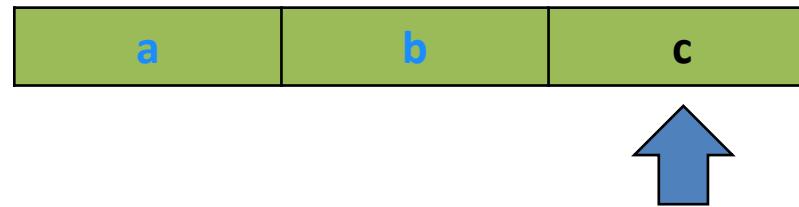
- Now  $\epsilon$  transition can non-deterministically take place



# NFA+ $\epsilon$ run example

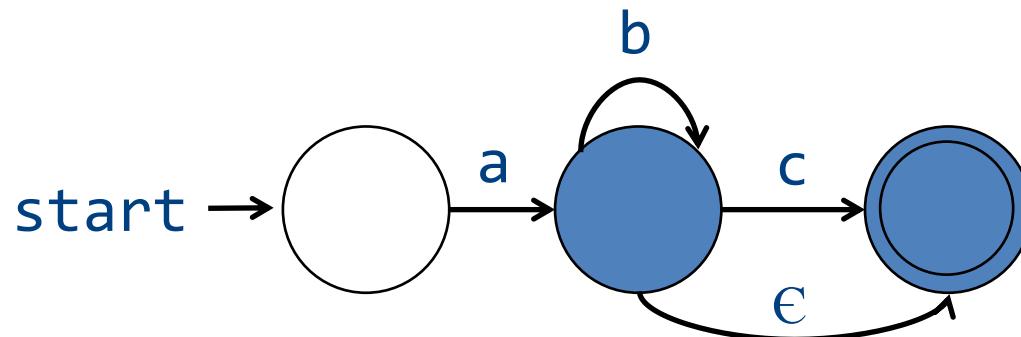
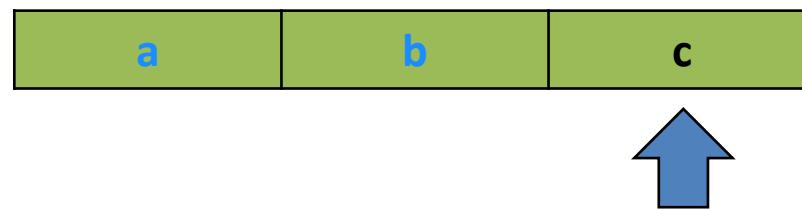


# NFA+ $\epsilon$ run example



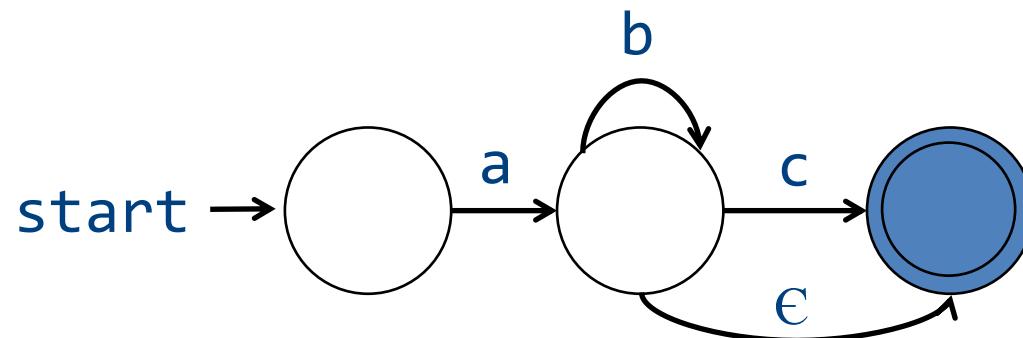
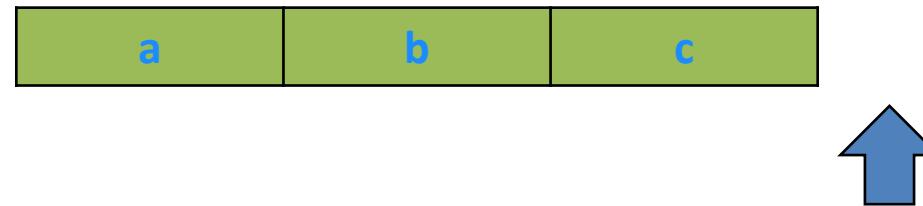
# NFA+ $\epsilon$ run example

- $\epsilon$  transitions can “fire” without reading the input



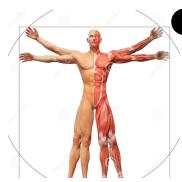
# NFA+ $\epsilon$ run example

- Word accepted



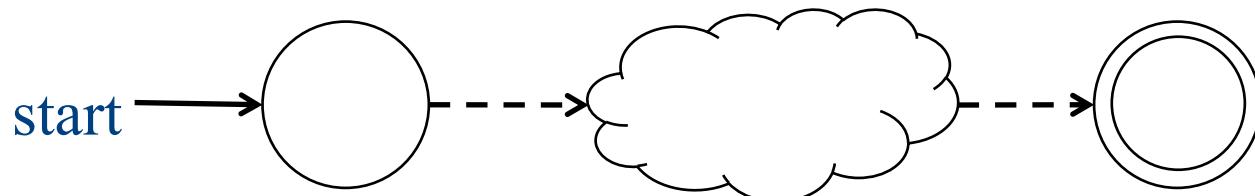
# From regular expressions to NFA

- Step 1: assign expression names and obtain pure regular expressions  $R_1 \dots R_m$
- Step 2: construct an NFA  $M_i$  for each regular expression  $R_i$
- Step 3: combine all  $M_i$  into a single NFA
- *Ambiguity resolution: prefer longest accepting word*



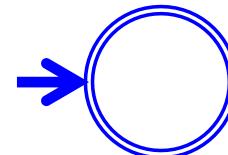
# From reg. exp. to automata

- Theorem: *there is an algorithm to build an NFA+ $\epsilon$  automaton for any regular expression*
- Proof: *by induction on the structure of the regular expression*



# Basic constructs

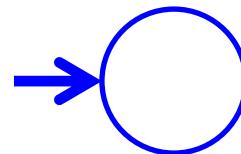
$R = \varepsilon$



$R = a$

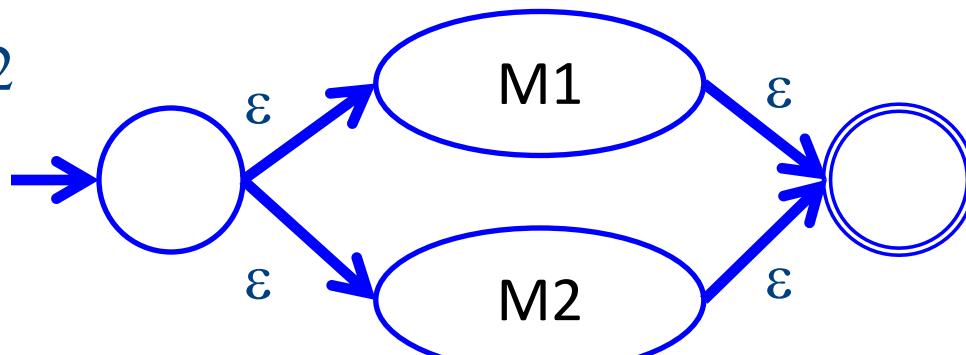


$R = \phi$

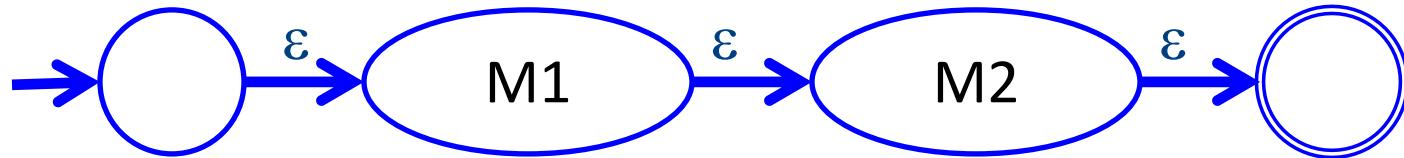


# Composition

$$R = R_1 \mid R_2$$

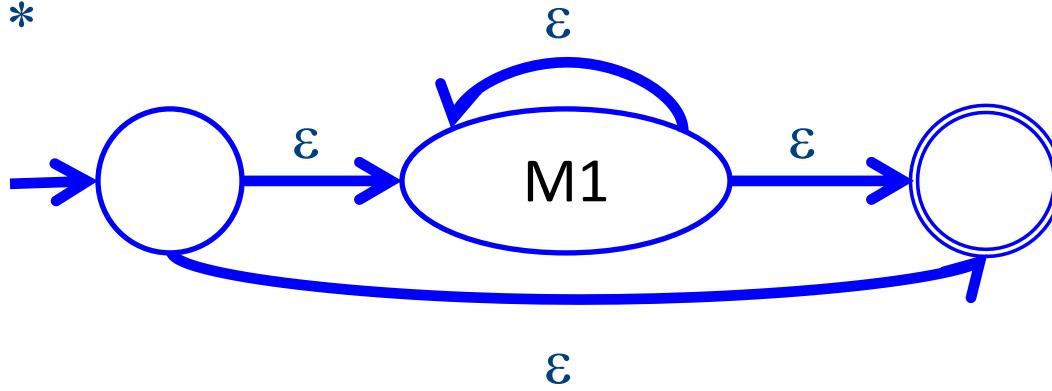


$$R = R_1 R_2$$

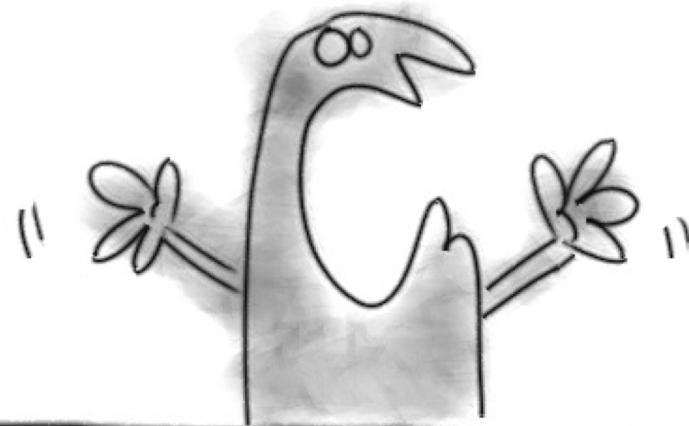


# Repetition

$R = R1^*$



# Now What?!!



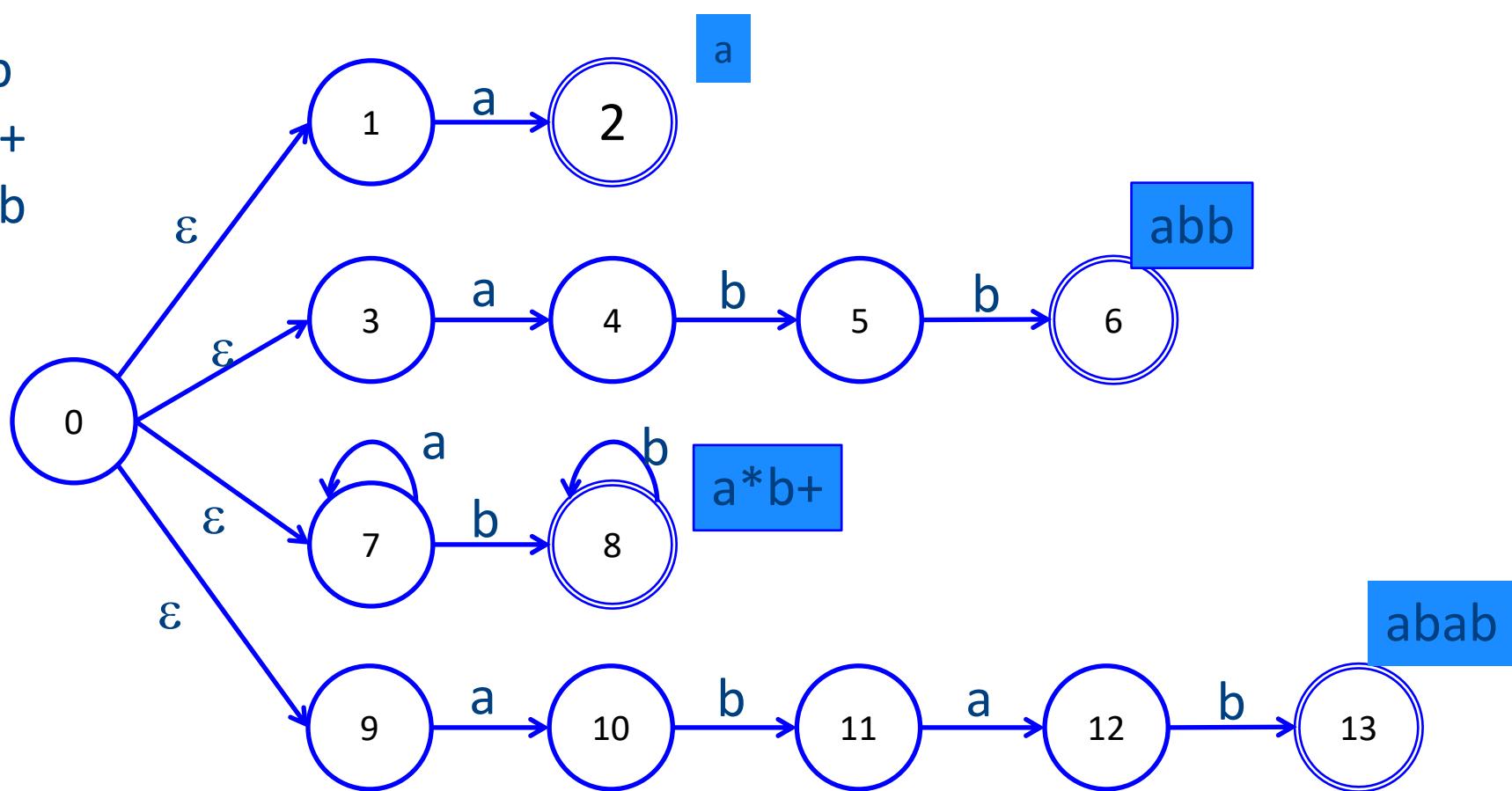
# Naïve approach

- Try each automaton separately
- Given a word  $w$ :
  - Try  $M_1(w)$
  - Try  $M_2(w)$
  - ...
  - Try  $M_n(w)$
- Requires resetting after every attempt

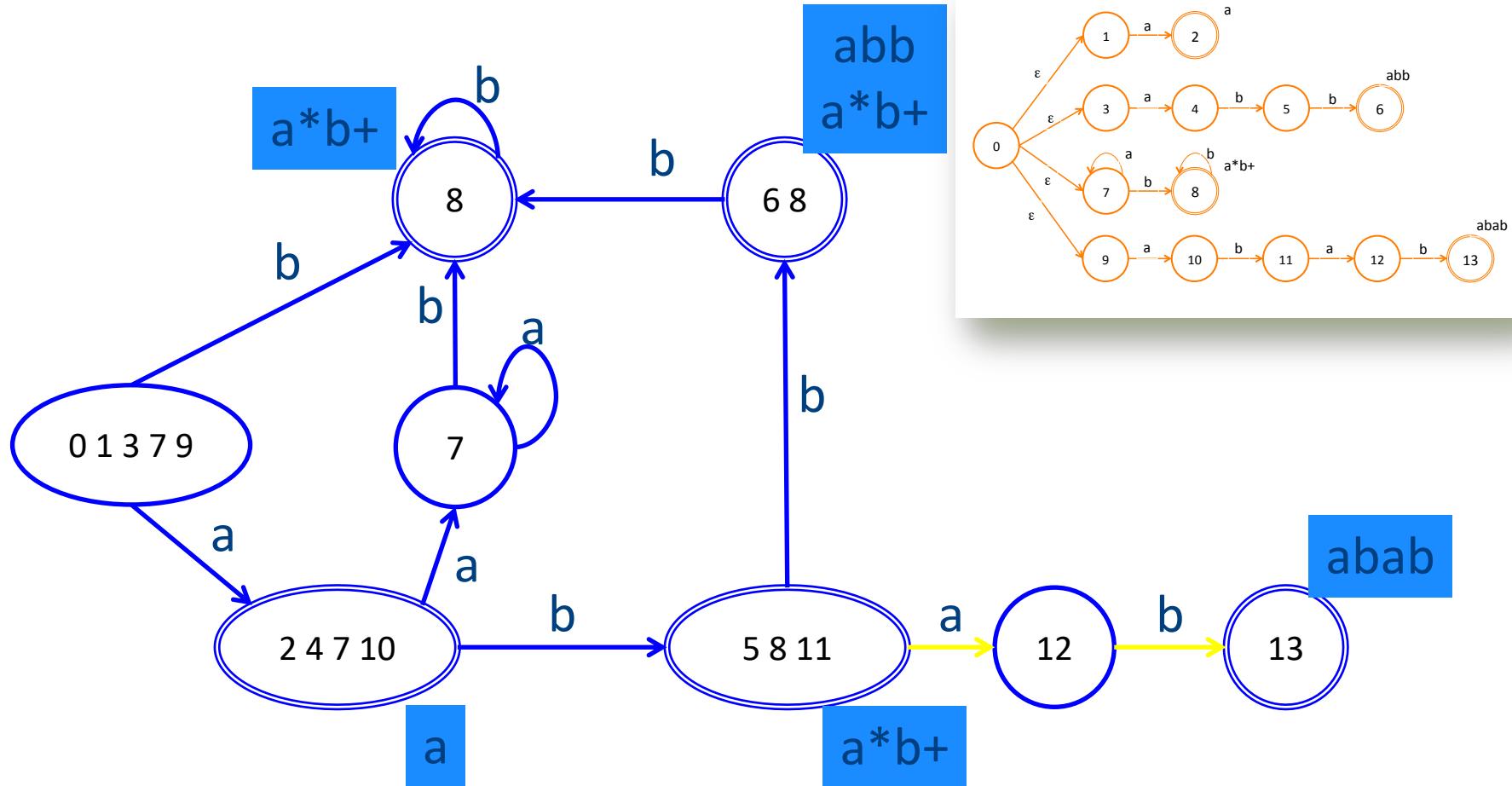
# Actually, we combine automata

combines

a  
abb  
 $a^*b^+$   
abab



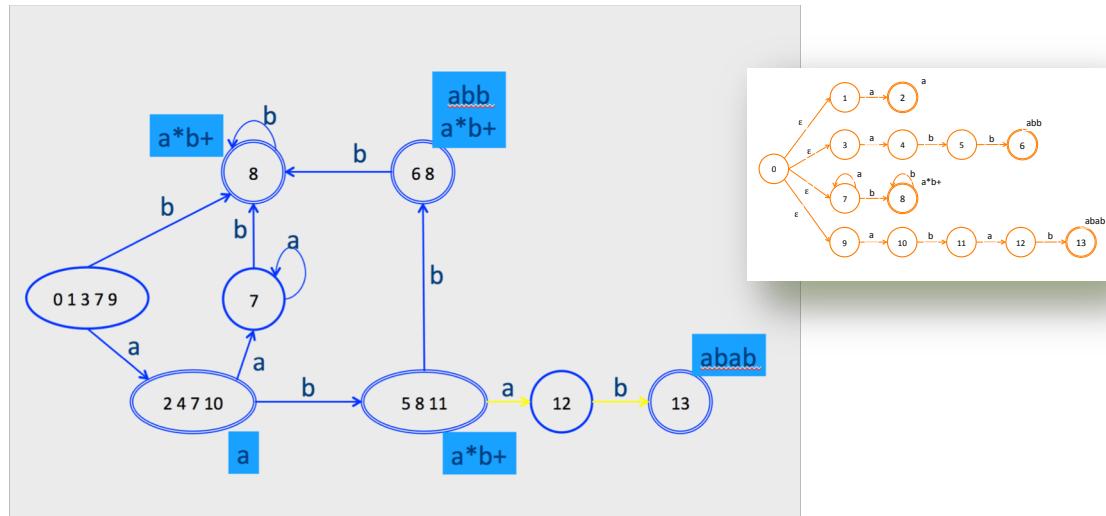
# Corresponding DFA



# Scanning with DFA

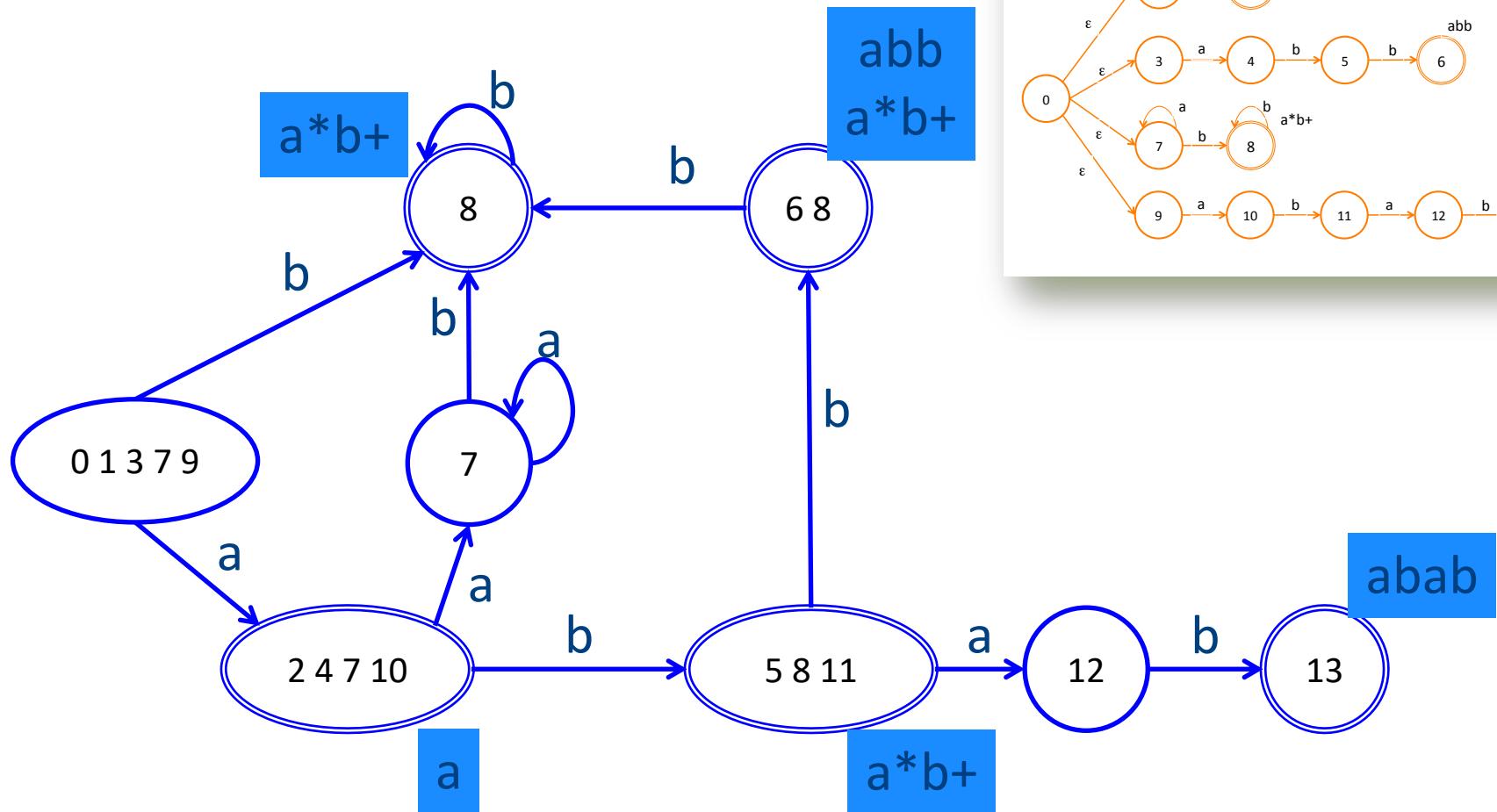
- Run until stuck
  - Remember last accepting state
- Go back to accepting state
- Return token

# Ambiguity resolution



- Longest word
- Tie-breaker based on **order of rules** when words have same length

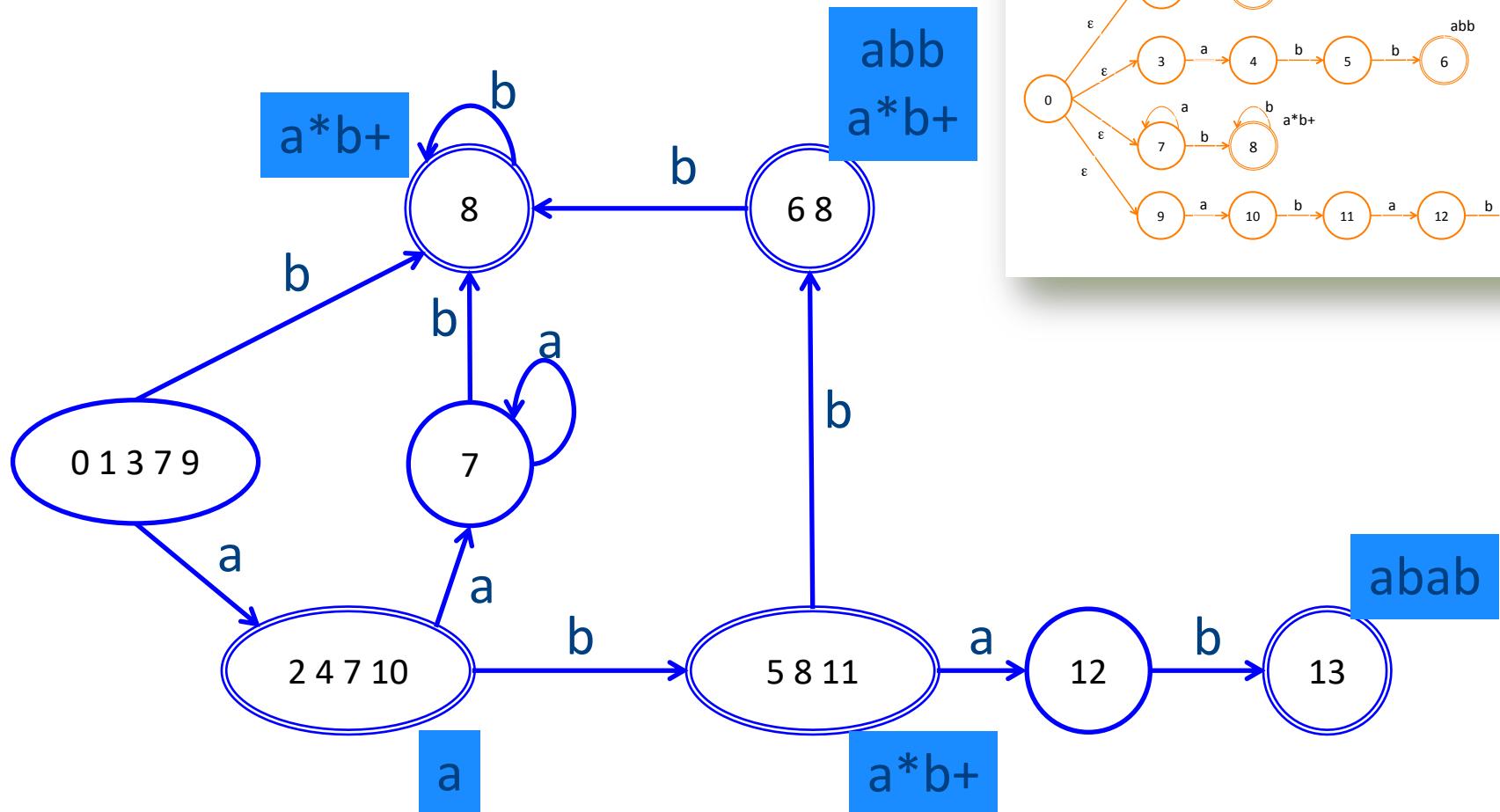
# Examples



abaa: gets stuck after aba in state 12, backs up to state (5 8 11) pattern is  $a^*b^+$ , token is ab

Tokens:  $\langle a^*b^+, ab \rangle \langle a,a \rangle \langle a,a \rangle$

# Examples



abba: stops after second b in (6 8), token is abb because it comes first in spec  
 Tokens:  $\langle abb, abb \rangle \langle a, a \rangle$

# Summary of Construction



- Describe tokens as **regular expressions**
  - Decide attributes (values) to save for each token
- Regular expressions turned into a **DFA**
  - Also, records which attributes (values) to keep
- Lexical analyzer **simulates the run of an automata** with the given transition table on any input string



# A Few Remarks

- Turning an NFA to a DFA is expensive, but
  - Exponential in the worst case
  - In practice, works fine
- The construction is done once per-language
  - At Compiler construction time
  - **Not** at compilation time

# Implementation

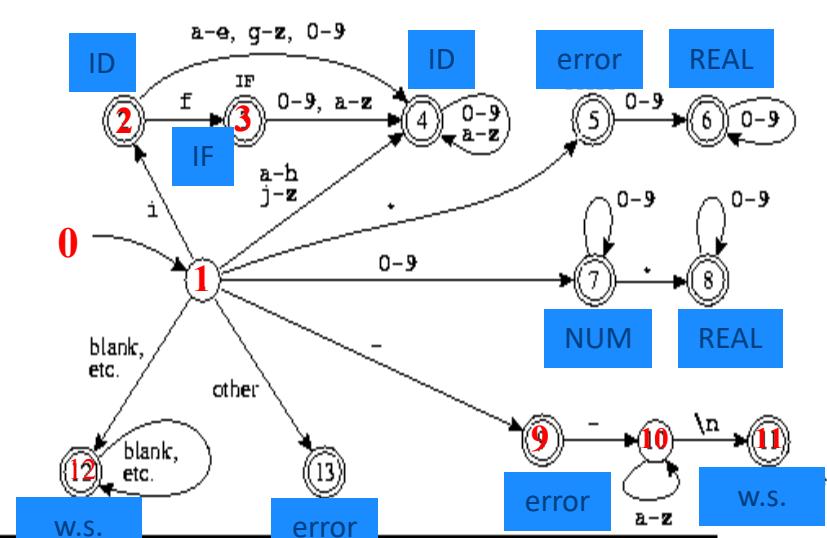


# Implementation by Example

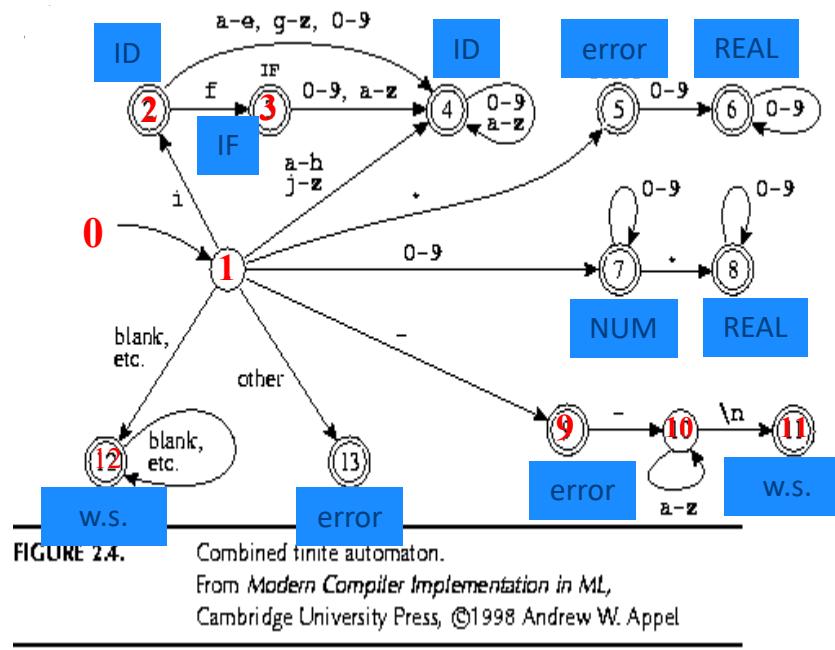
if  
 if  
 xy, i, zs98  
 [a-z][a-z0-9]\*  
 3,32, 032  
 [0-9]+  
 [0-9]”. “[0-9]+|[0-9]\*”. “[0-9]+  
 0.55, 33.1  
 (\-\\-[a-z]\*\n)|(“ “|\n|\t)  
 .  
 --comm\n  
 \n, \t, “ ”

```

{ return IF; }
{ return ID; }
{ return NUM; }
{ return REAL; }
{ ; }
{ error(); }
  
```



**FIGURE 24.** Combined finite automaton.  
 From *Modern Compiler Implementation in ML*,  
 Cambridge University Press, ©1998 Andrew W. Appel



```

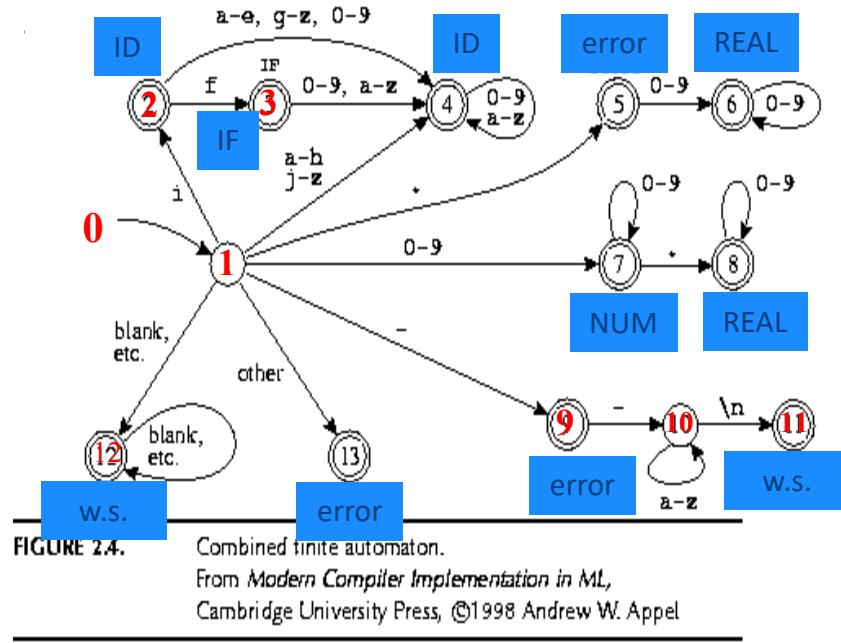
int edges[][][256] = {
    /* ... , 0, 1, 2, 3, ... , - , e, f, g, h, i, j, ... */
/* state 0 */ {0, ..., 0, 0, ..., 0, 0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, 0},
/* state 1 */ {13, ..., 7, 7, 7, 7, ..., 9, 4, 4, 4, 4, 2, 4, ..., 13, 13},
/* state 2 */ {0, ..., 4, 4, 4, 4, ..., 0, 4, 3, 4, 4, 4, 4, ..., 0, 0},
/* state 3 */ {0, ..., 4, 4, 4, 4, ..., 0, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0},
/* state 4 */ {0, ..., 4, 4, 4, 4, ..., 0, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0},
/* state 5 */ {0, ..., 6, 6, 6, 6, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* state 6 */ {0, ..., 6, 6, 6, 6, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* state 7 */ ...
/* state ... */
/* state 13 */ {0, ..., 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

```

# Pseudo Code for Scanner

```
char* input = ... ;  
  
Token nextToken() {  
    lastFinal = 0;  
    currentState = 1 ;  
    inputPositionAtLastFinal = input;  
    currentPosition = input;  
    while (not(isDead(currentState))) {  
        nextState = edges[currentState][*currentPosition];  
        if (isFinal(nextState)) {  
            lastFinal = nextState ;  
            inputPositionAtLastFinal = currentPosition;  
        }  
        currentState = nextState;  
        advance currentPosition;  
    }  
    input = inputPositionAtLastFinal + 1;  
    return action[lastFinal];  
}
```

# Example



Input: "if --not-a-com"



2 blanks

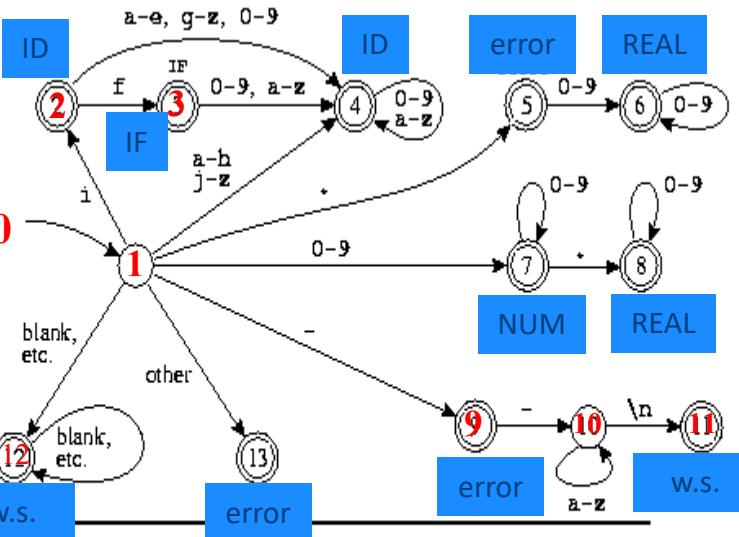


FIGURE 2.4.

Combined finite automaton.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel

return IF

final state	input
0	if --not-a-com
2	if --not-a-com
3	if --not-a-com
3	if --not-a-com

+

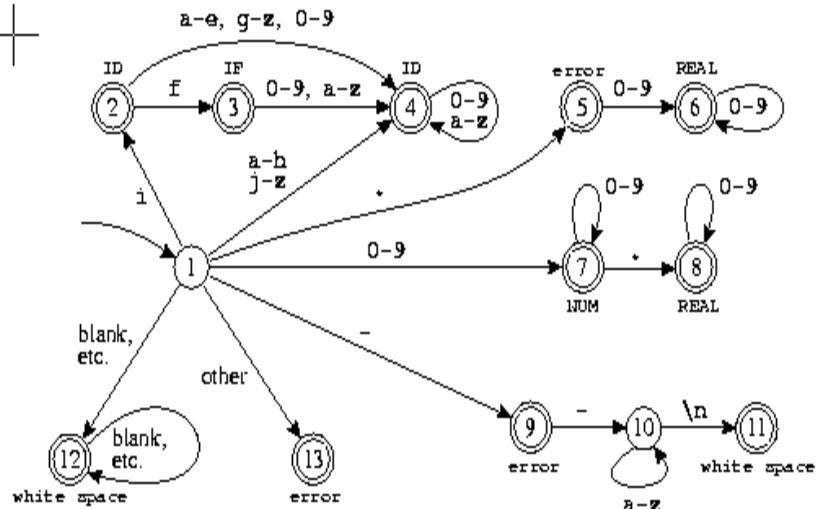


FIGURE 2.4.

Combined finite automaton.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel

found whitespace

final state	input
0	--not-a-com
12	--not-a-com
12	--not-a-com
12	--not-a-com

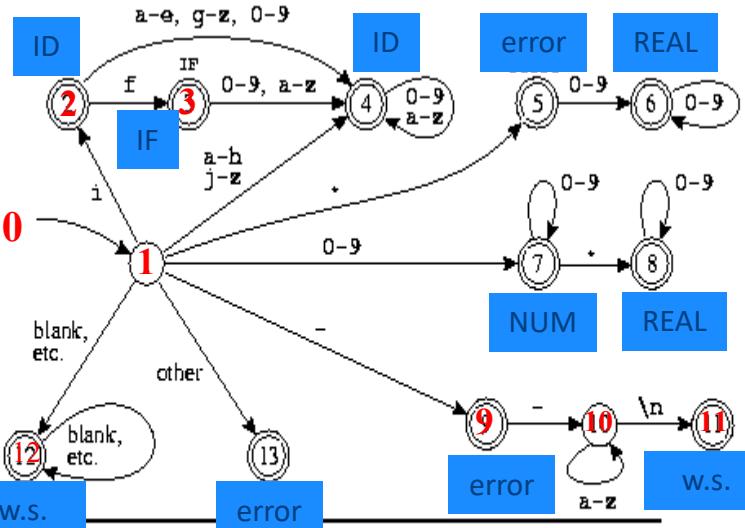


FIGURE 2.4.

Combined finite automaton.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	--not-a-com
9	9	--not-a-com
9	10	--not-a-com
10	10	--not-a-com
10	10	--not-a-com
10	0	--not-a-com

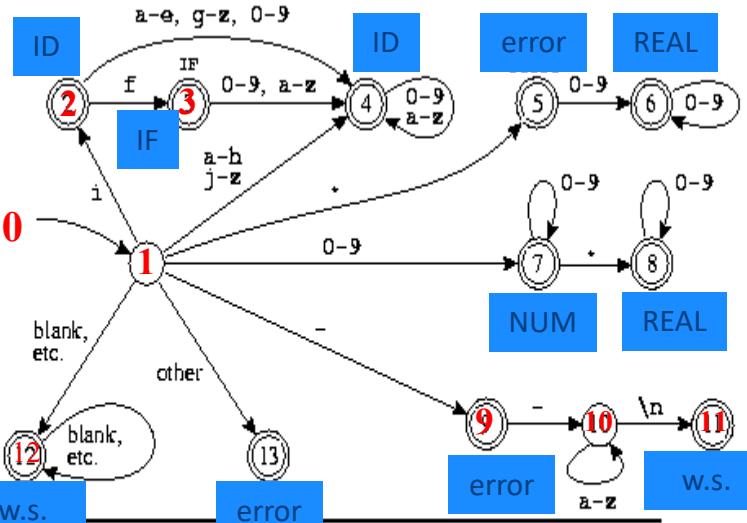


FIGURE 2.4.

Combined finite automaton.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	-not-a-com
9	9	-not-a-com
9	0	-not-a-com
9	0	-not-a-com
9	0	-not-a-com

# Concluding remarks

- Efficient scanner
- Minimization
- Error handling
- Automatic creation of lexical analyzers

# Efficient Scanners

- Efficient state representation
- Input buffering
- Using switch and gotos instead of tables

# Minimization

- Create a non-deterministic automaton (NDFA) from every regular expression
- Merge all the automata using epsilon moves (like the  $\mid$  construction)
- Construct a deterministic finite automaton (DFA)
  - State priority
- Minimize the automaton
  - separate accepting states by token kinds

# Example

```
if { return IF; }
[a-z][a-zA-Z]* { return ID; }
[0-9]+ { return NUM; }
```

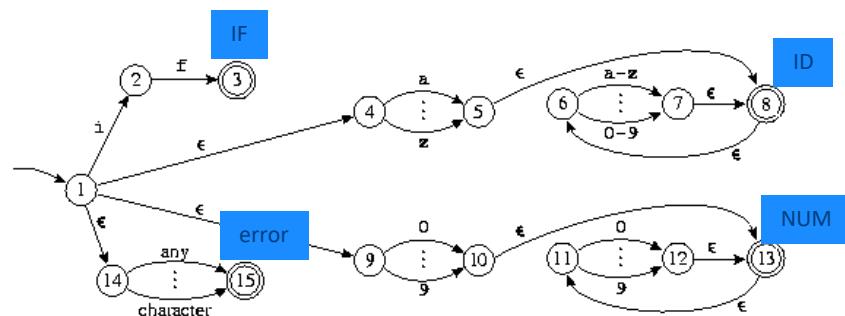


FIGURE 2.7.

Four regular expressions translated to an NFA.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel

# Example

```
if
[a-z][a-zA-Z]*
[0-9]+
```

```
{ return IF; }
{ return ID; }
{ return NUM; }
```

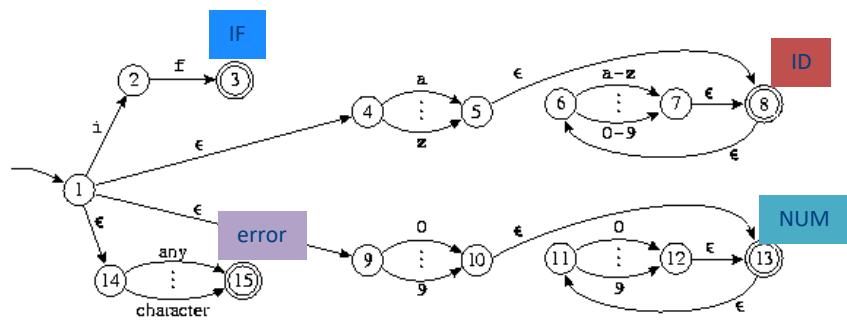
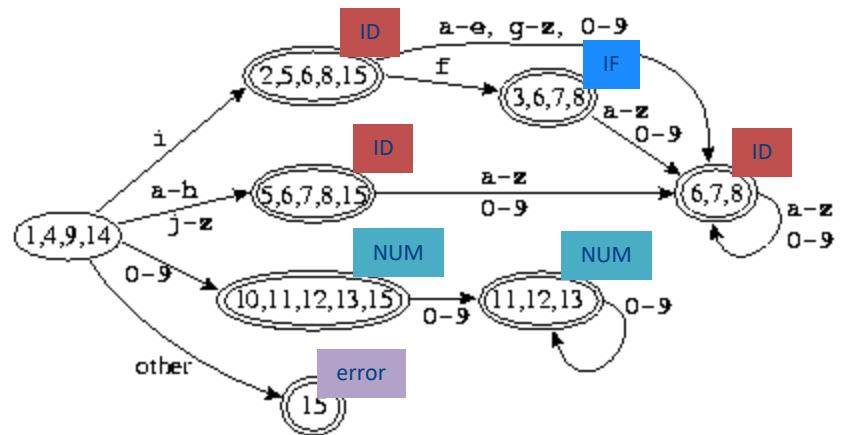
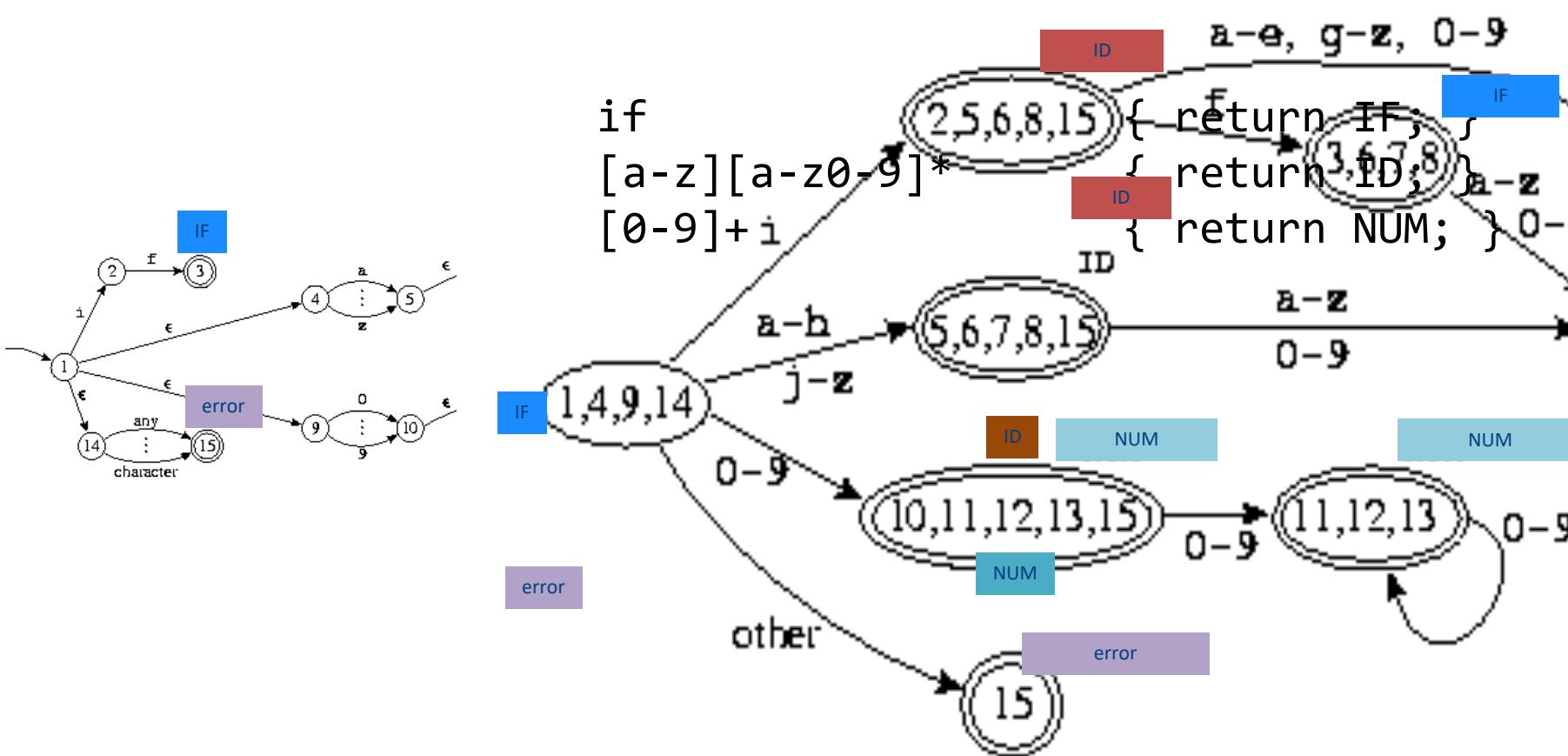


FIGURE 2.7.

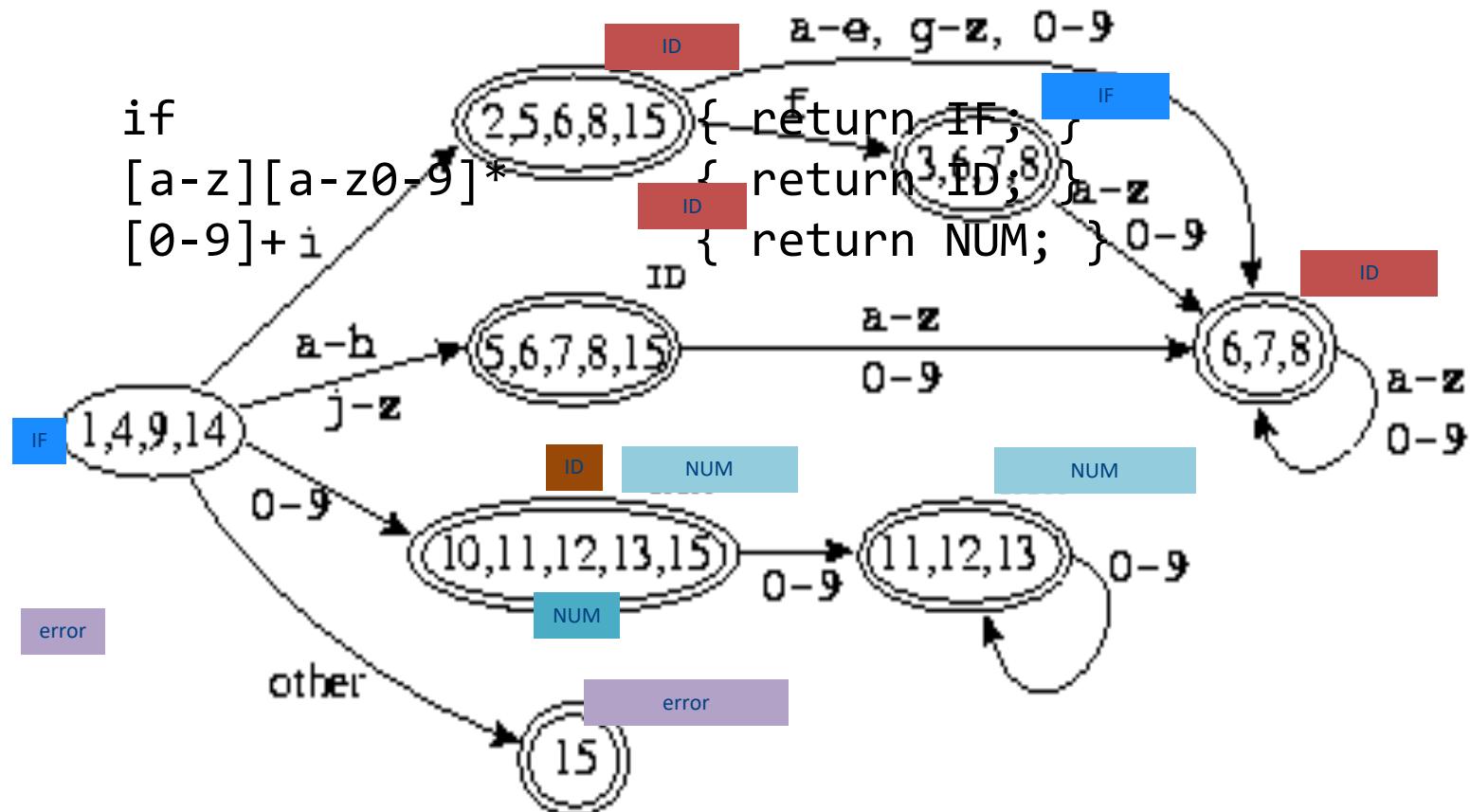
Four regular expressions translated to an NFA.  
From *Modern Compiler Implementation in ML*,  
Cambridge University Press, ©1998 Andrew W. Appel



# Example



# Example

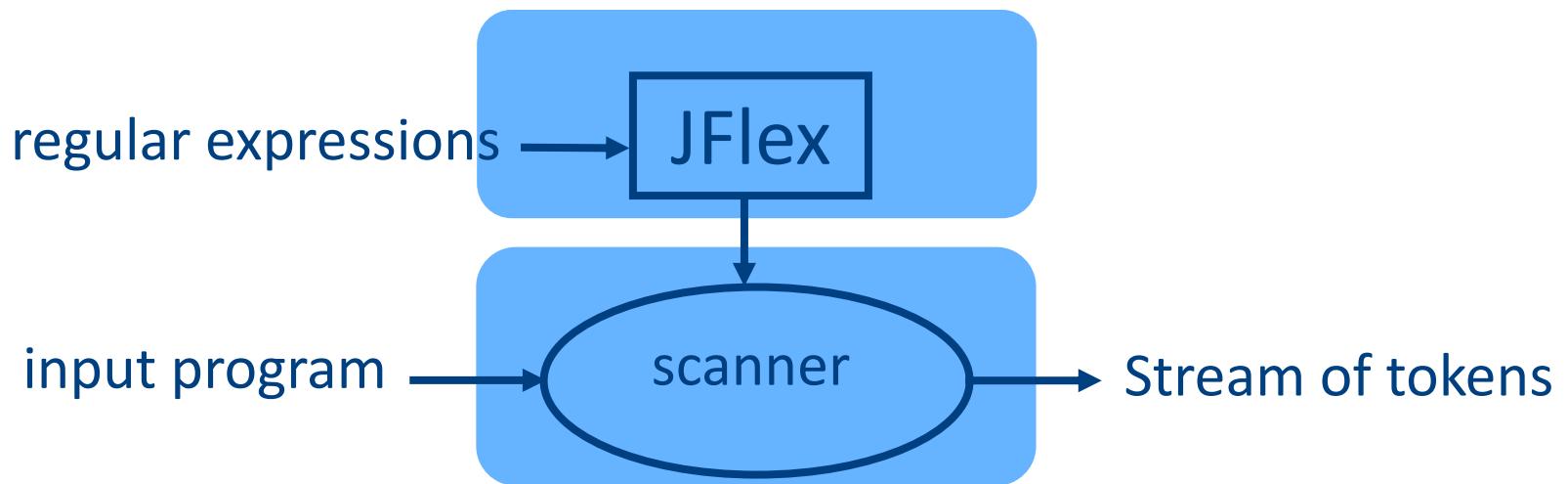


# Error Handling

- Many errors cannot be identified at this stage
- Example: “fi (a==f(x))”. Should “fi” be “if”? Or is it a routine name?
  - We will discover this later in the analysis
  - At this point, we just create an identifier token
- Sometimes the lexeme does not match any pattern
  - Easiest: eliminate letters until the beginning of a legitimate lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.
- Goal: allow the compilation to continue
- Problem: errors that spread all over

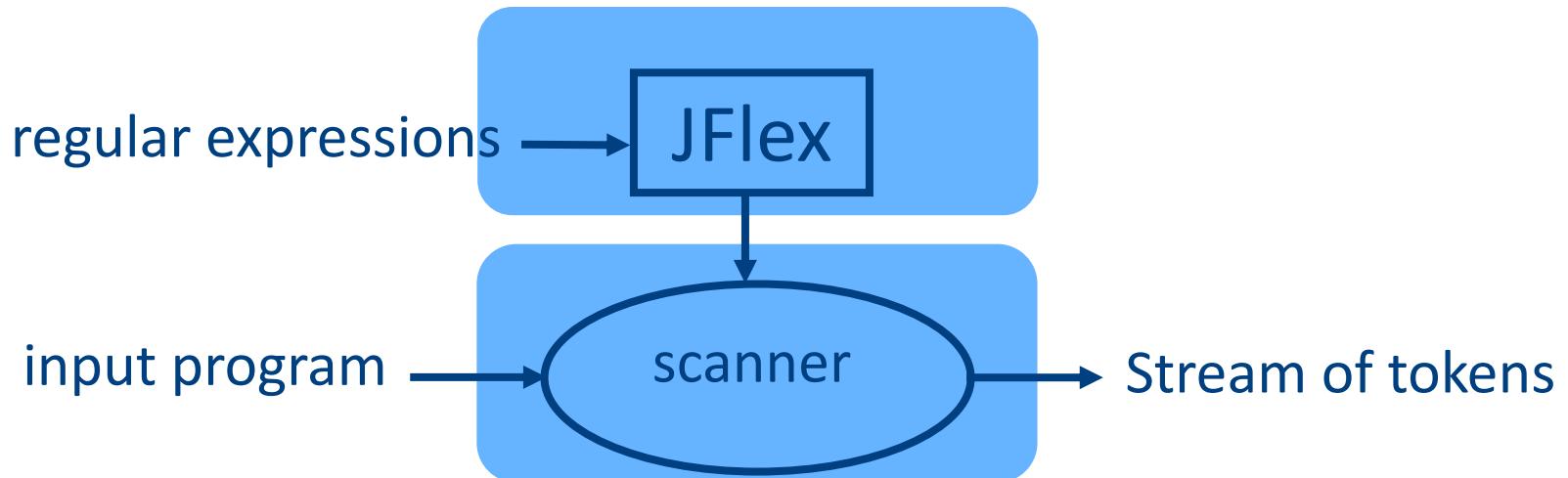
# Automatically generated scanners

- Use of Program-Generating Tools
  - Specification → Part of compiler
  - Compiler-Compiler



# Use of Program-Generating Tools

- Input: regular expressions and actions
  - Action = Java code
- Output: a scanner program that
  - Produces a stream of tokens
  - Invoke actions when pattern is matched



# Line Counting Example

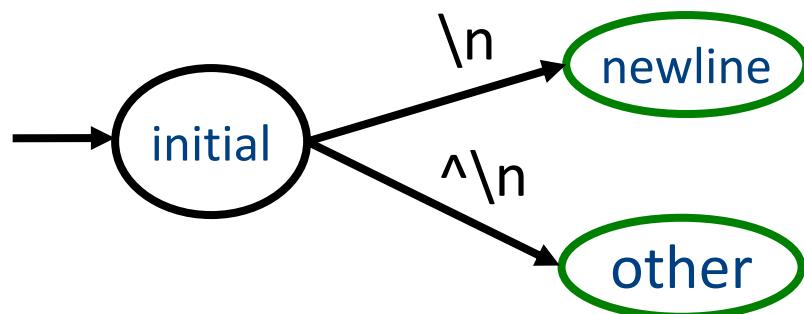
- Create a program that counts the number of lines in a given input text file

# Creating a Scanner using Flex

```
int num_lines = 0;  
%%  
\n      ++num_lines;  
.      ;  
%%  
main() {  
    yylex();  
    printf( "# of lines = %d\n", num_lines);  
}
```

# Creating a Scanner using Flex

```
int num_lines = 0;  
%%  
\n      ++num_lines;  
.      ;  
%%  
main() {  
    yylex();  
    printf( "# of lines = %d\n", num_lines);  
}
```



# JFlex Spec File

User code: Copied directly to Java file

%%

JFlex directives: macros, state names

%%

Lexical analysis rules:

- Optional state, regular expression, action
- How to break input to tokens
- Action when token matched

Possible source of  
javac errors down  
the road

DIGIT= [0-9]  
LETTER= [a-zA-Z]  
YYINITIAL

{LETTER}  
({LETTER}|{DIGIT})\*

# Creating a Scanner using JFlex

```
import java_cup.runtime.*;
%%
%cup
%{
    private int lineCounter = 0;
%}

%eofval{
    System.out.println("line number=" + lineCounter);
    return new Symbol(sym.EOF);
%eofval}

NEWLINE=\n
%%
{NEWLINE}        { lineCounter++; }
[^{NEWLINE}]     { }
```

# Catching errors

- What if input doesn't match any token definition?
- Trick: Add a “catch-all” rule that matches any character and reports an error
  - Add after all other rules

# A JFlex specification of C Scanner

```
import java_cup.runtime.*;
%%
%cup
%{
    private int lineCounter = 0;
}
Letter= [a-zA-Z_]
Digit= [0-9]
%%
"\t"    { }
"\n"    { lineCounter++; }
";"    { return new Symbol(sym.SemiColumn); }
"++"   { return new Symbol(sym.PlusPlus); }
"+="   { return new Symbol(sym.PlusEq); }
"+"    { return new Symbol(sym.Plus); }
"while" { return new Symbol(sym.While); }
{Letter}({Letter}|{Digit})*
        { return new Symbol(sym.Id, yytext()); }
"<="   { return new Symbol(sym.LessOrEqual); }
"<"    { return new Symbol(sym.LessThan); }
```

# Missing

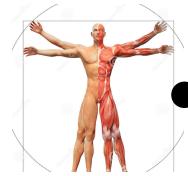
- Creating a lexical analysis by hand
- Table compression
- Symbol Tables
- Nested Comments
- Handling Macros

# Lexical Analysis: What

- Input: program text (file)
- Output: sequence of tokens

# Lexical Analysis: How

- Define tokens using regular expressions
- Construct a nondeterministic finite-state automaton (NFA) from regular expression
- Determinize the NFA into a deterministic finite-state automaton (DFA)
- DFA can be directly used to identify tokens



# Lexical Analysis: Why

- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- [Produce symbol table]

# The Real Anatomy of a Compiler

