CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 8A Lexical Analysis – Tokenization

LECTURE 10: TOKENIZATION

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- File Read/Write Access
- Understand what is Syntax Diagram
- Write Regular Expression
- Tokenization Using Regular Expression

Overview

SECTION 1

Overview of Tokenization

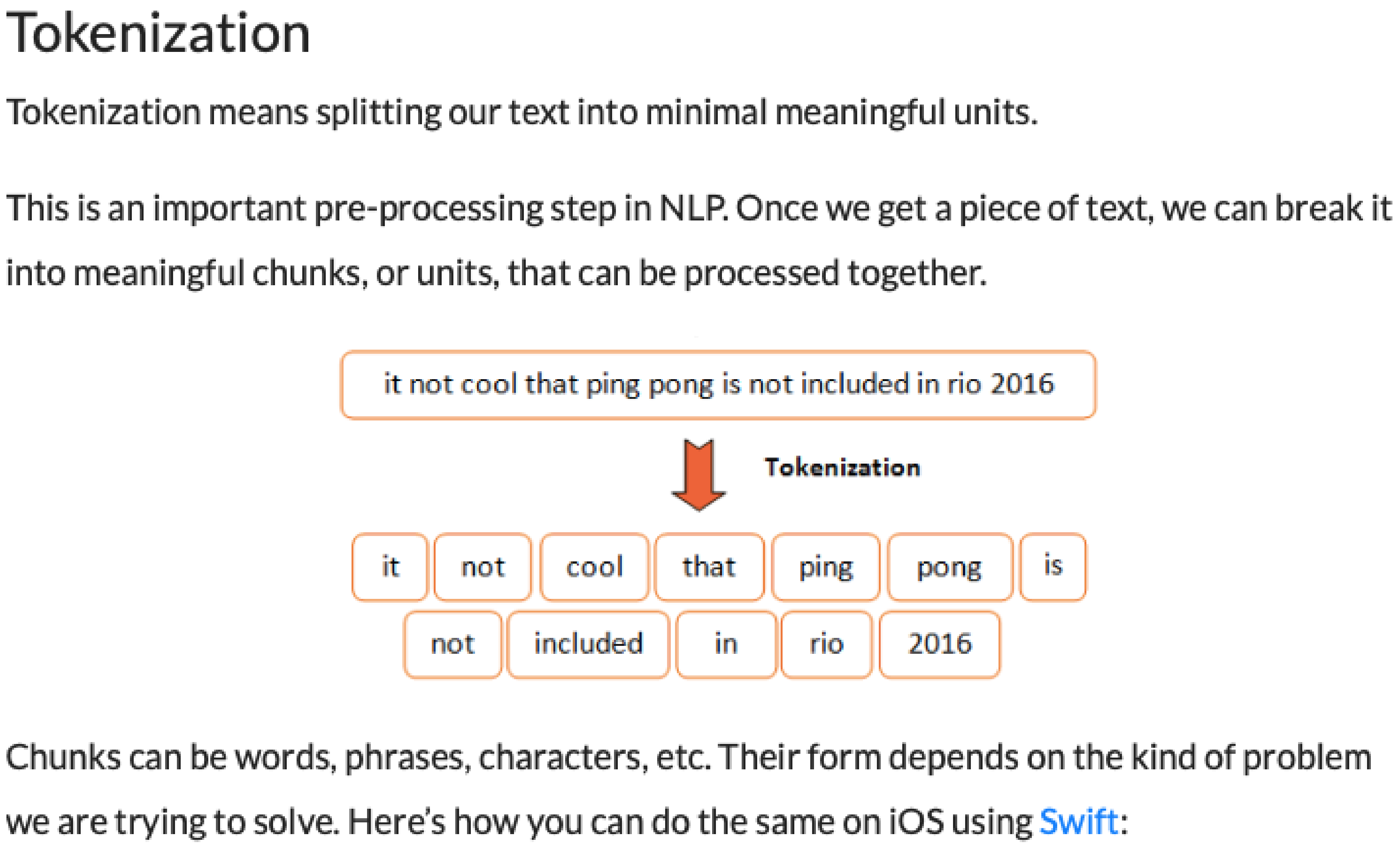
- Tokenization is a common task a data scientist comes across when working with text data. It consists of splitting an entire text into small units, also known as tokens.
- Most Natural Language Processing (NLP) projects have tokenization as the first step because it’s the foundation for developing good models and helps better understand the text we have.

Text Processing – Lexical Analysis

- Standard Steps:
  - Recognize language class [English] (very easy)
  - Recognize document structure
    - titles, sections, paragraphs, etc.
  - Break into tokens – type of markup
    - Tokens are delimited text
      - Hello, how are you.
      - _hello_,_how_are_you_._
    - usually space and punctuation delineated
    - special issues with Asian languages
  - Lemmatization, stemming/morphological analysis
  - What is left are terms
  - Store in inverted index

Text Processing – Lexical Analysis

- Lexical analysis is the process of converting a sequence of characters into a sequence of tokens.
  - A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**.



Tokenization

Tokenization means splitting our text into minimal meaningful units.

This is an important pre-processing step in NLP. Once we get a piece of text, we can break it into meaningful chunks, or units, that can be processed together.

it not cool that ping pong is not included in rio 2016

Tokenization

it | not | cool | that | ping | pong | is

not | included | in | rio | 2016

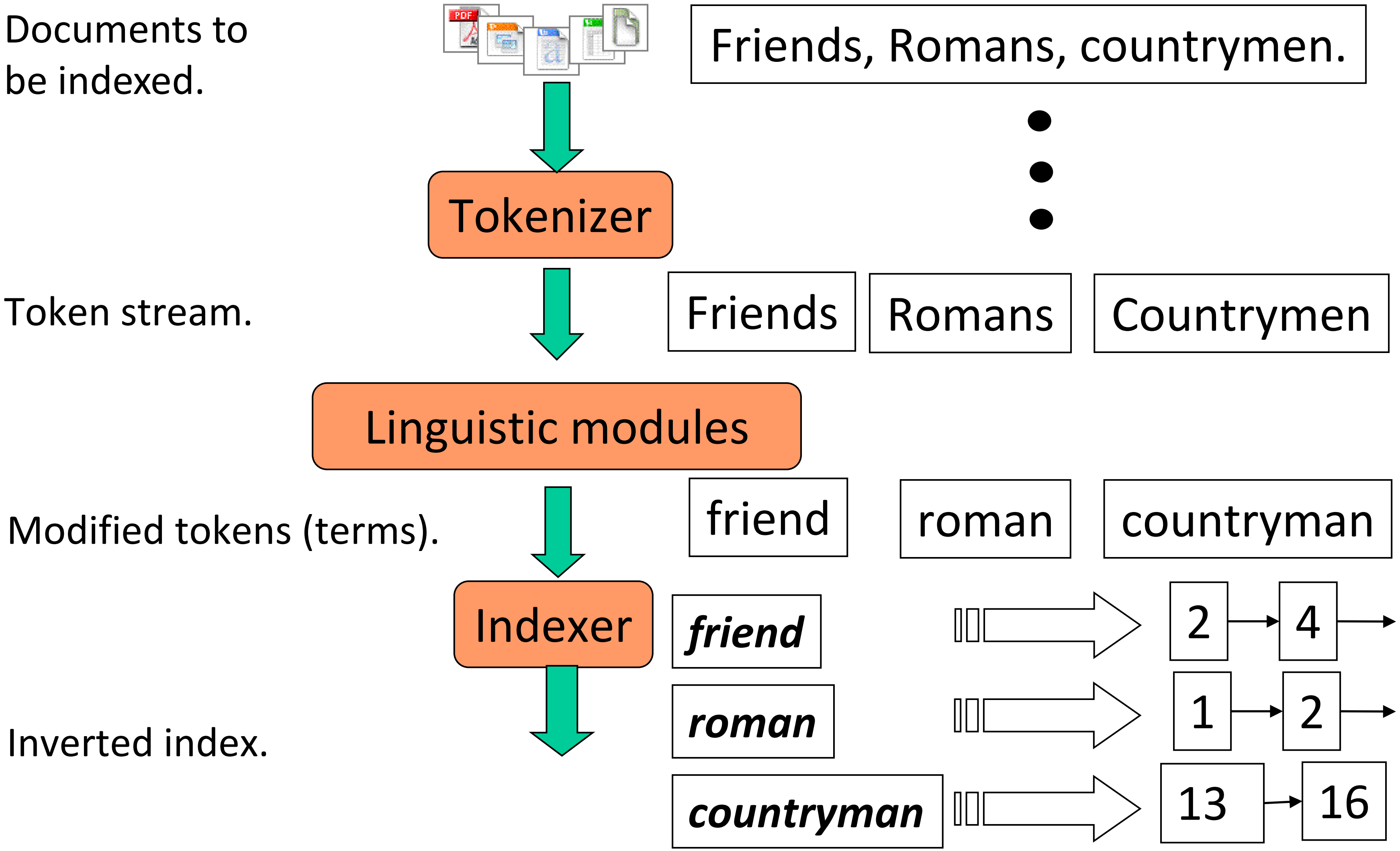
Chunks can be words, phrases, characters, etc. Their form depends on the kind of problem we are trying to solve. Here's how you can do the same on iOS using Swift:

Sometimes called ”parsers”

What tokenization did you use?

- For real problems always ask this!
- A fundamental question for all text processing
  - Natural language processing
  - Text mining
  - Machine learning and AI
  - Information retrieval and search

Basic indexing pipeline



Documents to be indexed.

Friends, Romans, countrymen.

Tokenizer

Token stream.

Friends

Romans

Countrymen

Linguistic modules

Modified tokens (terms).

friend

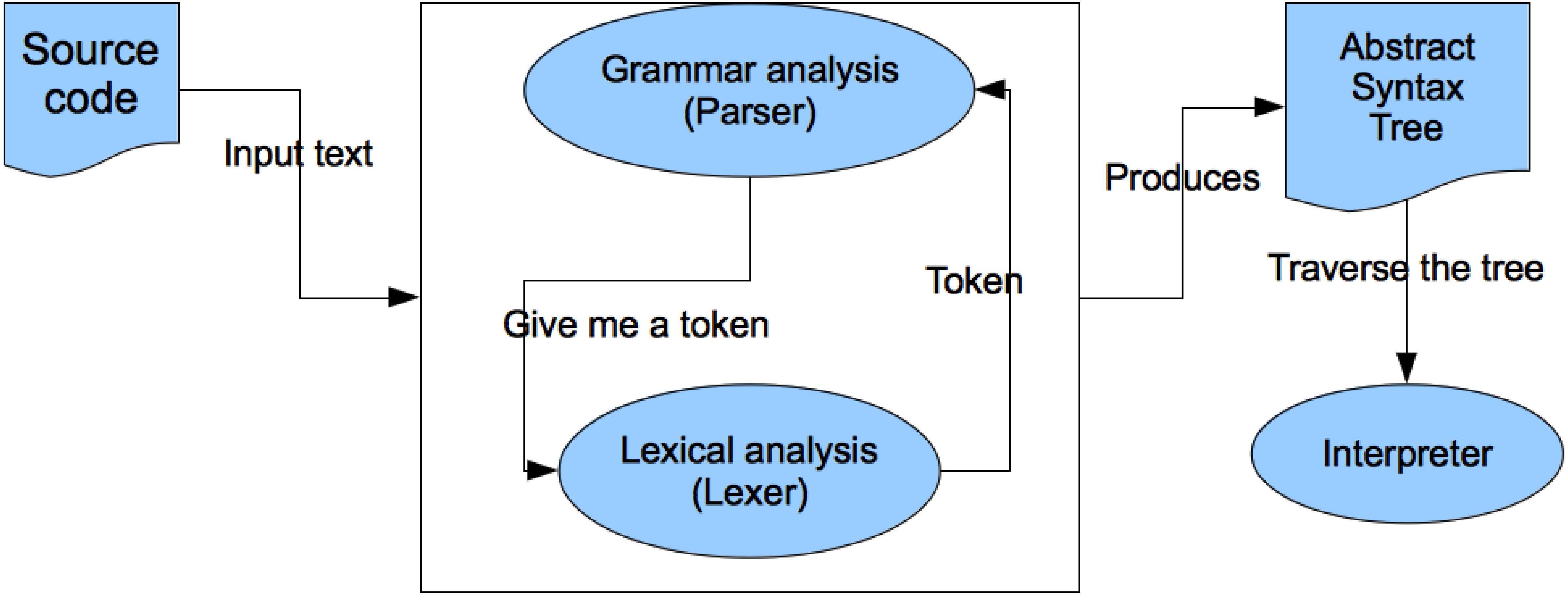
roman

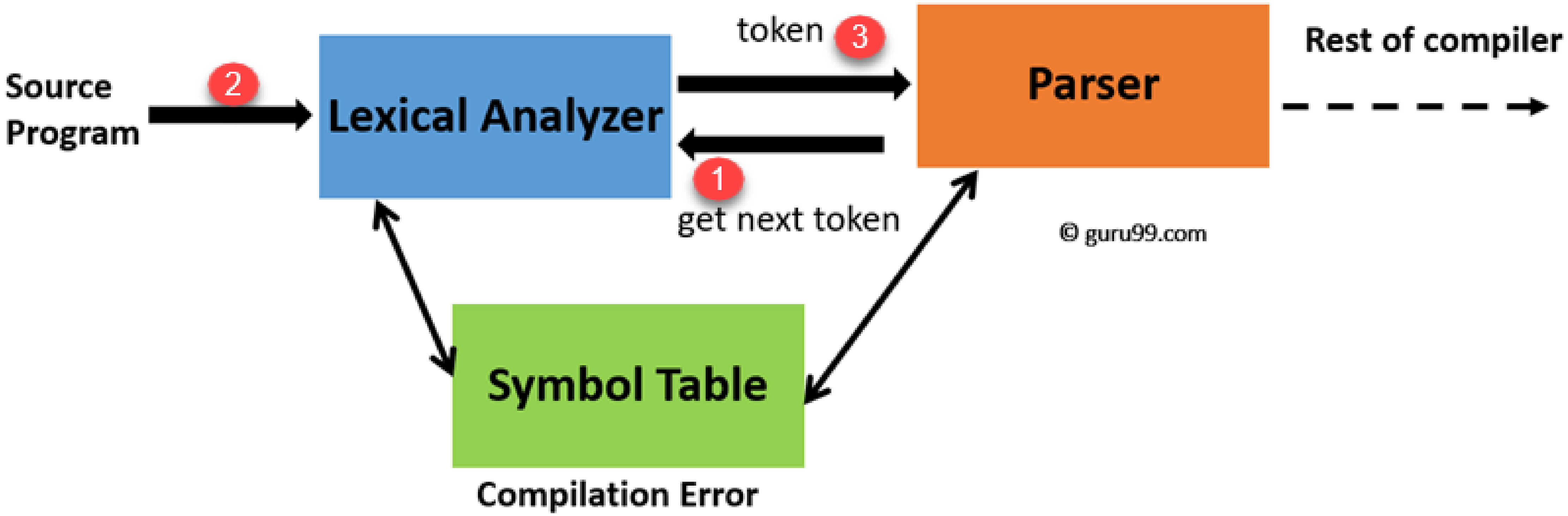
countryman

Indexer

Inverted index.

friend	2	4
roman	1	2
countryman	13	16


Source code
Input text
Grammar analysis (Parser)
Give me a token
Lexical analysis (Lexer)
Token
Produces
Abstract Syntax Tree
Traverse the tree
Interpreter



Source Program

2

Lexical Analyzer

token

3

Parser

Rest of compiler

1

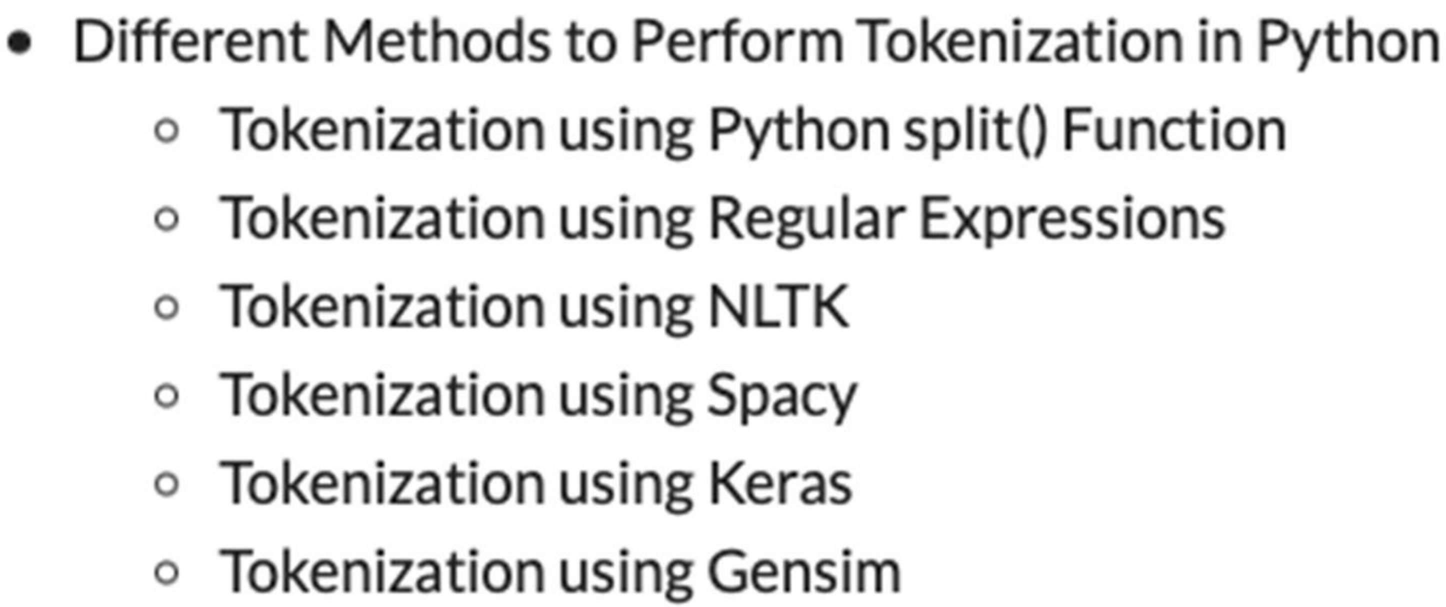
get next token

© guru99.com

Symbol Table

Compilation Error

Lots of Tokenizers out There



- Different Methods to Perform Tokenization in Python
  - Tokenization using Python split() Function
  - Tokenization using Regular Expressions
  - Tokenization using NLTK
  - Tokenization using Spacy
  - Tokenization using Keras
  - Tokenization using Gensim

Tokenization example

- **Input:** “Friends, Romans and Countrymen”
- **Output:** Tokens
  - friends
  - romans
  - countrymen
- Each such token is now a candidate for an index entry, after further processing
  - Described below
- But what are valid tokens to emit?

Tokenization

- Issues in tokenization:
  - ***Finland's capital*** →
  - ***Finland? Finlands? Finland's***?
  - ***Hewlett-Packard*** →
    - ***Hewlett*** and ***Packard*** as two tokens?
    - ***State-of-the-art***: break up hyphenated sequence.
    - co-education ?
    - the hold-him-back-and-drag-him-away-maneuver ?
  - ***San Francisco***: one token or two? How do you decide it is one token?

Tokenization By Split()

SECTION 2

Batch Mode Tokenization

- Reading the whole file in a batch operation.
- **Advantage:** Easy coding
- **Disadvantage:** When file is big, there may be memory limit issue. There might also be problem with possessive term, plural nouns, and other grammatical issues.

Read in the Whole File

batch.py

```
f = open("text.txt", "r")
fstr = f.read().strip()
tokens = fstr.split()
print(tokens)
f.close()
```

```
['Here’s', 'to', 'the', 'crazy', 'ones,', 'the', 'misfits,',
'the', 'rebels,', 'the', 'troublemakers,', 'the', 'round',
'pegs', 'in', 'the', 'square', 'holes.', 'The', 'ones', 'who',
'see', 'things', 'differently', '—', 'they’re', 'not', 'fond',
'of', 'rules.', 'You', 'can', 'quote', 'them,', 'disagree',
'with', 'them,', 'glorify', 'or', 'vilify', 'them,', 'but',
'the', 'only', 'thing', 'you', 'can’t', 'do', 'is', 'ignore',
'them', 'because', 'they', 'change', 'things.', 'They', 'push',
'the', 'human', 'race', 'forward,', 'and', 'while', 'some',
'may', 'see', 'them', 'as', 'the', 'crazy', 'ones,', 'we', 'see',
'genius,', 'because', 'the', 'ones', 'who', 'are', 'crazy',
'enough', 'to', 'think', 'that', 'they', 'can', 'change', 'the',
'world,', 'are', 'the', 'ones', 'who', 'do.']
```

Line by Line Tokenization

- Reading the whole file in a line-by-line format.
- **Advantage:** Smaller buffer needed. More efficient. No memory limit issue.
- **Disadvantage:** There might also be problem with possessive term, plural nouns, and other grammatical issues.

Read In Line by Line

line.py

```
f = open("text.txt", "r")
line = f.readline()
tokens=[]
while line:
    line = line.strip()
    tokens += line.split()
    line = f.readline()
print(tokens)
f.close()
```

```
['Here’s', 'to', 'the', 'crazy', 'ones,', 'the', 'misfits,',
'the', 'rebels,', 'the', 'troublemakers,', 'the', 'round',
'pegs', 'in', 'the', 'square', 'holes.', 'The', 'ones', 'who',
'see', 'things', 'differently', '—', 'they’re', 'not', 'fond',
'of', 'rules.', 'You', 'can', 'quote', 'them,', 'disagree',
'with', 'them,', 'glorify', 'or', 'vilify', 'them,', 'but',
'the', 'only', 'thing', 'you', 'can’t', 'do', 'is', 'ignore',
'them', 'because', 'they', 'change', 'things.', 'They', 'push',
'the', 'human', 'race', 'forward,', 'and', 'while', 'some',
'may', 'see', 'them', 'as', 'the', 'crazy', 'ones,', 'we', 'see',
'genius,', 'because', 'the', 'ones', 'who', 'are', 'crazy',
'enough', 'to', 'think', 'that', 'they', 'can', 'change', 'the',
'world,', 'are', 'the', 'ones', 'who', 'do.']
```

REPL Mode

- **Prompt Toolkit:** https://python-prompt-toolkit.readthedocs.io/en/latest/

Some features:

- Syntax highlighting of the input while typing. (For instance, with a Pygments lexer.)
- Multi-line input editing.
- Advanced code completion.
- Selecting text for copy/paste. (Both Emacs and Vi style.)
- Mouse support for cursor positioning and scrolling.
- Auto suggestions. (Like fish shell.)
- No global state.

Lexical Analysis

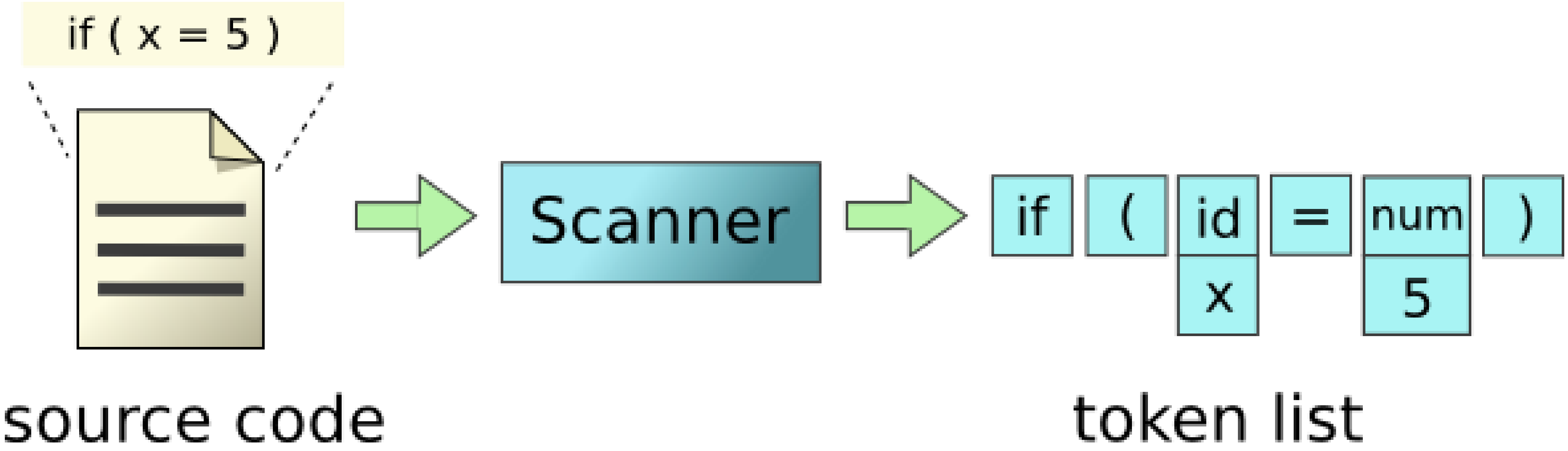
SECTION 3

Overview

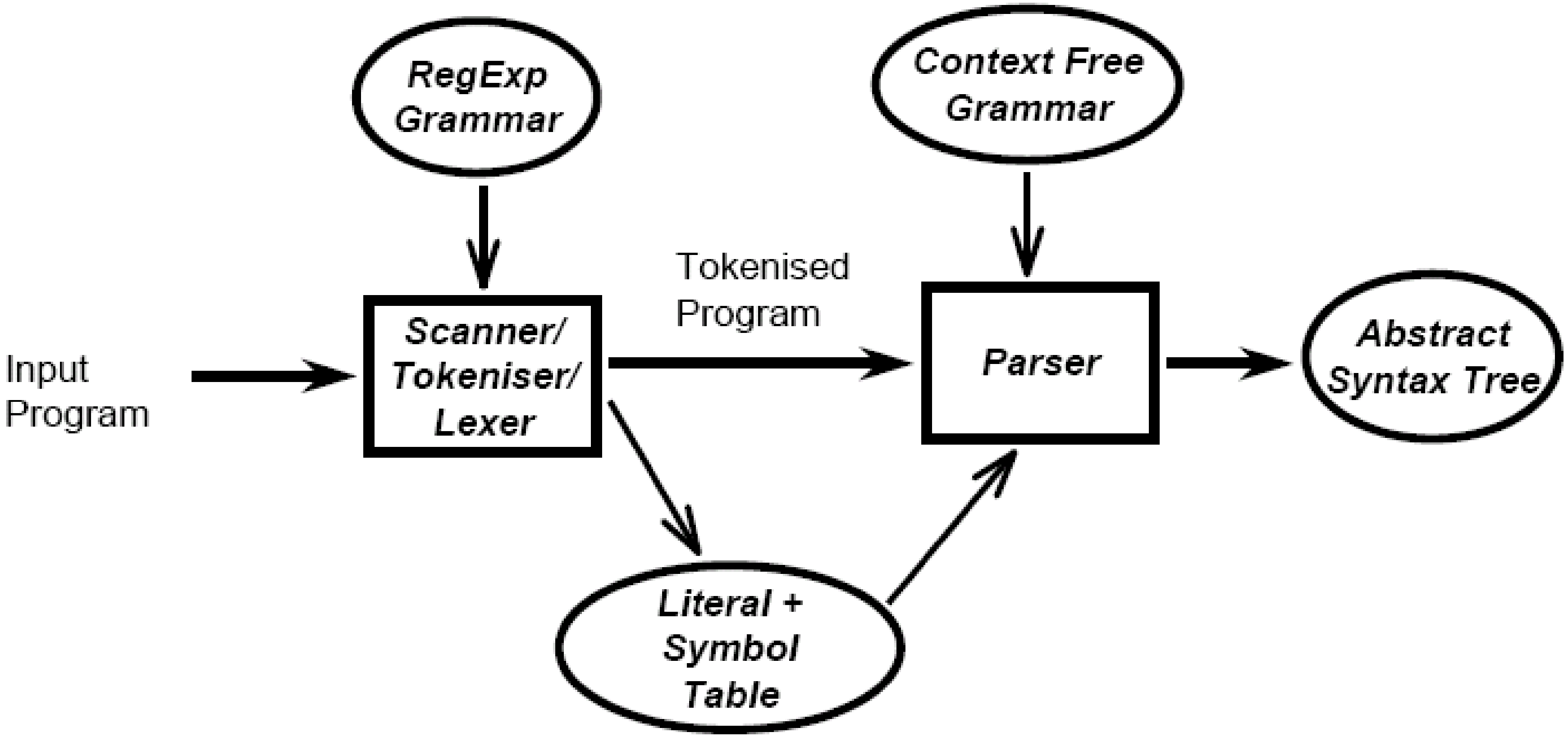
- Scanner (Tokenizer)
- From Regular Expression to Non-Deterministic Finite Automata
- From Non-Deterministic Finite Automata
- DFA Minimization
- Scanner Styles

Lexical Analysis

From Source File to Token List



Syntax Analysis



Calculator Language

Exemplary Language

C-Style Comment Line:

$$\begin{aligned} \textit{comment} \longrightarrow\ & \texttt{/*}\ (\ \textit{non-}\texttt{*} \mid \texttt{*}\ \textit{non-}\texttt{/}\ )^{*}\ \texttt{*}^{*}\texttt{/} \\ & \mid \texttt{//}\ (\ \textit{non-newline}\ )^{*}\ \textit{newline} \end{aligned}$$

$$\textit{assign} \longrightarrow \texttt{:=}$$

$$\textit{plus} \longrightarrow \texttt{+}$$

$$\textit{minus} \longrightarrow \texttt{-}$$

$$\textit{times} \longrightarrow \texttt{*}$$

$$\textit{div} \longrightarrow \texttt{/}$$

$$\textit{lparen} \longrightarrow \texttt{(}$$

$$\textit{rparen} \longrightarrow \texttt{)}$$

$$\begin{aligned} \textit{id} \longrightarrow\ & \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^{*} \\ & \text{except for } \texttt{read} \text{ and } \texttt{write} \end{aligned}$$

$$\textit{number} \longrightarrow \textit{digit}\ \textit{digit}^{*} \mid \textit{digit}^{*}\ (\ \texttt{.}\ \textit{digit} \mid \textit{digit}\ \texttt{.}\ )\ \textit{digit}^{*}$$

Rules with Terminal

Recall scanner is responsible for

- tokenizing source
- removing comments
- (often) dealing with pragmas (i.e., significant comments)
- saving text of identifiers, numbers, strings
- saving source locations (file, line, column) for error messages

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens
  `{ ( ) [ ] < > , ; = + - etc }`
  we announce that **token**
- If it is a **`.`**, we look at the next character
  - If that is a dot, we announce **`.`**
  - Otherwise, we announce **`.`** and reuse the look-ahead
  - (field selector a.b/range delimiter ..)

Scanning

- If it is a **<**, we look at the next character
  - if that is a **=** we announce **<=**
  - otherwise, we announce **<** and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserve word

If it is a digit, we keep reading until we find a non-digit

- if that is not a `.` we announce an **`integer`**
- otherwise, we keep looking for a real number
- if the character after the `.` is not a digit we announce an integer and reuse the `.` and the look-ahead

This is a deterministic finite automaton (DFA)

- Lex, scangen, etc. build these things automatically from a set of regular expressions
- Specifically, they construct a machine that accepts the language

```
identifier | int const
| real const | comment | symbol | ...
```

Scanning

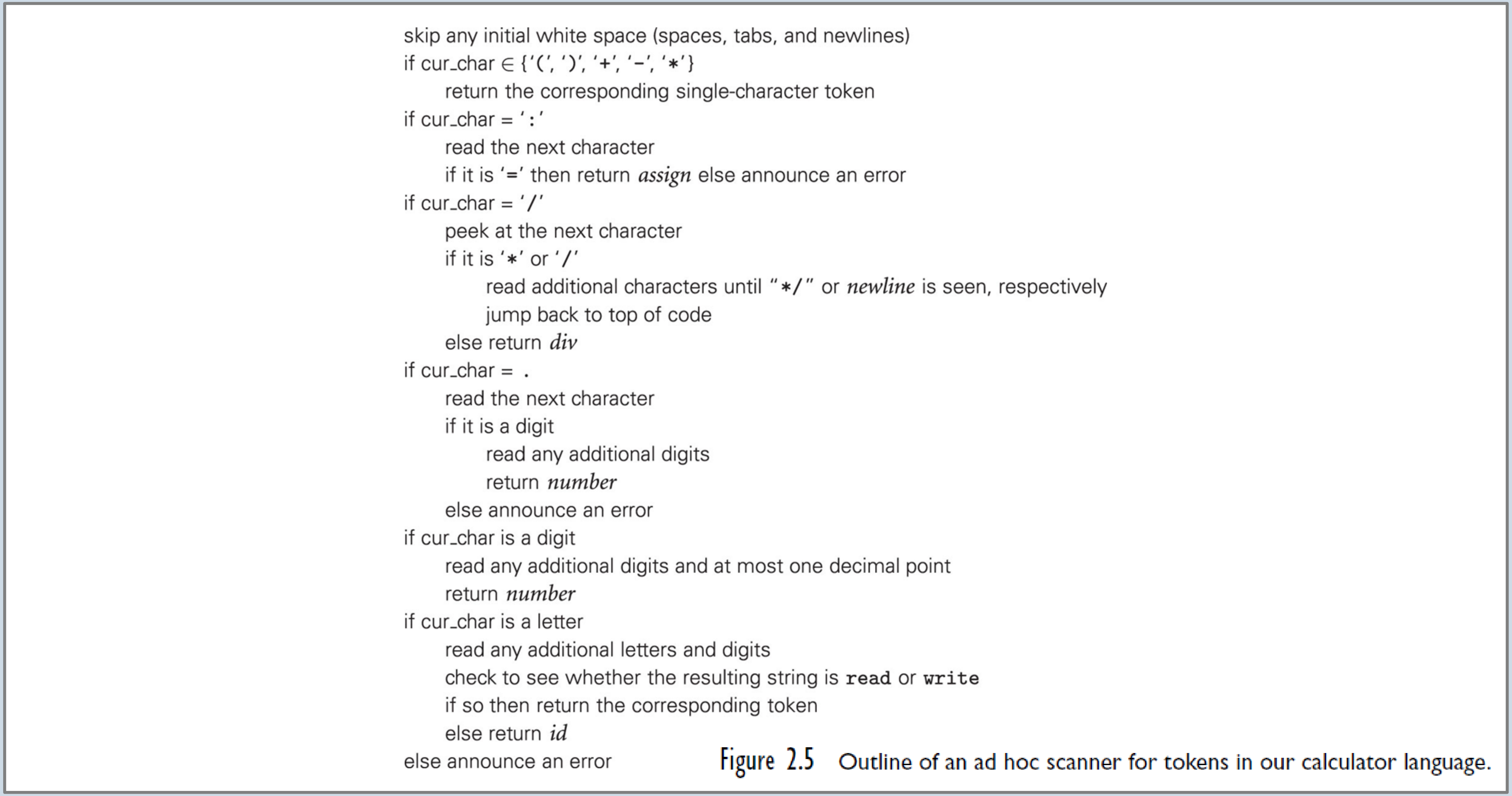
- Scanners tend to be built three ways
  - ad-hoc
  - semi-mechanical pure DFA
    (usually realized as nested case statements)
  - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

An Ad hoc Scanner

- After the scanner returns a token to the parser. When invoked again it repeats the algorithm from the beginning, using the next available characters of input (including any that were peeked at but not consumed the last time).
- As a rule, we accept the longest possible token in each invocation of the scanner. Thus foobar is always foobar and never f or foo or foob. More to the point, in a language like C, 3.14159 is a real number and never 3, ., and 14159. White space is generally ignored, except to the extent that it separates tokens.

An Ad hoc Scanner

- Figure 2.5 could be extended to outline a scanner for a larger programming language. The result could create code in some implementation language.
- Production compilers often use ad hoc scanners. During development, however, it is usually preferable to build a scanner in a structured way, as an explicit representation of a finite automaton.
- Finite automata can be generated automatically from a set of regular expressions, making it easy to regenerate a scanner when token definitions change. [LEX/FLEX]

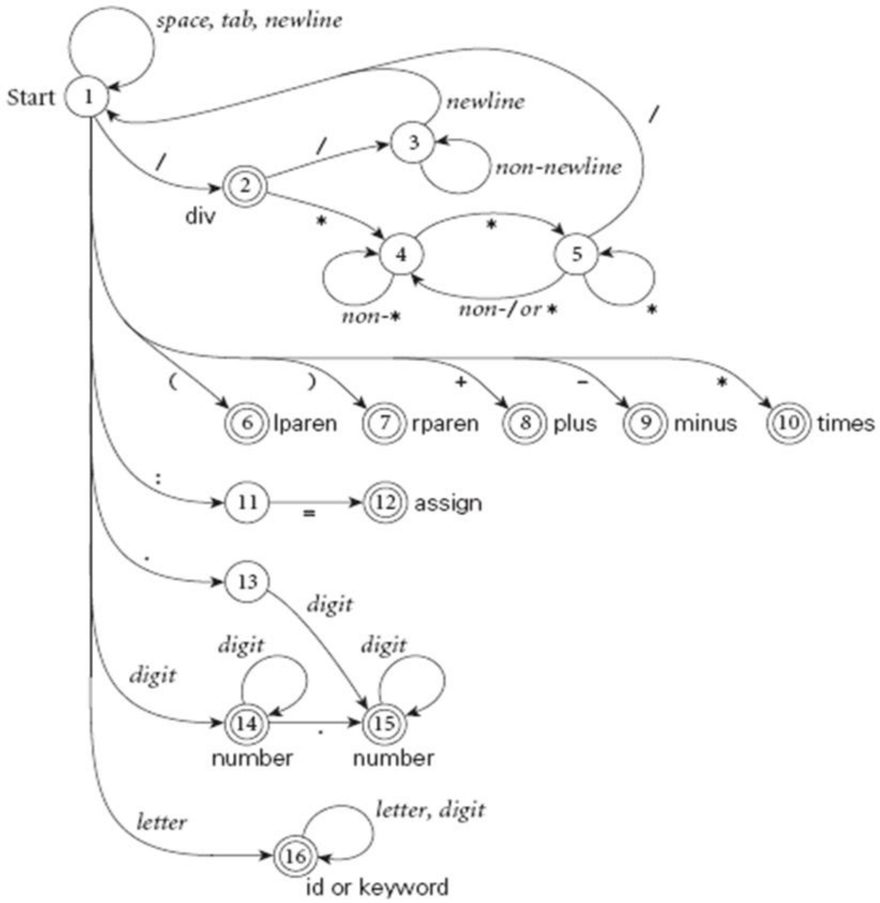


```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {‘(’, ‘)’, ‘+’, ‘−’, ‘*’}
    return the corresponding single-character token
if cur_char = ‘:’
    read the next character
    if it is ‘=’ then return assign else announce an error
if cur_char = ‘/’
    peek at the next character
    if it is ‘*’ or ‘/’
        read additional characters until “*/” or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

Figure 2.5 Outline of an ad hoc scanner for tokens in our calculator language.

Scanner for Calculator Tokens

in the form of a finite automaton (Case Study)



comment ⟶ /* (non-* | * non-/)* ** /
| // (non-newline)* newline

assign ⟶ :=

plus ⟶ +

minus ⟶ -

times ⟶ *

div ⟶ /

lparen ⟶ (

rparen ⟶)

id ⟶ letter (letter | digit)*
except for **read** and **write**

number ⟶ digit digit* | digit* (. digit | digit .) digit*

Tokenization by Ad hoc Scanning

SECTION 4

Ad hoc Scanner Implementation

System Variables

LexScan.py

```
DEBUG_MODE = False
TYPE   = { # Token Types
    "div": 'DIV',
    "lparen": 'LPAREN',
    "rparen": 'RPAREN',
    "plus": 'PLUS',
    "minus": 'MINUS',
    "times": 'TIMES',
    "assign": 'ASSIGN',
    "number": 'NUM',
    "id": "ID"
}
```

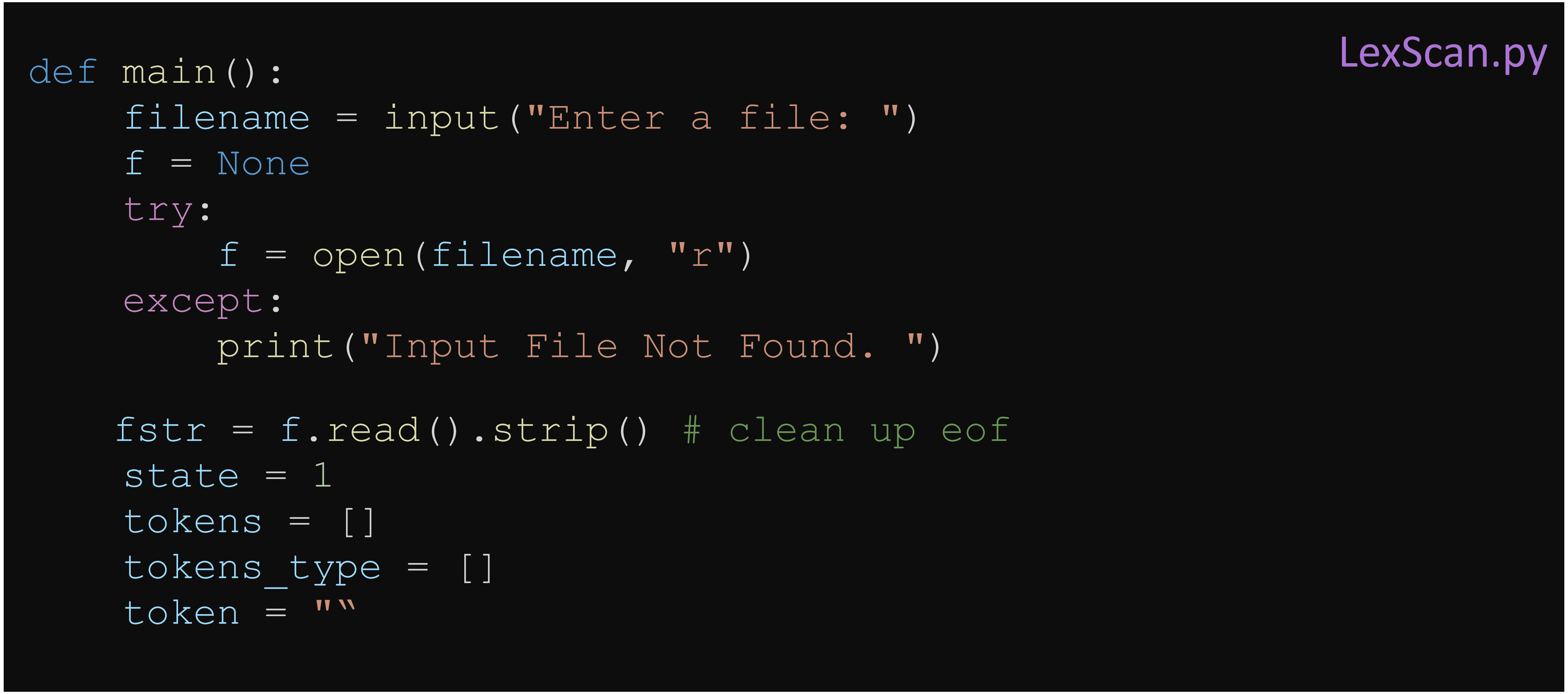
Ad hoc Scanner Implementation

System Variables

- **Filename:** Input file name for the Calculator Language
- **fstr:** raw input stream
- **state:** state variable for the DFA
- **tokens:** result tokens
- **tokens_type:** result token types
- **token:** temporary token buffer

Ad hoc Scanner Implementation

System Variables



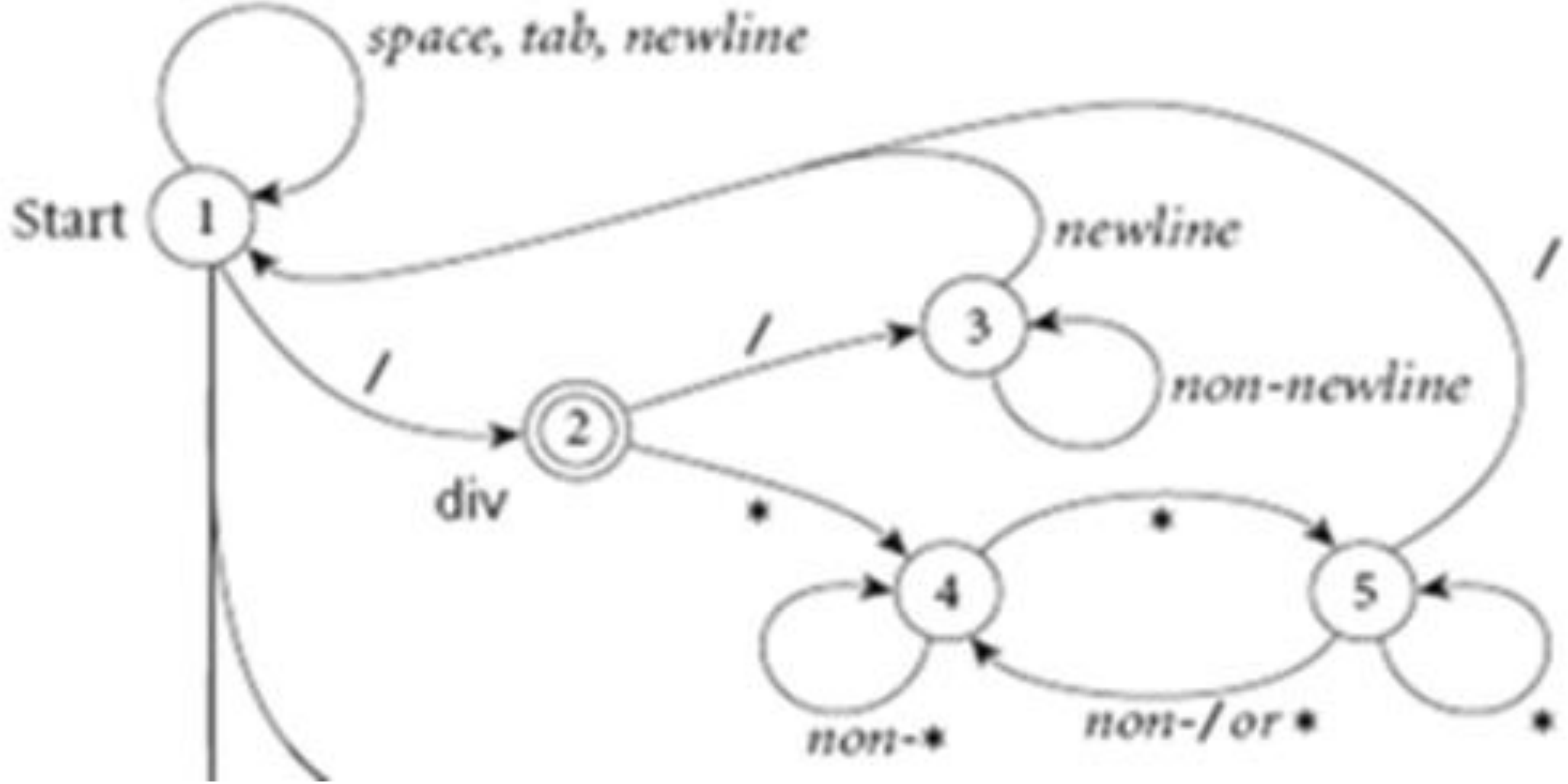
LexScan.py

```python
def main():
    filename = input("Enter a file: ")
    f = None
    try:
        f = open(filename, "r")
    except:
        print("Input File Not Found. ")

    fstr = f.read().strip() # clean up eof
    state = 1
    tokens = []
    tokens_type = []
    token = ""
```

Main Scanner For Loop

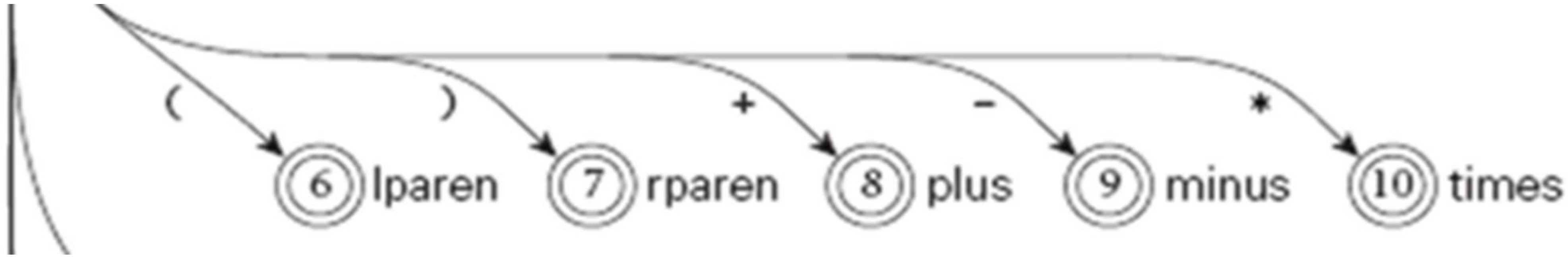
For each symbol from the raw input stream:

1. Evaluate (state, ch) to determine next state and tokenization
2. Some next state will be depending on the 1-symbol look-ahead or end of file symbol
3. Tokenize if final-state of a token reached. Reset state variable and the token buffer.


space, tab, newline
Start
1
newline
/
/
3
non-newline
/
2
div
*
*
4
5
non-*
non-/ or *
*

LexScan.py

```
for i in range(len(fstr)):
    ch = fstr[i]
    if DEBUG_MODE: print("Processing...", "(state=%d, ch=\'%s\')"% (state, ch))
    if (state, ch)==(1, '/'):
        token += ch
        if i!=len(fstr)-1:
            if fstr[i+1] in '/*': # look ahead
                state = 2
                continue
            else: # div
                terminate_with(tokens, tokens_type, token, 'div')
                state =1
                token=""
                continue
        else:
            report("E1: / symbol at the end of line error!")
            return
```

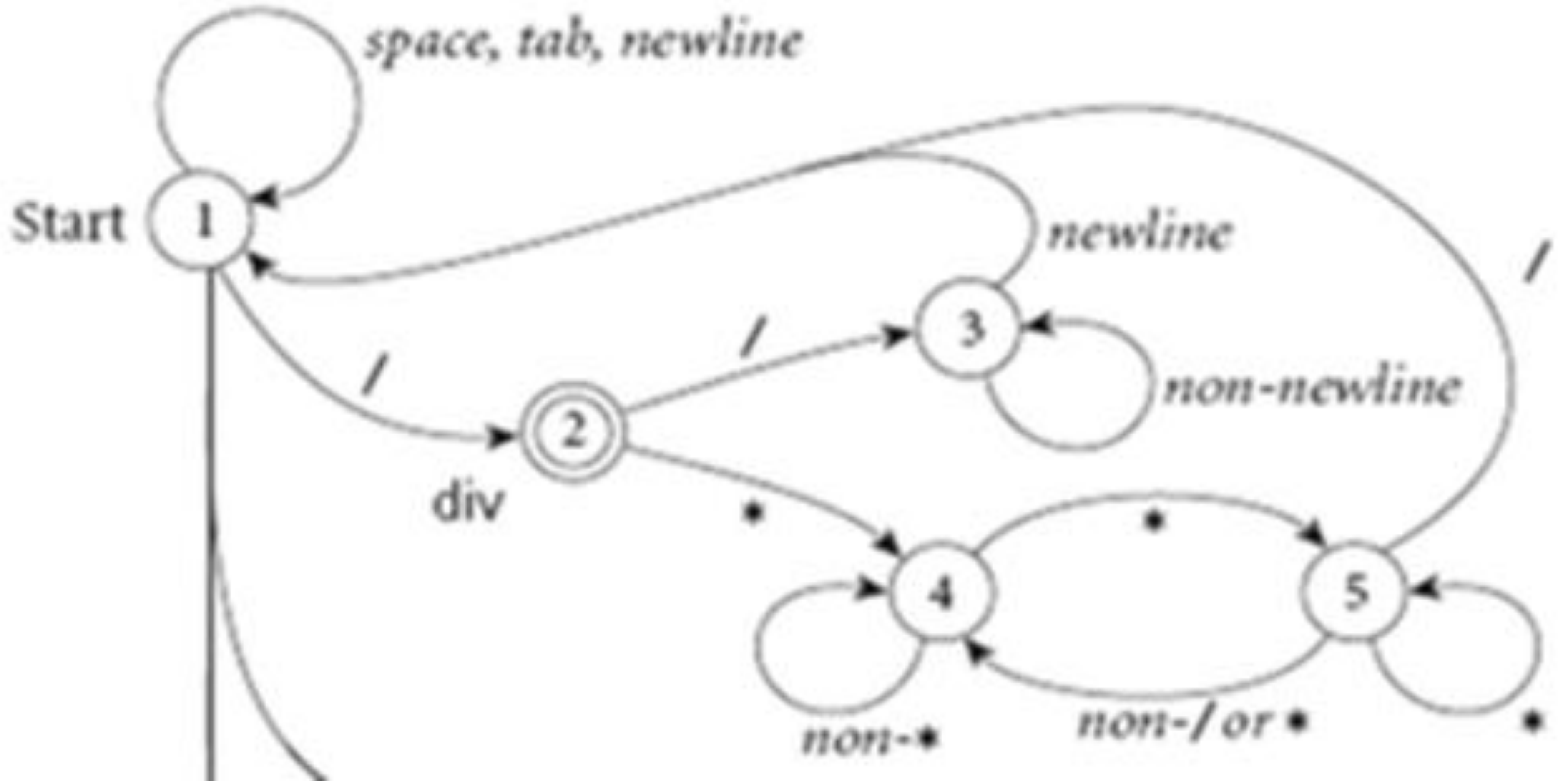

(
)
+
−
*
6
lparen
7
rparen
8
plus
9
minus
10
times

LexScan.py

```
    elif (state, ch)==(1, '('):  # (
        token += ch
        terminate_with(tokens, tokens_type, token, 'lparen')
        state = 1
        token = ""
        continue
    elif (state, ch)==(1, ')'):  # )
        token += ch
        terminate_with(tokens, tokens_type, token, 'rparen')
        state = 1
        token = ""
        continue
    elif (state, ch)==(1, '+'):  # +
        token += ch
        terminate_with(tokens, tokens_type, token, 'plus')
        state = 1
        token = ""
        continue
    elif (state, ch)==(1, '-'):  # -
        token += ch
        terminate_with(tokens, tokens_type, token, 'minus')
        state = 1
        token = ""
        continue
```

LexScan.py

```python
elif (state, ch)==(1, '*'):  # *
    token += ch
    terminate_with(tokens, tokens_type, token, 'times')
    state = 1
    token = ""
    continue
elif (state, ch)==(1, ':'):
    token += ch
    state = 11
    continue
elif (state, ch)==(11, '='):
    token += ch
    terminate_with(tokens, tokens_type, token, 'assign')
    state = 1
    token = ""
    continue
elif state==11:
    report("E2: : not followed by = error")
    return
```

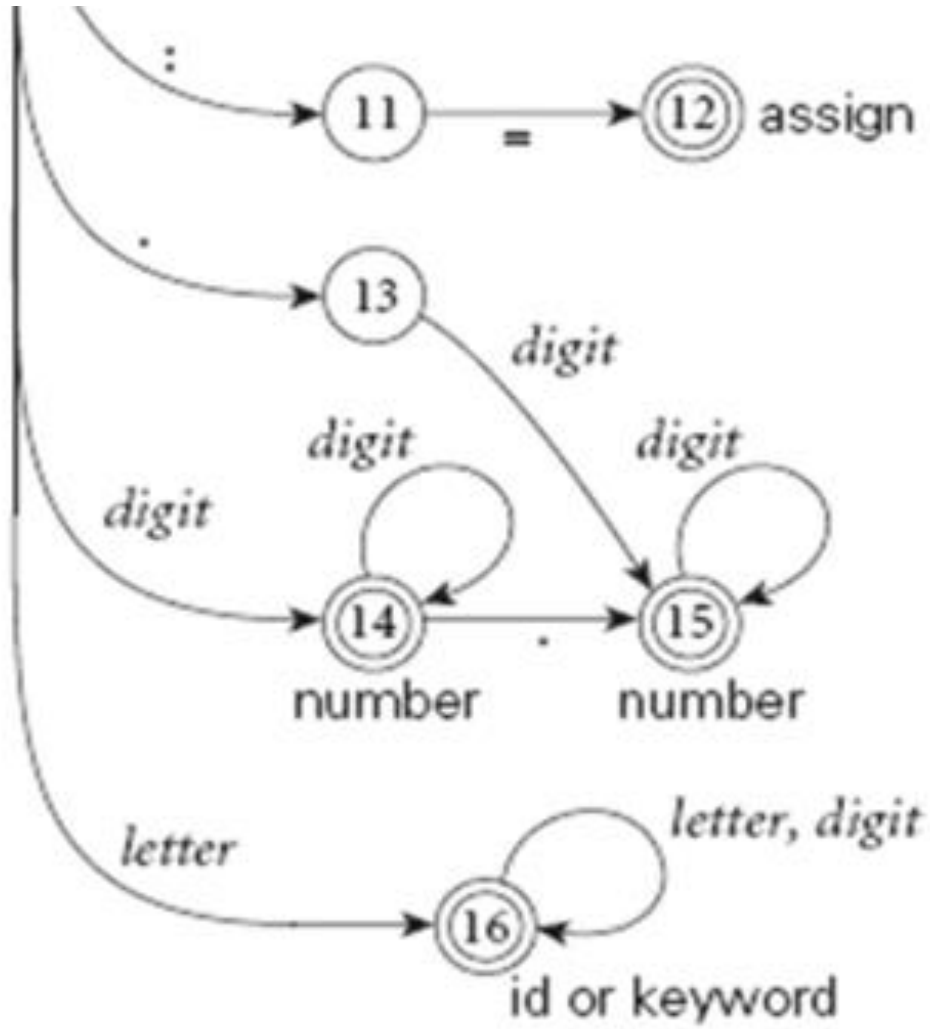

space, tab, newline
Start
1
/
2
div
/
3
newline
non-newline
/
*
4
*
5
non-*
non-/ or *
*

LexScan.py

```
        elif state==1 and (ch==" " or ch=="\t" or ch=="\n"):
            token = ""
            state = 1
            continue
        elif (state, ch)==(2, '/'):
            token=""
            state = 3
            continue
        elif (state, ch)==(2, '*'):
            token=""
            state = 4
            continue
        elif (state, ch)==(3, '\n'):
            token=""
            state = 1
            continue
        elif state==3:
            token=""
            continue
```

LexScan.py

```
        elif (state, ch)==(4, '*'):
            token=""
            state = 5
            continue
        elif state==4:
            token=""
            continue
        elif (state, ch)==(5, '/'):
            token=""
            state = 1
            continue
        elif (state, ch)==(5, '*'):
            continue
        elif state==5:
            token=""
            state = 4
            continue
```


:
11
=
12
assign
.
13
digit
digit
digit
digit
14
.
15
number
number
letter
letter, digit
16
id or keyword

LexScan.py

```python
        elif (state, ch)==(1, '.'):
            token=""
            state = 13
            continue
        elif state==13 and ch.isdigit():
            token += ch
            state = 15
            continue
        elif state==13:
            report("E4: Number Format Error")
            return
        elif state==1 and ch.isdigit():
            token += ch
            if i==len(fstr)-1 or not fstr[i+1].isdigit():
                terminate_with(tokens, tokens_type, token, 'number')
                state = 1
                token=""
            else:
                state = 14
            continue
```

LexScan.py

```
        elif state==14:
            if ch.isdigit():
                token += ch
                if i==len(fstr)-1 or not fstr[i+1].isdigit():
                    terminate_with(tokens, tokens_type, token, 'number')
                    state = 1
                    token=""
                else:
                    state = 14
                continue
            if ch=='.':
                token += ch
                state = 15
                continue
            else:
                report("E3: : number format error")
        elif state==15:
            token += ch
            if i==len(fstr)-1 or not fstr[i+1].isdigit():
                terminate_with(tokens, tokens_type, token, 'number')
                state = 1
                token=""
            else:
                state = 15
            continue
```

LexScan.py

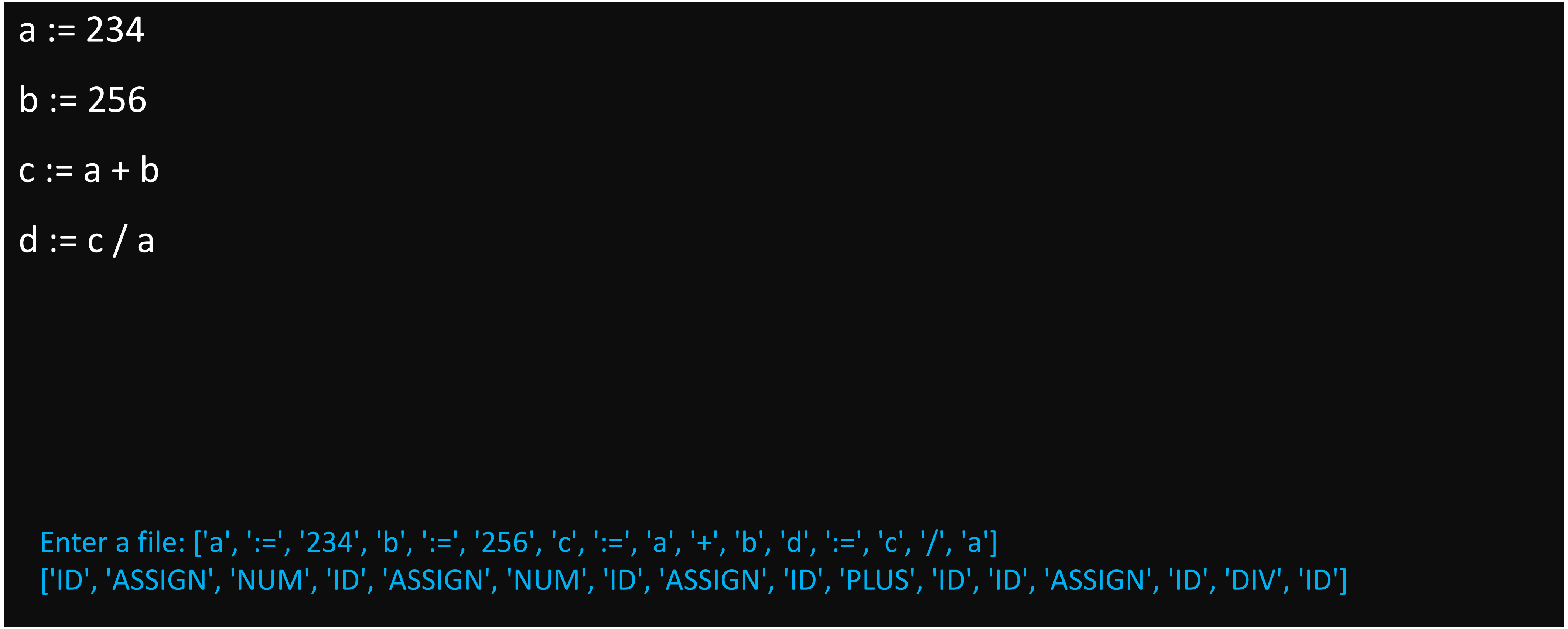
```
        elif state==1 and ch.isalpha():
            token += ch
            if i==len(fstr)-1 or not (fstr[i+1].isalpha() or fstr[i+1].isdigit()):
                terminate_with(tokens, tokens_type, token, 'id')
                state = 1
                token=""
                continue
            state = 16
            continue
        elif state==16 and (ch.isalpha() or ch.isdigit()):
            token += ch
            if i==len(fstr)-1 or not (fstr[i+1].isalpha() or fstr[i+1].isdigit()):
                terminate_with(tokens, tokens_type, token, 'id')
                state = 1
                token=""
            continue
        elif state==16:
            state = 1
            token=""
            continue
```

Result: a.cal

```
/* Comments */

a := 1   // useless comments

b := 2

c := (a + b)
```

```
Enter a file: ['a', ':=', '1', 'b', ':=', '2', 'c', ':=', '(', 'a', '+', 'b', ')']
['ID', 'ASSIGN', 'NUM', 'ID', 'ASSIGN', 'NUM', 'ID', 'ASSIGN', 'LPAREN', 'ID', 'PLUS', 'ID', 'RPAREN']
```

Result: b.cal



```
a := 234

b := 256

c := a + b

d := c / a
```

```
Enter a file: ['a', ':=', '234', 'b', ':=', '256', 'c', ':=', 'a', '+', 'b', 'd', ':=', 'c', '/', 'a']
['ID', 'ASSIGN', 'NUM', 'ID', 'ASSIGN', 'NUM', 'ID', 'ASSIGN', 'ID', 'PLUS', 'ID', 'ID', 'ASSIGN', 'ID', 'DIV', 'ID']
```

Tokenization by Regular Expression

SECTION 5

Grammar

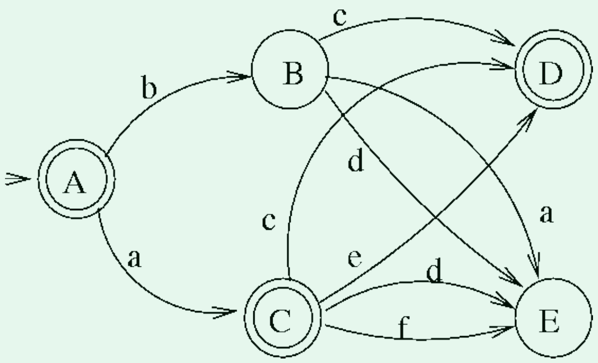
$$\begin{aligned} S &\rightarrow aS \mid bX \\ X &\rightarrow aX \mid bY \\ Y &\rightarrow aY \mid bZ \mid \Lambda \\ Z &\rightarrow aZ \mid \Lambda \end{aligned}$$

Formal

Language Input

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

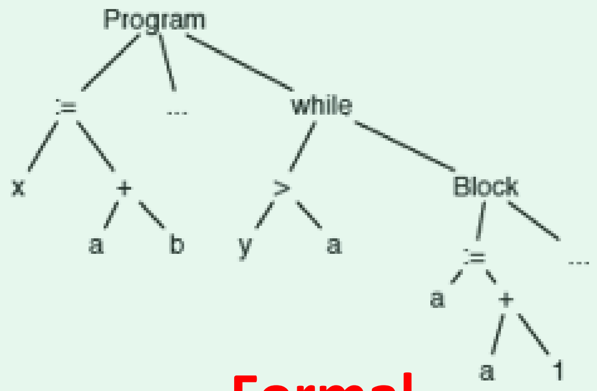
Informal



Formal

State Machine

Validated Tokens Syntax Tree



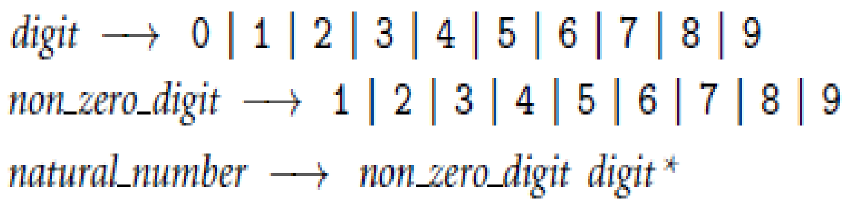
Formal

Target Code

Machine Code
Assembly
Byte Code
Object Code

Informal

Syntax of Arabic numerals



$$
\begin{aligned}
&\mathit{digit} \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\\
&\mathit{non\_zero\_digit} \longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\\
&\mathit{natural\_number} \longrightarrow \mathit{non\_zero\_digit}\ \mathit{digit}^{\,*}
\end{aligned}
$$

Alphabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
| : options
* : Kleene star *, zero or more repetition

Similar syntax rules and semantic interpretations can be devised for rational numbers, (limited-precision) real numbers, arithmetic, assignments, control flow, declarations, and indeed all of programming languages.

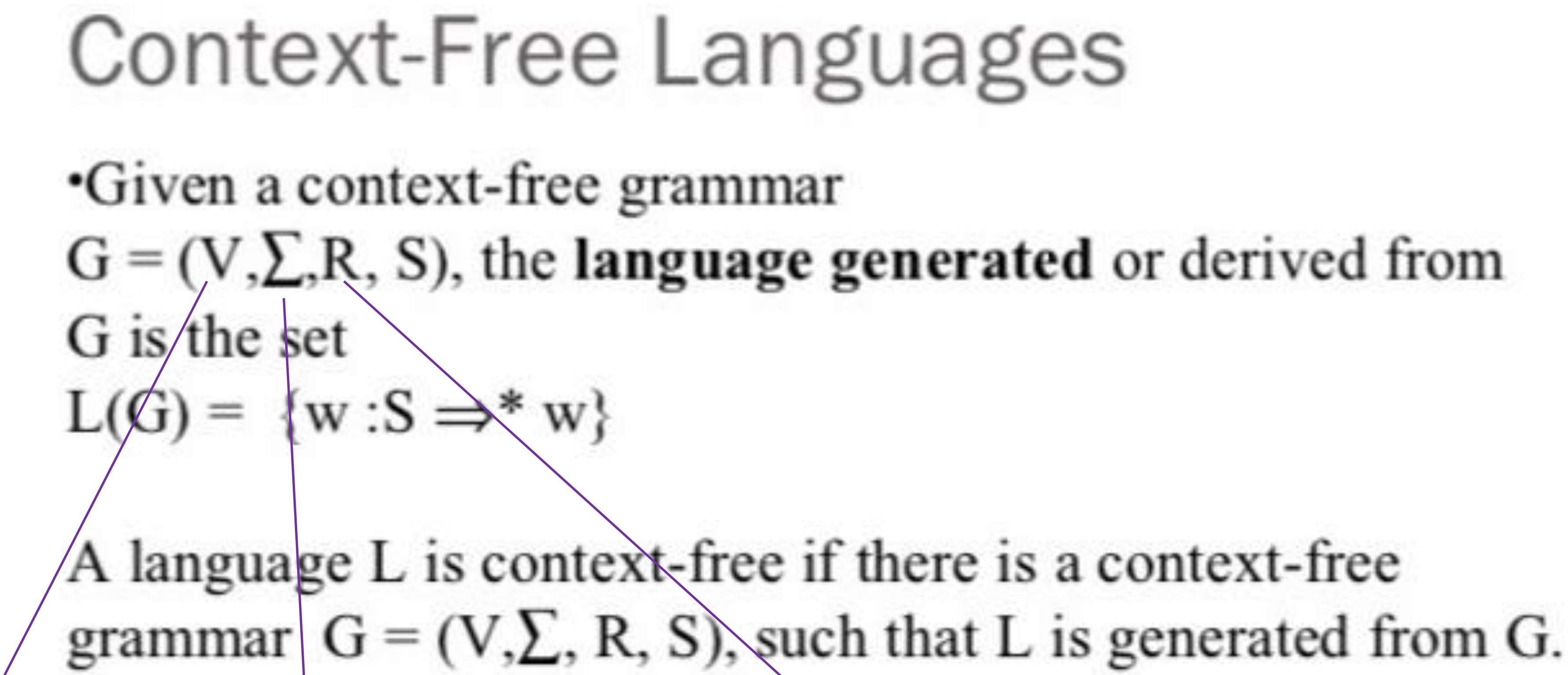
- A **regular language** over an alphabet Σ is one that contains either a single string of length 0 or 1, or strings which can be obtained by using the operations of union, concatenation, or Kleene* on strings of length 0 or 1.

- **Operations on formal languages:**

  Let L_1 = {10} and L_2 = {011, 11}.

  - Union: $L_1 \cup L_2$ = {10, 011, 11}
  - Concatenation: $L_1 L_2$ = {10011, 1011}
  - Kleene Star: L_1^{*} = {λ, 10, 1010, 101010, … }

  Other operations: intersection, complement, difference



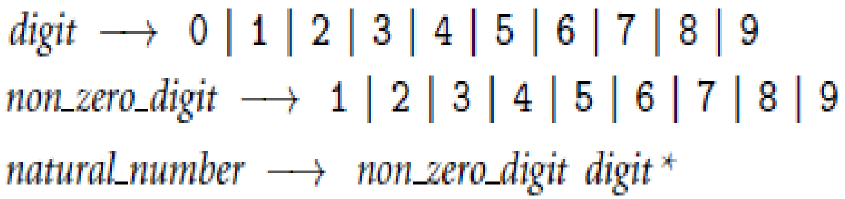
Context-Free Languages

•Given a context-free grammar $G = (V, \Sigma, R, S)$, the **language generated** or derived from G is the set $L(G) = \{w : S \Rightarrow^{*} w\}$

A language L is context-free if there is a context-free grammar $G = (V, \Sigma, R, S)$, such that L is generated from G.

digit
non-zero-digit
natural_numbers

Alphabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

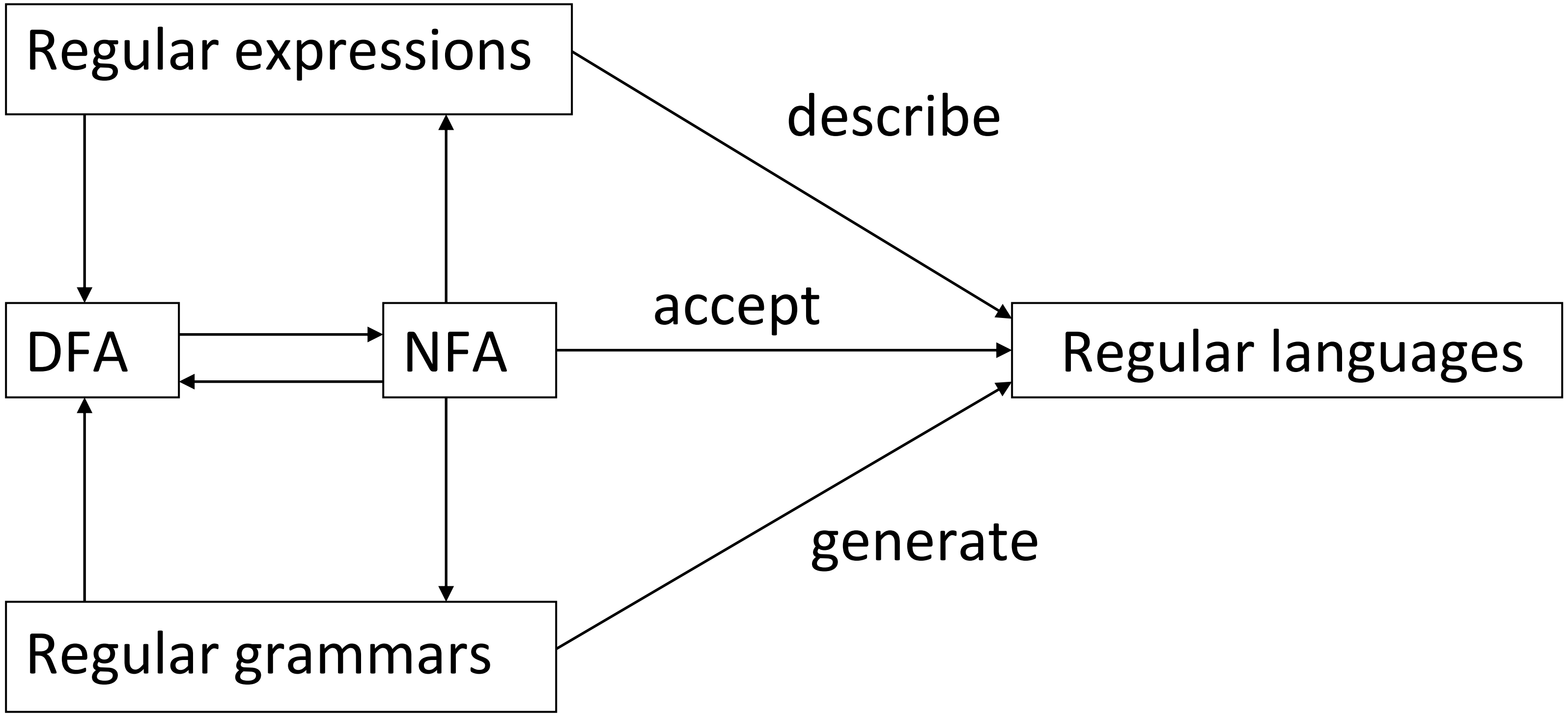


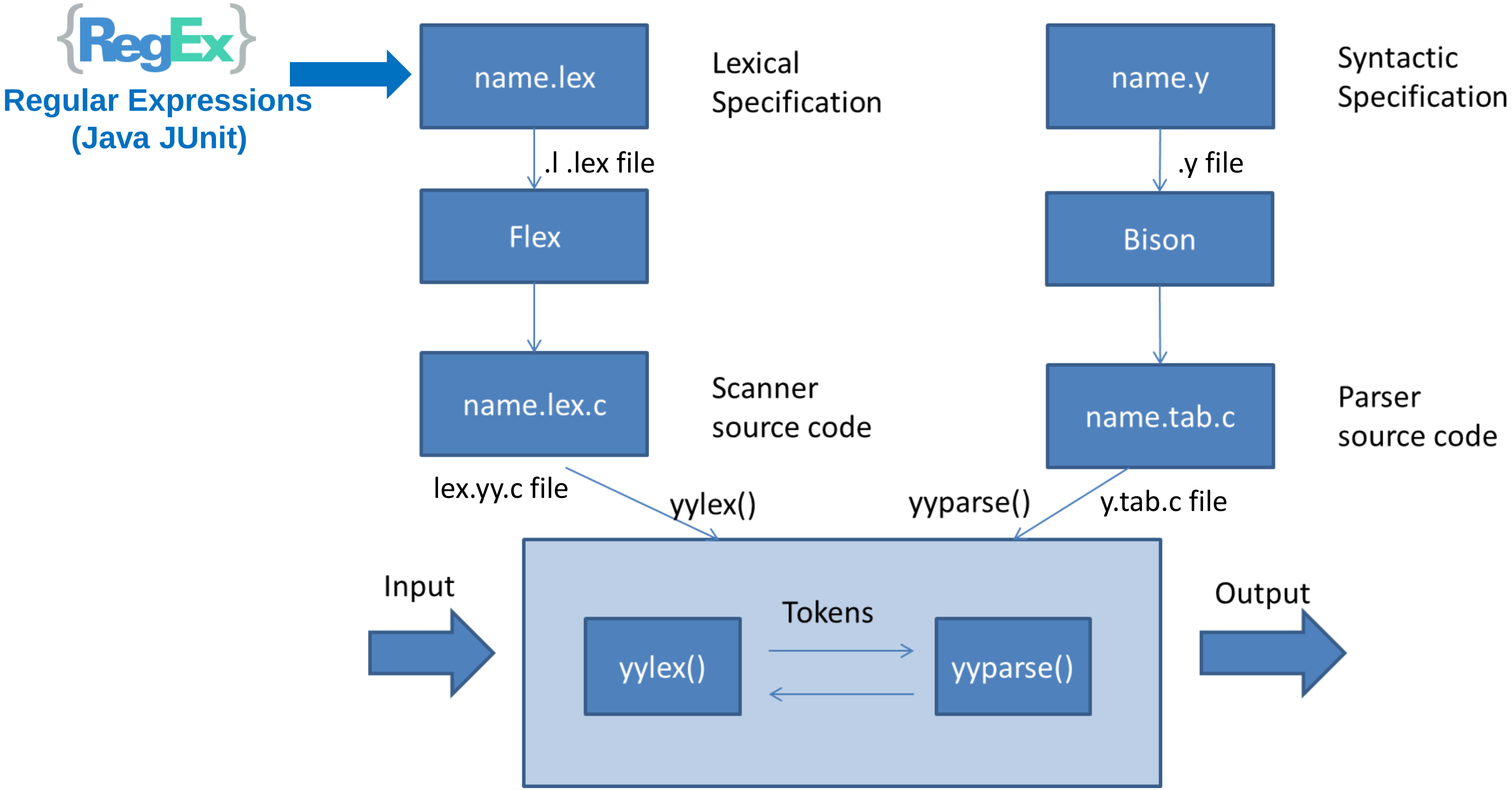
$$\begin{aligned} \textit{digit} &\longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \textit{non\_zero\_digit} &\longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \textit{natural\_number} &\longrightarrow \textit{non\_zero\_digit}\ \textit{digit}^{*} \end{aligned}$$

Expression, Grammar, Language

- Expression is a special patterns (for the tokens of a language)
- Grammar is a set of rules.
- Language is all the composition of symbol sequences generated by the grammar.

3 ways of specifying regular languages





RegEx

Regular Expressions (Java JUnit)

name.lex

Lexical Specification

.l .lex file

Flex

name.lex.c

Scanner source code

lex.yy.c file

yylex()

name.y

Syntactic Specification

.y file

Bison

name.tab.c

Parser source code

y.tab.c file

yyparse()

Input

yylex()

Tokens

yyparse()

Output

Why are we studying these?

- Formally, define a language using formal grammar. (avoidance of ambiguity.)
- Utilize the tools for syntax design in general programming. (Regular Expression, Lex, Yacc)
- Design a compiler. (Useful for software development including development tools, EDA, system automation)

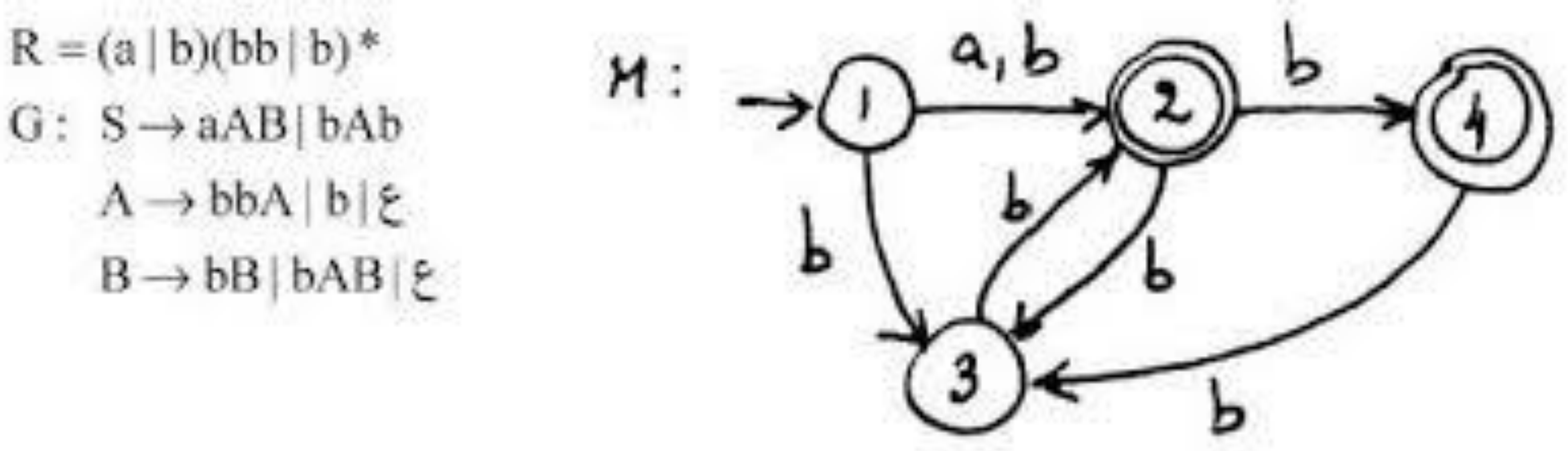
Scanning I
Finite Automata

SECTION 6

Finite Automata

3-in-1 R, G, M

Regular expressions, regular grammars and finite automata are simply three different formalisms for the same thing. There are algorithms to convert from any of them to any other.



$R = (a \mid b)(bb \mid b)^*$

$G: \; S \rightarrow aAB \mid bAb$

$A \rightarrow bbA \mid b \mid \varepsilon$

$B \rightarrow bB \mid bAB \mid \varepsilon$

A Formal Definition of Regular Grammars

Terminal, Non-terminal and Rules

A *regular grammar* is a mathematical object, *G*, with four components, *G* = (*N*, *Σ*, *P*, *S*), where

•*N* is a nonempty, finite set of *nonterminal symbols*,

•*Σ* is a finite set of *terminal symbols* , or *alphabet, symbols,*

•*P* is a set of grammar rules, each of one having one of the forms

- $A \rightarrow aB$
- $A \rightarrow a$
- $A \rightarrow \varepsilon$, for *A, B* ∈ *N, a* ∈ *Σ*, and *ε* the empty string, and

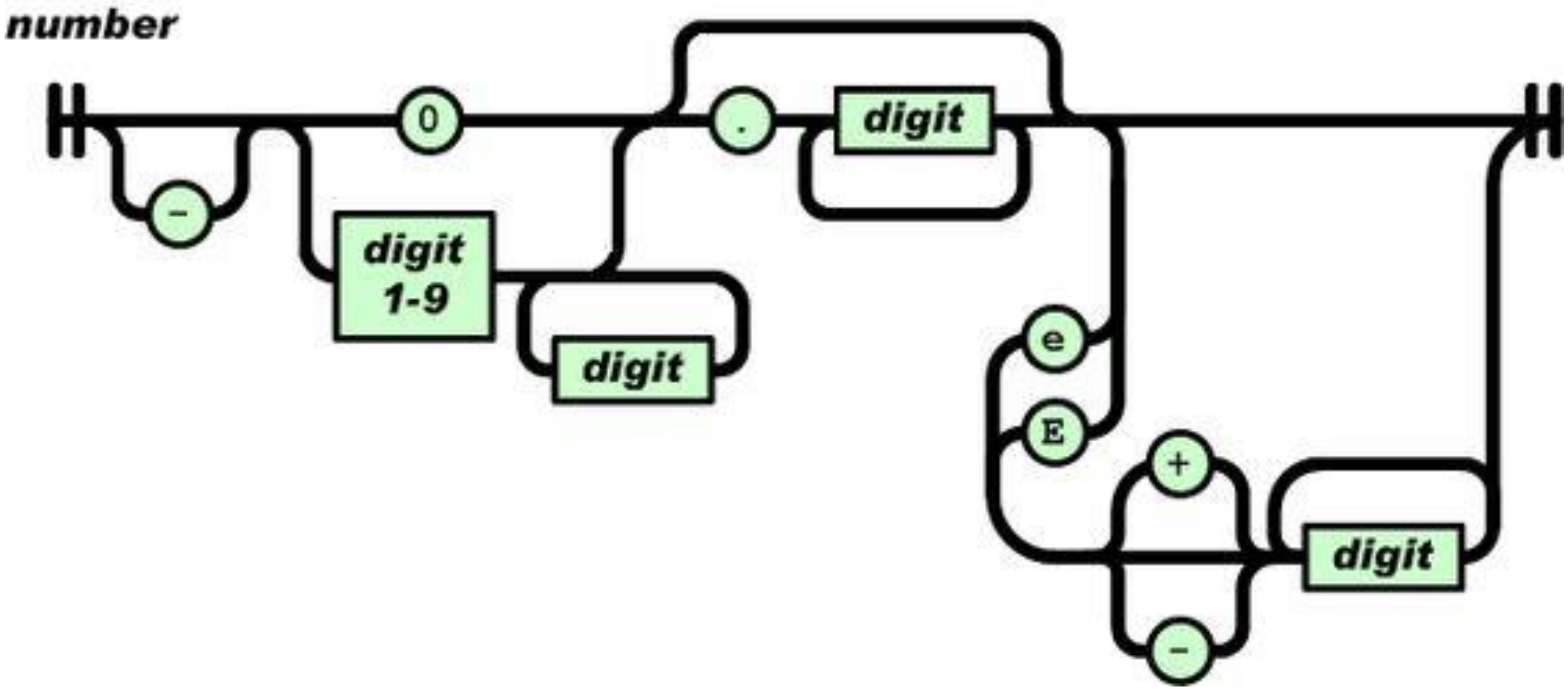
•*S* ∈ *N* is the *start symbol*.

Notice that this definition captures all of the components of a regular grammar that we have heretofore identified (*heretofor*, how often does one get to use that word?).

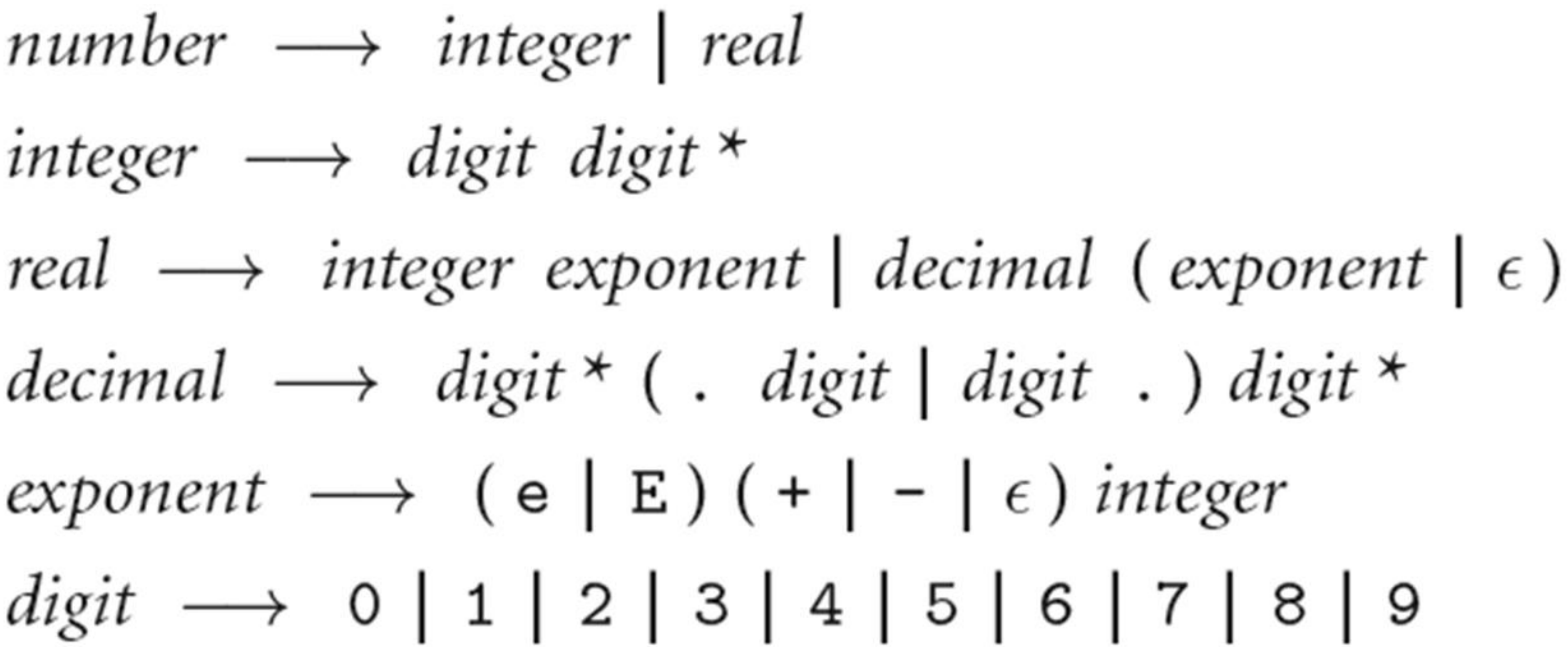
A regular expression is one of the following

- A character
- The empty string, denoted by ε
- Two regular expressions concatenated
- Two regular expressions separated by | (i.e., or)
- A regular expression followed by the Kleene star * (concatenation of zero or more strings)

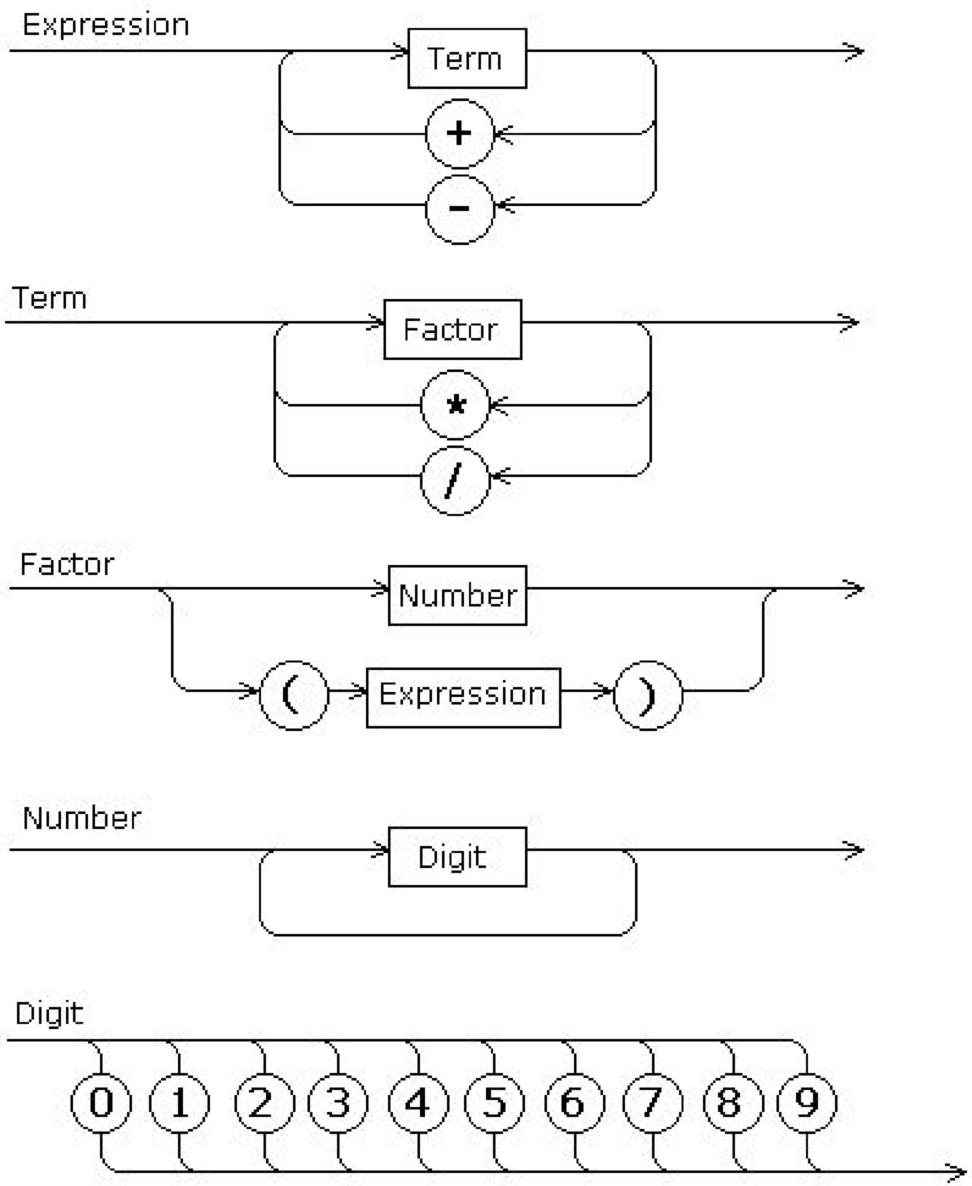
Backus-Naur Form for Number



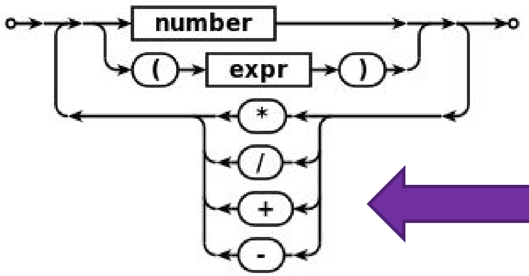
Numerical constants accepted by a simple hand-held calculator



$$
\begin{aligned}
\textit{number} &\longrightarrow \textit{integer} \mid \textit{real} \\
\textit{integer} &\longrightarrow \textit{digit} \;\, \textit{digit}\,^{*} \\
\textit{real} &\longrightarrow \textit{integer} \;\, \textit{exponent} \mid \textit{decimal} \;\, (\, \textit{exponent} \mid \epsilon \,) \\
\textit{decimal} &\longrightarrow \textit{digit}\,^{*} \; (\, \texttt{.} \;\, \textit{digit} \mid \textit{digit} \;\, \texttt{.} \,) \; \textit{digit}\,^{*} \\
\textit{exponent} &\longrightarrow (\, \texttt{e} \mid \texttt{E} \,) \; (\, \texttt{+} \mid \texttt{-} \mid \epsilon \,) \; \textit{integer} \\
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

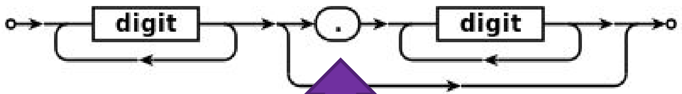

Expression
Term
+
-
Term
Factor
*
/
Factor
Number
(
Expression
)
Number
Digit
Digit
0
1
2
3
4
5
6
7
8
9

expr:



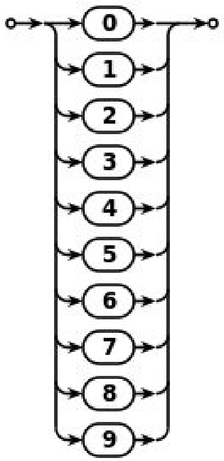
No precedence

number:



With decimal point for floating point and integer

digit:

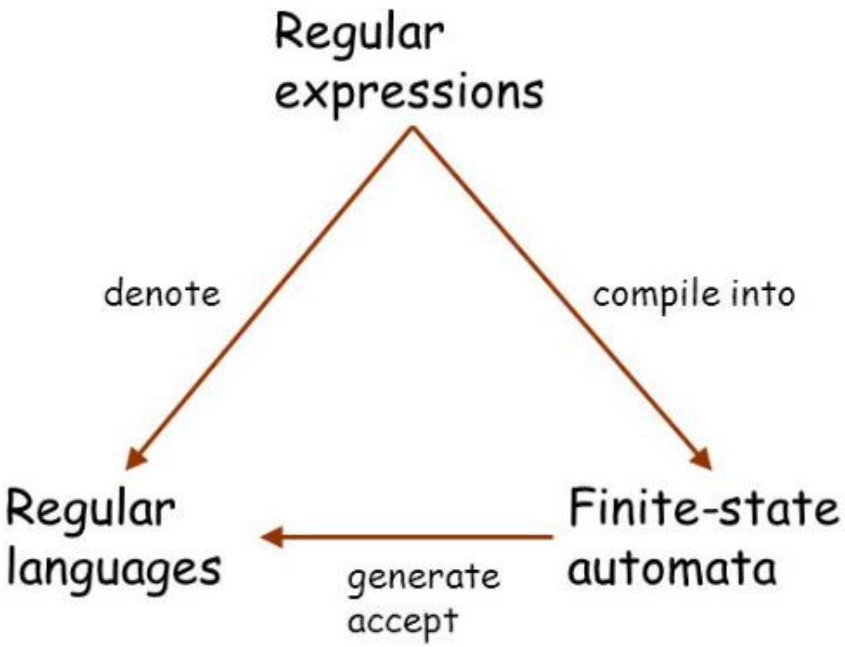


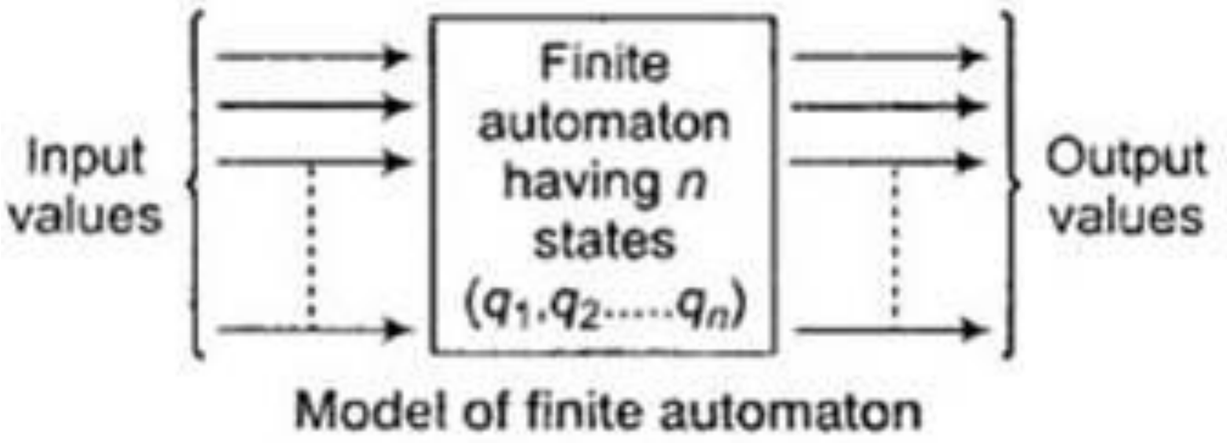
Scanning II

Scanner Design 1: Regex to NFA

SECTION 7

Regular Expression, Regular Languages, Finite State Automata, and Finite State Machine




Input values
Finite automaton having n states $(q_1, q_2, \ldots, q_n)$
Output values

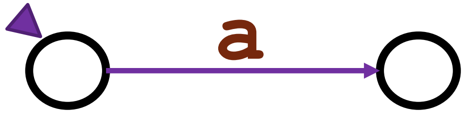
Model of finite automaton

Conversion of Regular Expression to Finite State Automaton

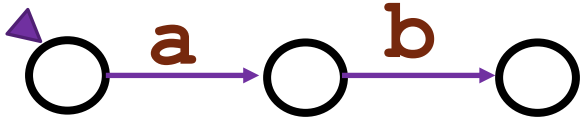
▲ Starting state

Regular Expression | **Finite Automaton**

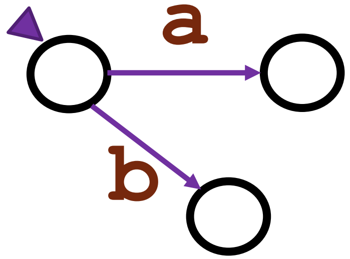
a



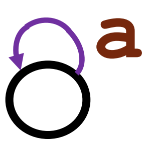
ab



a|b

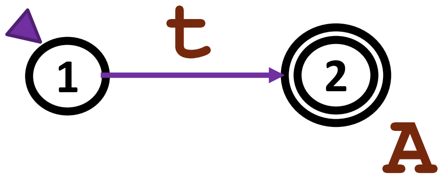


a*

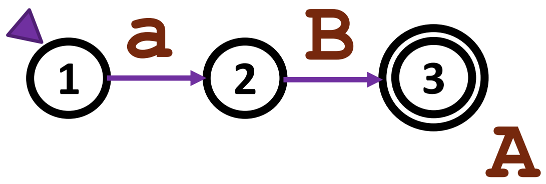


Regular Grammar Rules | **Finite Automaton**

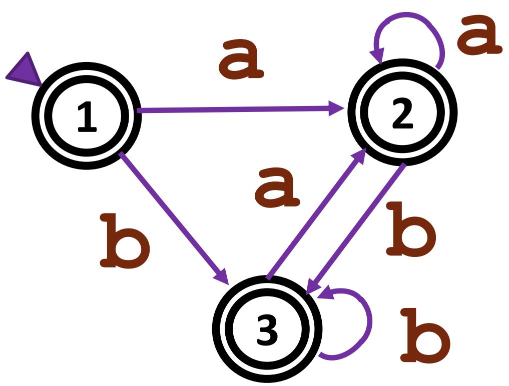
A -> t



A -> aB



A -> (a|b)*



Generating Finite Automaton

Three Step Conversion from Regular Expression to DFA

- The first step converts the regular expressions into a nondeterministic finite automaton (NFA).
- The second step is to convert nondeterministic finite automaton (NFA) to deterministic finite automaton(DFA).
- The third step is to minimize the DFA.

Non-deterministic Finite Automaton

The automaton has the desirable property that its actions are deterministic: in any given state with a given input character there is never more than one possible outgoing transition (arrow) labeled by that character. [DFA]

For NFA,

1. There may be more than one transition out of a given state labeled by a given character, and
2. There may be so-called epsilon transitions: arrows labeled but the empty string symbol, ε. The NFA is said to accept an input string(token) if there exists a path from the start state to a final state whose non-epsilon transitions are labeled, in order, by the characters of the token.

Constructing an NFA for a Given Regular Expression

A trivial regular expression consisting of a single character c is equivalent to a simple two-state NFA (in fact, a DFA), illustrated in part (a) of Figure 2.7 (next slide). Similarly, the regular expression ε is equivalent to a two-state NFA whose arc is labeled by ε.

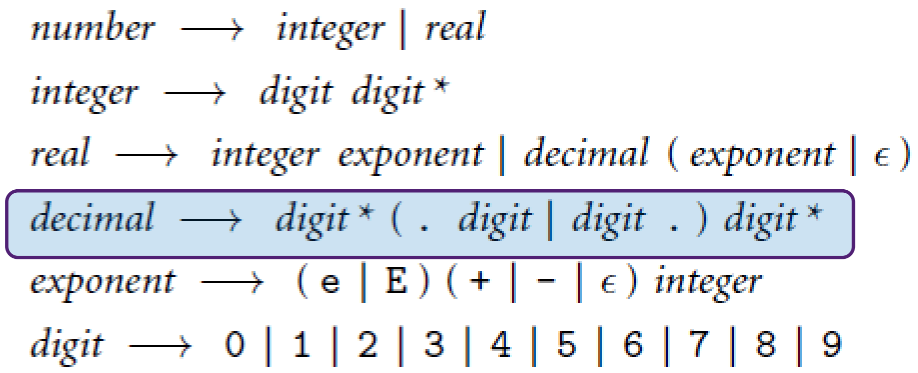
Starting with this base we can use three sub-constructions, illustrated in parts (b) through (d) of the same figure, to build larger NFAs to represent the concatenation, alternation, or Kleene closure of the regular expressions represented by smaller NFAs. Each step preserves three invariants: there are no transitions into the initial state, there is a single final state, and there are no transitions out of the final state.

These invariants allow smaller machines to be joined into larger machines without any ambiguity about where to create the connections, and without creating any unexpected paths.

NFA for d *(.d | d.) d *

- To make these constructions concrete, we consider a small but nontrivial example—the decimal strings of **Example 2.3**.
- These consist of a string of decimal digits containing a single decimal point. With only one digit, the point can come at the beginning or the end: `(.d | d.)`, where for brevity we use d to represent any decimal digit. Arbitrary numbers of digits can then be added at the beginning or the end: `d* (.d|d.) d*`.
- Starting with this regular expression and using the constructions of Figure 2.7, we illustrate the construction of an equivalent NFA in Figure 2.8.

Example 2.3:



number ⟶ *integer* | *real*

integer ⟶ *digit digit* *

real ⟶ *integer exponent* | *decimal* (*exponent* | ϵ)

decimal ⟶ *digit* * (. *digit* | *digit* .) *digit* *

exponent ⟶ (e | E) (+ | - | ϵ) *integer*

digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

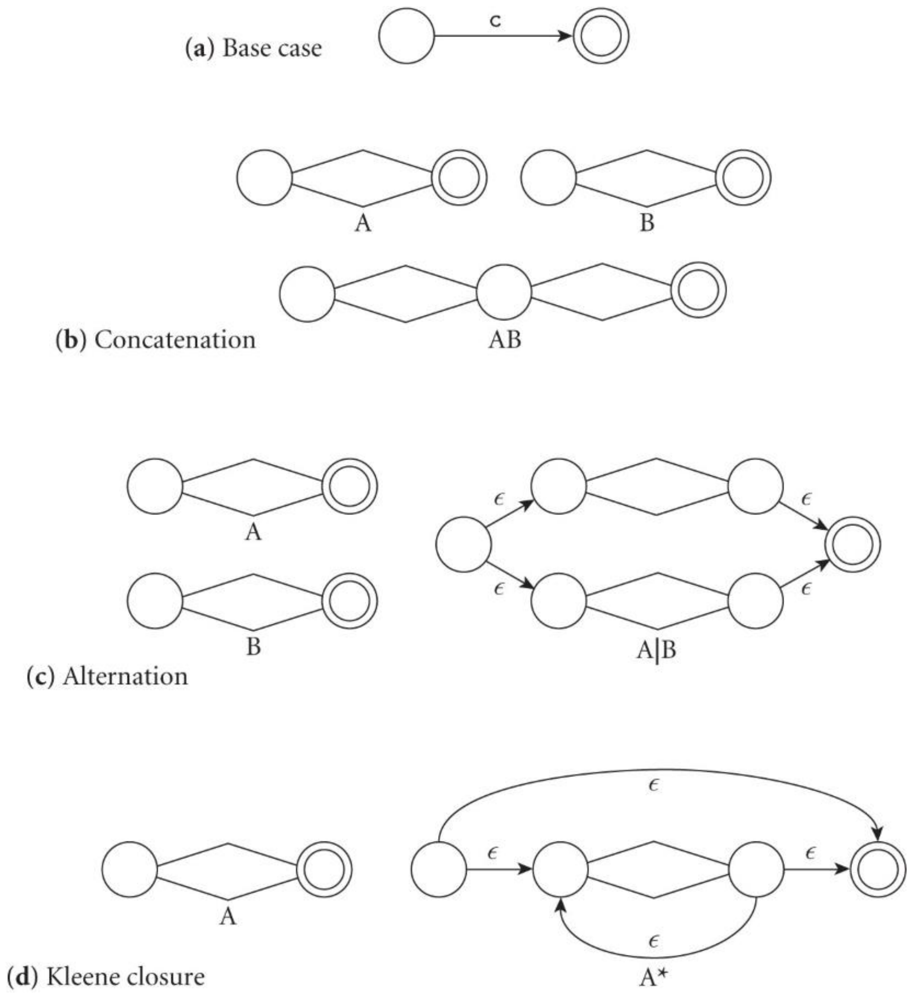


Figure 2.7 Construction of an NFA equivalent to a given regular expression. Part (a) shows the base case: the automaton for the single letter c. Parts (b), (c), and (d), respectively, show the constructions for concatenation, alternation, and Kleene closure. Each construction retains a unique start state and a single final state. Internal detail is hidden in the diamond-shaped center regions.

NFA Representation

$\textit{decimal} \longrightarrow \textit{digit}^* \; (\; . \; \textit{digit} \mid \textit{digit} \; . \;) \; \textit{digit}^*$

- To avoid such a complex and time-consuming strategy, we can use a “set of subsets” construction to transform the NFA into an equivalent DFA.
- The key idea is for the state of the DFA after reading a given input to represent the set of states that the NFA might have reached on the same input.

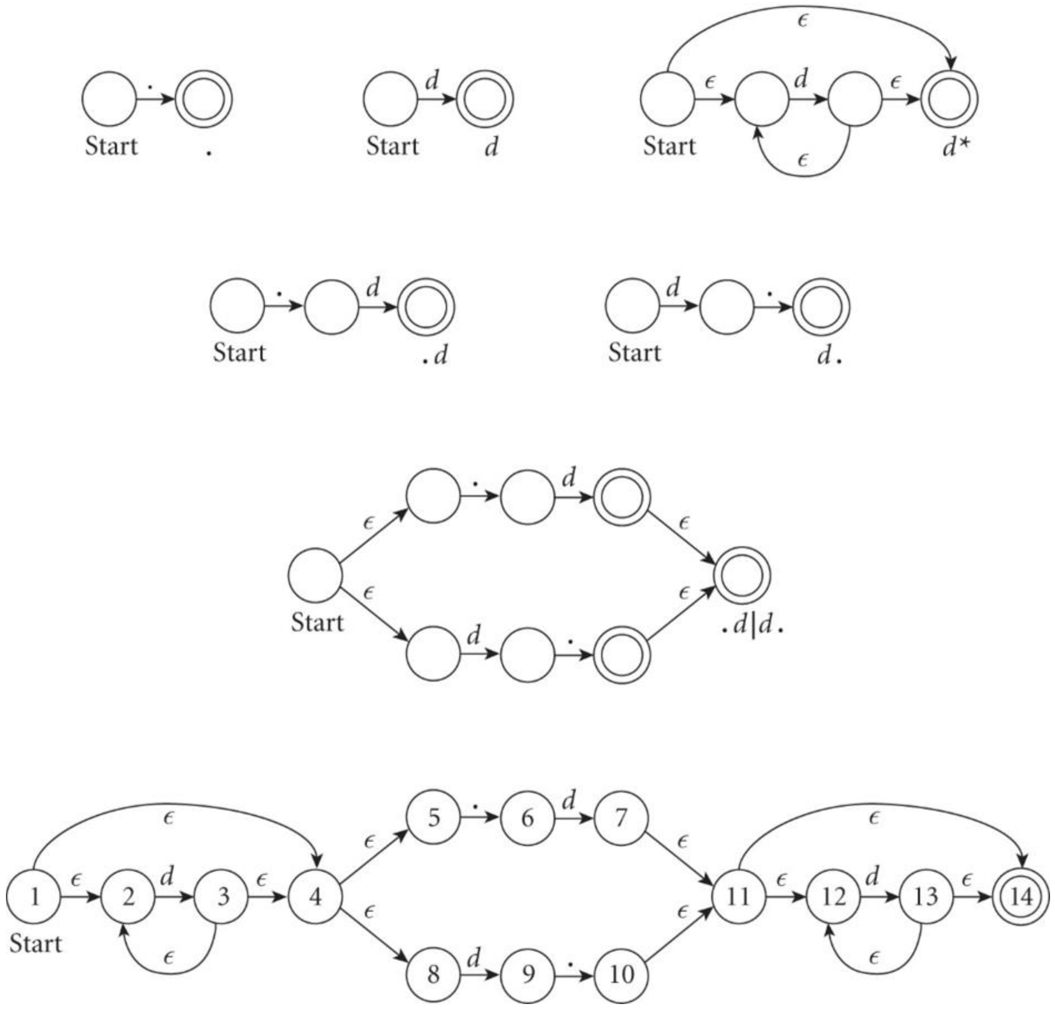


Figure 2.8 Construction of an NFA equivalent to the regular expression $d^*(\; . \; d \mid d \; . \;) \; d^*$. In the top row are the primitive automata for . and d, and the Kleene closure construction for d^*. In the second and third rows we have used the concatenation and alternation constructions to build $. \, d$, $d \, .$, and $(\; . \; d \mid d \; . \;)$. The fourth row uses concatenation again to complete the NFA. We have labeled the states in the final automaton for reference in subsequent figures.

Scanning III

Scanner Design 2: NFA to DFA

SECTION 8

NFA to DFA Conversion

Problem Statement

- Let X = (Qx, ∑, δx, q0, Fx) be an NFA which accepts the language L(X).

- We have to design an equivalent DFA Y = (Qy, ∑, δy, q0, Fy) such that L(Y) = L(X).

- If a NFA and a corresponding DFA accept the same language, then the NFA is equivalent to the DFA.

NFA to DFA Conversion

Algorithm

Input – An NFA

Output – An equivalent DFA

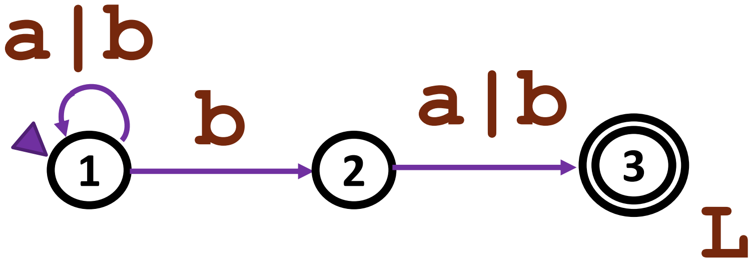
- **Step 1** – Create state table from the given NFA.
- **Step 2** – Create a blank state table under possible input alphabets for the equivalent DFA.
- **Step 3** – Mark the start state of the DFA by q0 (Same as the NFA).
- **Step 4** – Find out the combination of States $\{Q_0, Q_1, \ldots , Q_n\}$ for each possible input alphabet.
- **Step 5** – Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.
- **Step 6** – The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

Conversion Example

No-epsilon Case

A Language L is accepted by a NFA:

L -> (a | b)* b (a | b)



State Transition Table

	a	b
1	1	1 2
2	3	3
3	-	-

NFA - State Transition Table

	a	b
1	1	1 2
2	3	3
3	-	-

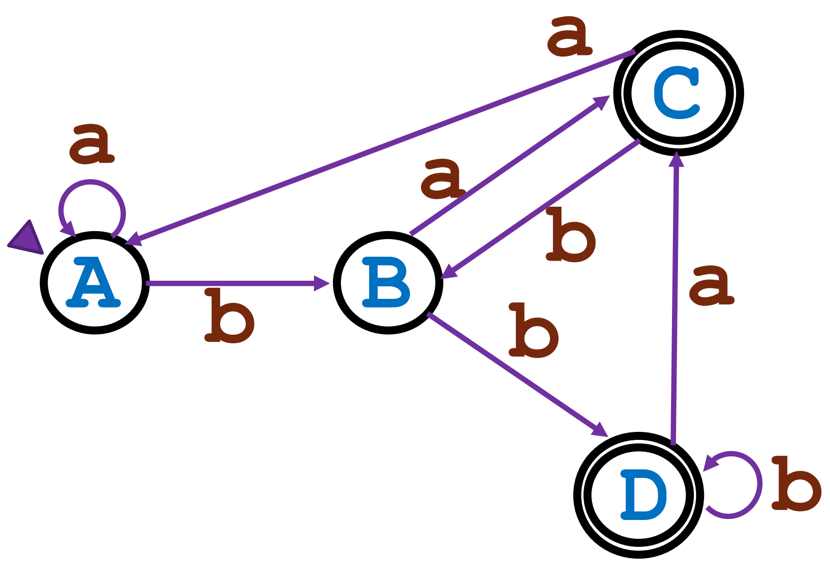
Target DFA - State Transition Table

DFA	NFA	a	b
A	1	1	1 2
B	1 2	1 3	1 2 3
C	1 3	1	1 2
D	1 2 3	1 3	1 2 3

DFA - State Transition Table

	a	b
A	A	B
B	C	D
C	A	B
D	C	D

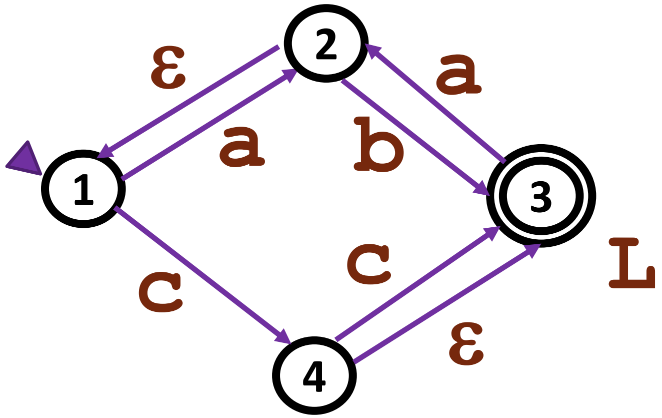
DFA



Conversion Example

With-epsilon Case

NFA with ε:



State Transition Table

	a	b	c	ε*
1	2	-	4	1
2	-	3	-	2 1
3	2	-	-	3
4	-	-	3	4 3

State Transition Table

	a	b	c	ε*
1	2	-	4	1
2	-	3	-	2 1
3	2	-	-	3
4	-	-	3	4 3

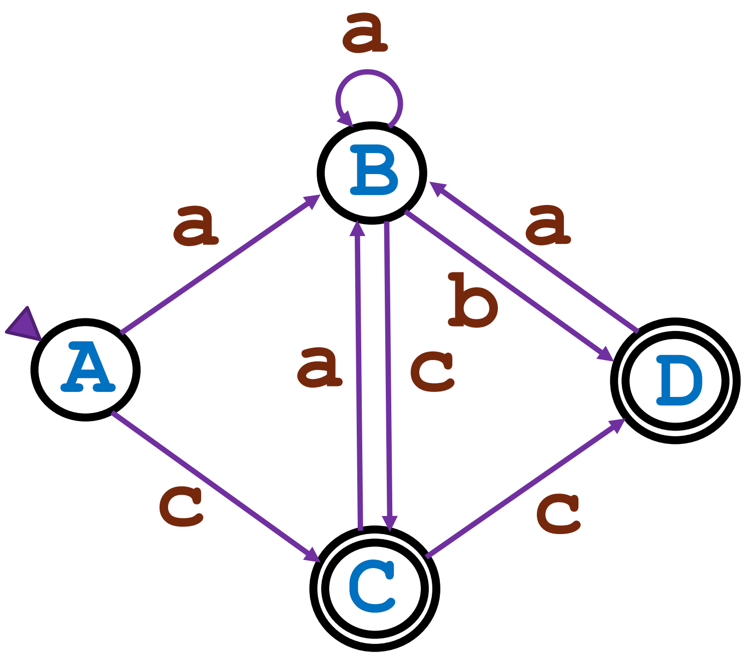
DFA State Transition Table

DFA	NFA	aε*	bε*	cε*
A	1	2 1	-	4 3
B	2 1	2 1	3	4 3
C	4 3	2 1	-	3
D	3	2 1	-	-

DFA State Transition Table

DFA	NFA	a	b	c
A	1	B	-	C
B	2 1	B	D	C
C	4 3	B	-	D
D	3	B	-	-

DFA



From NFA to DFA

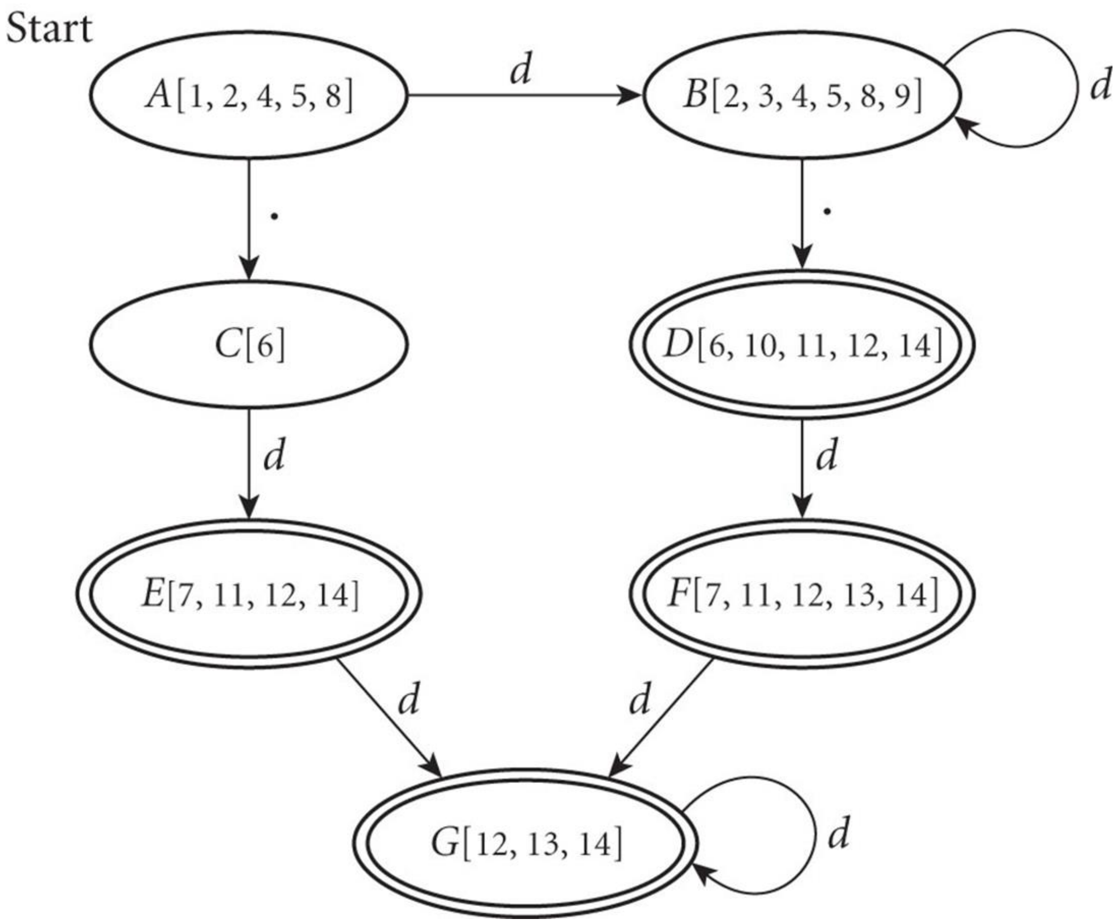


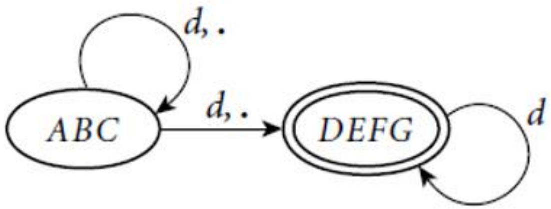
Figure 2.9 A DFA equivalent to the NFA at the bottom of Figure 2.8. Each state of the DFA represents the set of states that the NFA could be in after seeing the same input.

Note: As long as you can find a DFA so that the NFA and the DFA accept the same inputs. Then, the two automaton are equivalent.

Scanning IV

Scanner Design 3: DFA Minimization

SECTION 9



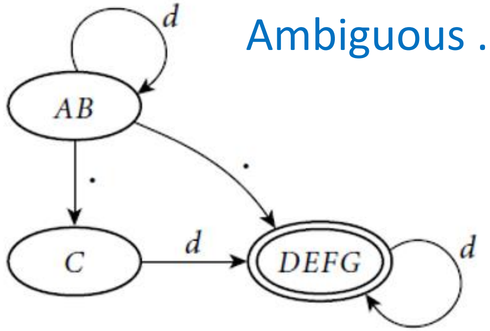
Minimization of DFA I

Starting from a regular expression we have now constructed an equivalent DFA.

Though this DFA has seven states, a smaller one should exist.

In particular, once we have seen both a **d** and a `.` , the only valid transitions are on **d**, and we ought to be able to make do with a single final state. We can formalize this intuition, allowing us to apply it to any DFA, via the following inductive construction.

Initially we place the states of the (not necessarily minimal) DFA into two equivalence classes: final states and nonfinal states. We then repeatedly search for an equivalence class χ and an input symbol c such that when given c as input, the states in χ make transitions to states in $k > 1$ different equivalence classes. We then partition χ into k classes in such a way that all states in a given new class would move to a member of the same old class on c.



Ambiguous .

Minimization of DFA II

When we are unable to find a class to partition in this fashion we are done. In our example, the original placement puts States D, E, F, and G in one class (final states) and States A, B, and C in another, as shown in the upper left of Figure 2.10.

Unfortunately, the start state has ambiguous transitions on both **d** and **.** . To address the *d* ambiguity, we split ABC into AB and C, as shown in the upper right.

New State AB has a self-loop on ***d***; new State C moves to State DEFG. State AB still has an ambiguity on **.**, however, which we resolve by splitting it into States A and B, as shown at the bottom of the figure.

At this point there are no further ambiguities, and we are left with a four-state minimal DFA.

Minimization of DFA

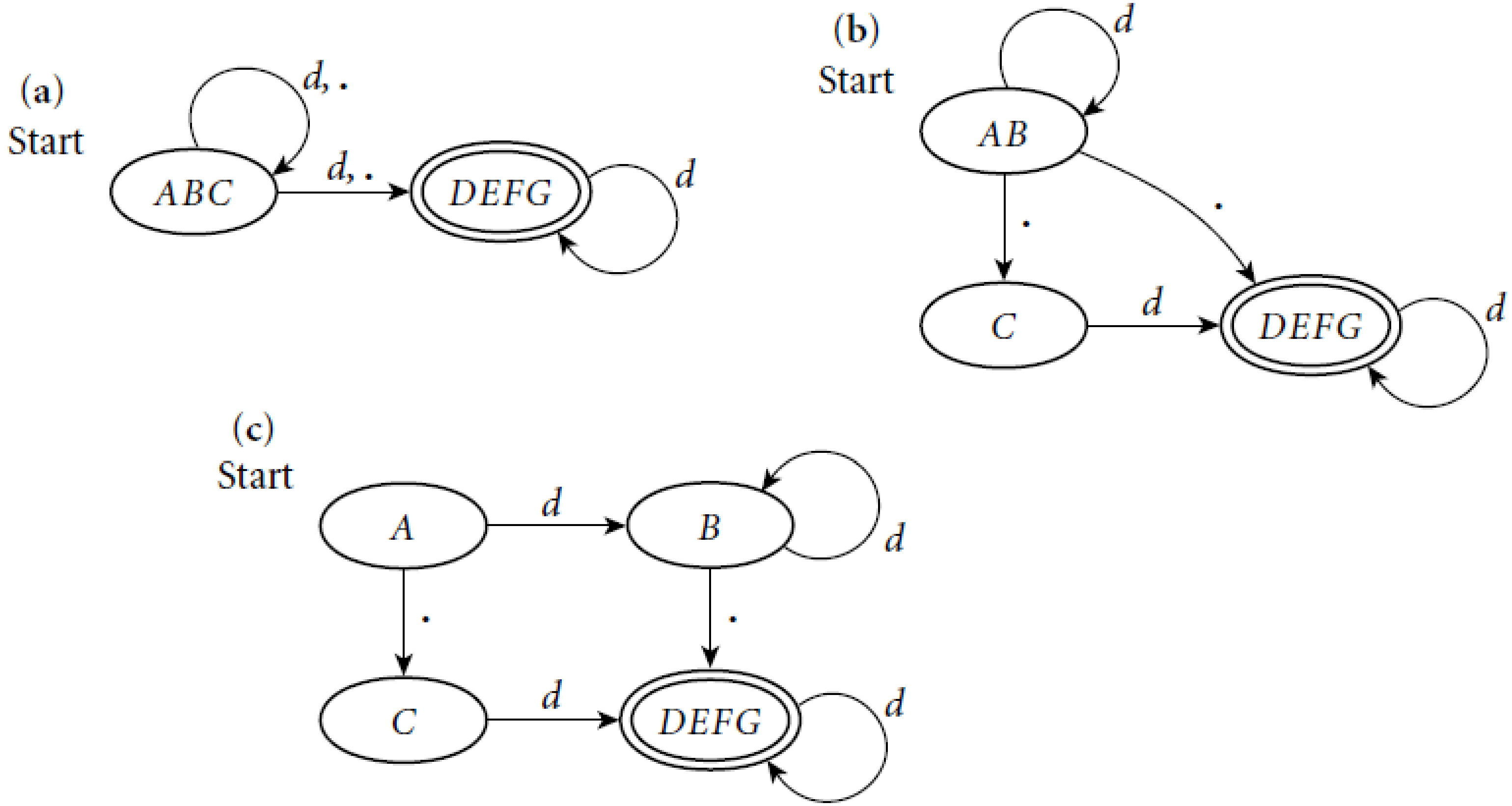


Figure 2.10 Minimization of the DFA of Figure 2.9. In each step we split a set of states to eliminate a transition ambiguity.

Scanning V

Scanner Styles and pragmas

SECTION 10

Scanner Design (Scanner Code)

- We can implement a scanner that explicitly captures the “circles-and arrows” structure of DFA in either of two main ways.
- Implementation 1: one embeds the automaton in the control flow of the program using **`goto`**s or nested **`case(switch)`** statements.
- Implementation 2: Table and Driver.

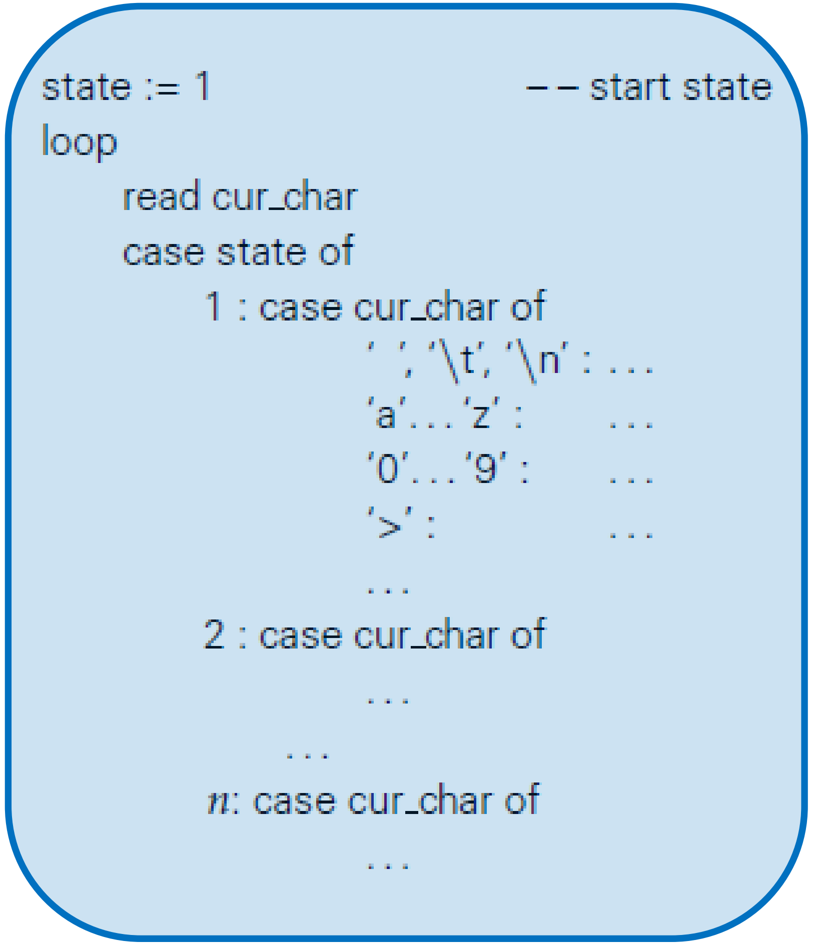
As a general rule handwritten automata tend to use nested case statements, while automatically generated automata use tables.

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use **perl**, **awk**, **sed**
  - for details see Figure 2.11
- Table-driven DFA is what **lex** and **scangen** produce
  - **lex (flex)** in the form of **C** code
  - **scangen** in the form of numeric tables and a separate driver (for details see Figure 2.12)

Nested **case** Statement Automaton

- Scanning through the character stream with look-ahead.
- Use case statement as the dispatcher of the next state for the state machine(automaton).
- The outer case statement covers the states of the finite automaton.
- The inner case statements cover the transitions out of each state. Most inner clauses simply set a new state. Some return from the scanner with the current token.



```
state := 1                              -- start state
loop
    read cur_char
    case state of
        1 : case cur_char of
                    ' ', '\t', '\n' : . . .
                    'a'. . . 'z' :      . . .
                    '0'. . . '9' :      . . .
                    '>' :               . . .
                    . . .
        2 : case cur_char of
                    . . .
                . . .
        n: case cur_char of
                    . . .
```

Scanning and Look-ahead

- identifiers: keyword | variable | method_name | class_name | …
- In general, upcoming characters that a scanner must examine in order to make a decision are known as its loo-ahead. In the next lecture, we will see a similar notion of look-ahead tokens in parsing.
- In messier languages, a scanner may need to look an arbitrary distance ahead.

Scanning

- In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have

```
      DO 5 I = 1,25    loop
      DO 5 I = 1.25    assignment
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
    - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
    - In Pascal, for example, when you have a 3 and you a see a dot
        - do you proceed (in hopes of getting 3.14)?
          or
        - do you stop (in fear of getting 3..5)?

Table-Driven Scanning

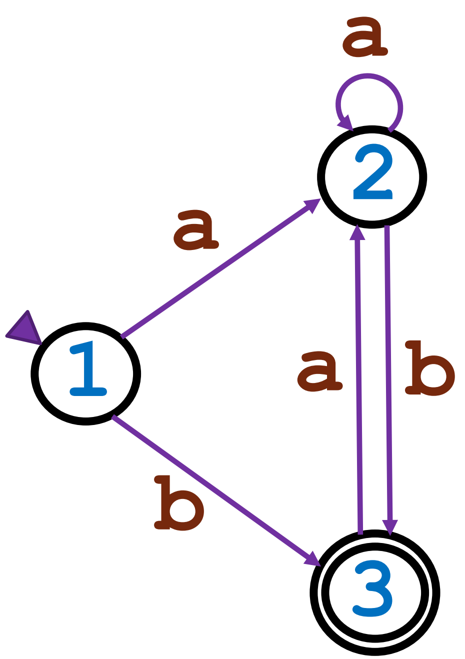
- In the preceding subsection we sketched how control flow – **a loop and nested case statements** – can be used to represent a finite automaton.
- An alternative approach represents the automaton as a data structure: a **two-dimensional transition table**.

Table-Driven Automaton

Table-Driven Scanner is an Application of Table Driven Automaton

DFA

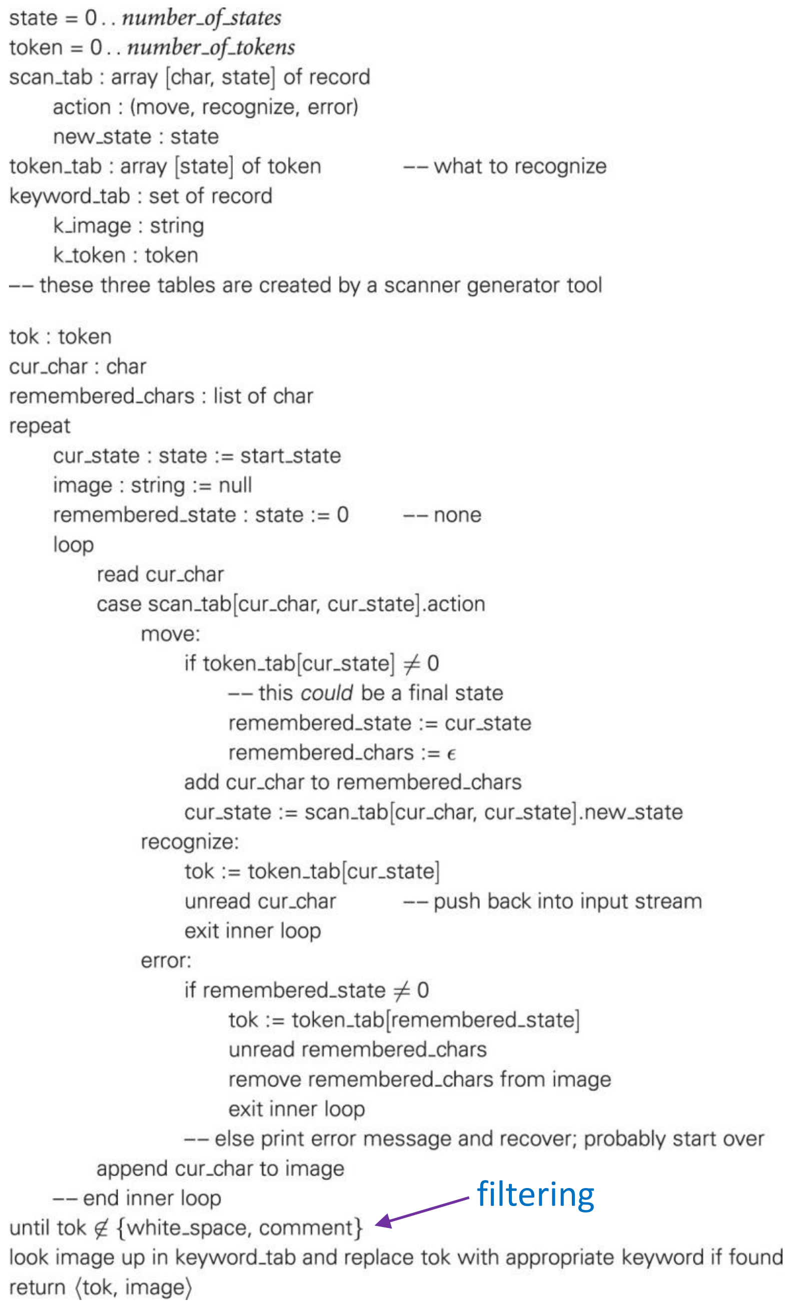
DFA	a	b
1	2	3
2	2	3
3	2	-1



```
#define isFinal(s)  ((s)<0)
int scanner(){
   char ch;
   int currState = 1;
   while(TRUE){
      ch = NextChar();
      if (ch = EOF) return 0;
      currState = T[currState, ch];
       if (isFinal(currState){
             return 1;   /* success */
       }
    }
}
```

A Driver Program for Table-Driven Scanner

- Outer repeat loop is used for state transition management. Outer loop is also used to filter out the comments and white-space characters.
- Inner loop is used for character scanning and next-state update.
- move: perform state transition. (no break on this case.)
- recognize: token generation. (following move)
- error: lexical error handler.



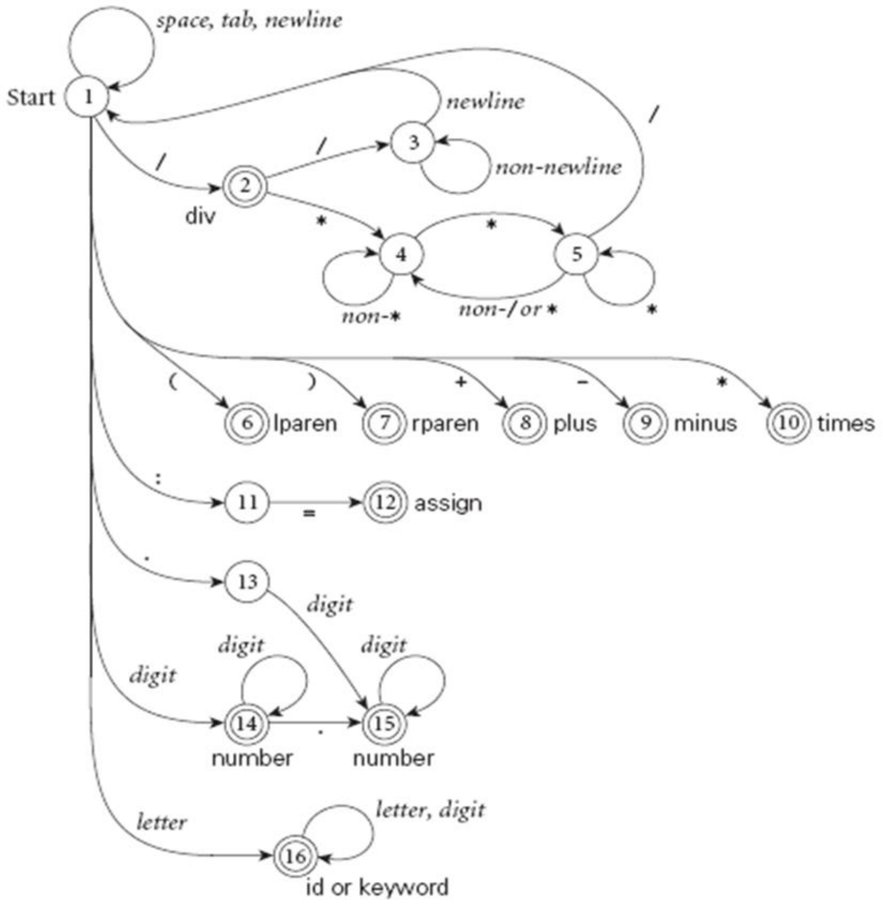
```
state = 0 . . number_of_states
token = 0 . . number_of_tokens
scan_tab : array [char, state] of record
    action : (move, recognize, error)
    new_state : state
token_tab : array [state] of token              –– what to recognize
keyword_tab : set of record
    k_image : string
    k_token : token
–– these three tables are created by a scanner generator tool

tok : token
cur_char : char
remembered_chars : list of char
repeat
    cur_state : state := start_state
    image : string := null
    remembered_state : state := 0               –– none
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                if token_tab[cur_state] ≠ 0
                    –– this could be a final state
                    remembered_state := cur_state
                    remembered_chars := ϵ
                add cur_char to remembered_chars
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := token_tab[cur_state]
                unread cur_char                 –– push back into input stream
                exit inner loop
            error:
                if remembered_state ≠ 0
                    tok := token_tab[remembered_state]
                    unread remembered_chars
                    remove remembered_chars from image
                    exit inner loop
                –– else print error message and recover; probably start over
        append cur_char to image
    –– end inner loop
until tok ∉ {white_space, comment}
look image up in keyword_tab and replace tok with appropriate keyword if found
return ⟨tok, image⟩
```

filtering

Figure 2.11 **Driver for a table-driven scanner,** with code to handle the ambiguous case in which one valid token is a prefix of another, but some intermediate string is not.

Scanner for Calculator Tokens

in the form of a finite automaton (Case Study)



comment ⟶ $/\ast\ (\ non\text{-}\ast \mid \ast\ non\text{-}/\ )^{\ast}\ \ast^{\ast}/$
$\mid //\ (\ non\text{-}newline\ )^{\ast}\ newline$

assign ⟶ :=

plus ⟶ +

minus ⟶ -

times ⟶ *

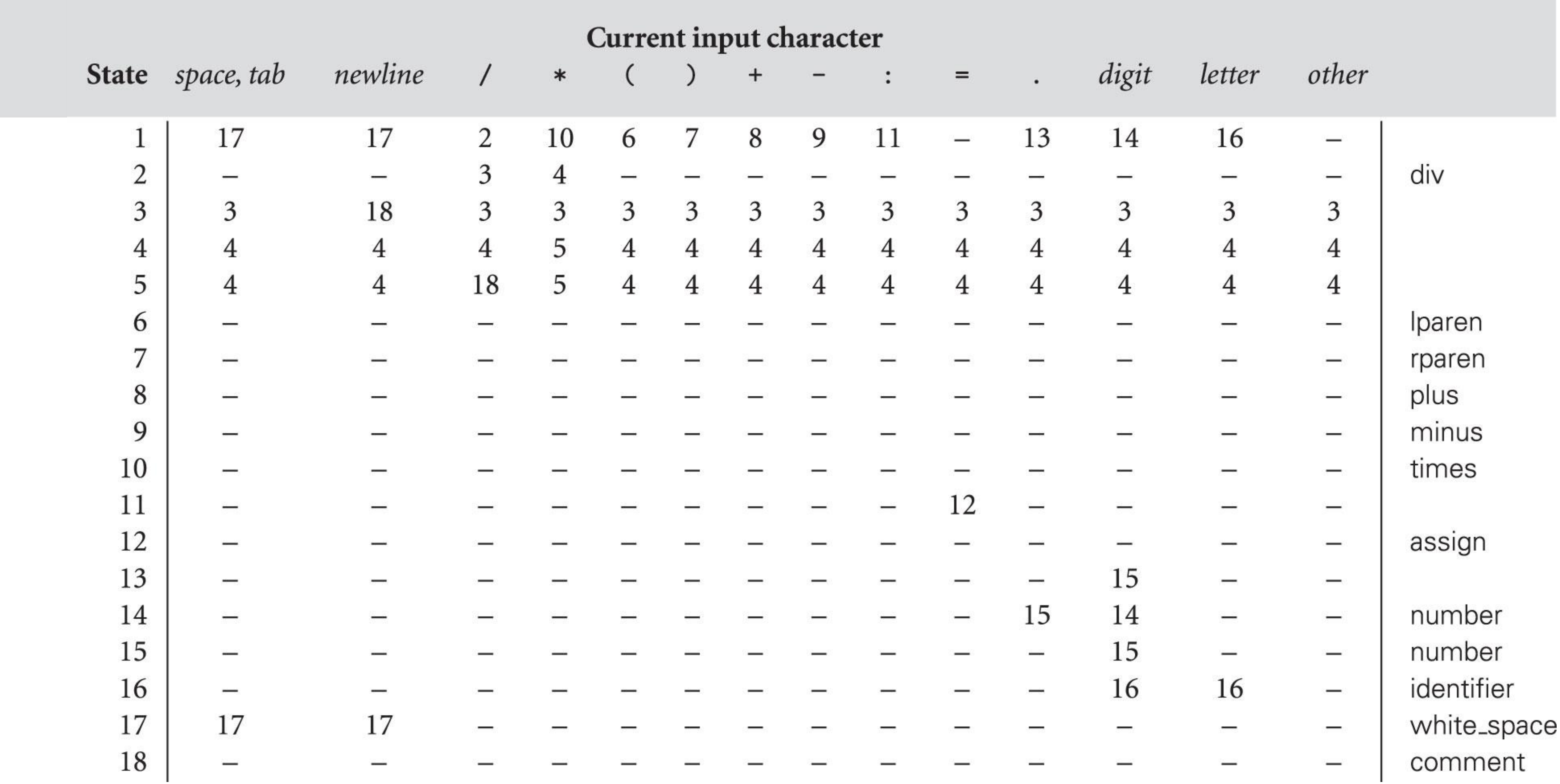
div ⟶ /

lparen ⟶ (

rparen ⟶)

id ⟶ *letter* (*letter* | *digit*)*
except for `read` and `write`

number ⟶ $digit\ digit^{\ast} \mid digit^{\ast}\ (\ .\ digit \mid digit\ .\ )\ digit^{\ast}$



	Current input character														
State	*space, tab*	*newline*	/	*	(	)	+	–	:	=	.	*digit*	*letter*	*other*	
1	17	17	2	10	6	7	8	9	11	–	13	14	16	–	
2	–	–	3	4	–	–	–	–	–	–	–	–	–	–	div
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	lparen
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	rparen
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	plus
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	minus
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	times
11	–	–	–	–	–	–	–	–	–	12	–	–	–	–	
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	assign
13	–	–	–	–	–	–	–	–	–	–	–	15	–	–	
14	–	–	–	–	–	–	–	–	–	–	15	14	–	–	number
15	–	–	–	–	–	–	–	–	–	–	–	15	–	–	number
16	–	–	–	–	–	–	–	–	–	–	–	16	16	–	identifier
17	17	17	–	–	–	–	–	–	–	–	–	–	–	–	white_space
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	comment

Figure 2.12 Scanner tables for the calculator language. These could be used by the code of Figure 2.11. States are numbered as in Figure 2.6, except for the addition of two states—17 and 18—to "recognize" white space and comments. The right-hand column represents table token tab; the rest of the figure is scan tab. Dashes indicate no way to extend the current token. Table keyword tab (not shown) contains the strings read and write.

EBNF:

SECTION 11

EBNF Format in this course

There are a few different typographic conventions used in the **EBNF** lecture, compared to the **EBNF** used in actual Python Documentation. Here is a short summary of the six differences, and a short and large example.

1. Write <= (separating LHS from RHS) as **::=**
2. Italicized *names* (of rules) are just written as **names**
3. Boxed characters (which stand for themselves) are written within **quotes**
4. () do not stand for themselves; they are used for grouping (see rule 5) write "(" and ")" for parentheses in the EBNF used for Python Documentation
5. **{item}** is written as **item***; **{item1 ... itemN}** is written **(item1 ... itemN)***
6. Writing **+** superscript means repeat 1 or more times

EBNF in this course

::= derivation

| option

() optional set ; () sometimes

a* repetition ; Kleene’s star *

a+ ; a a* is equivalent to a+

“51” “(“ ; literals

EBNF Example in Class Note

Short Example:

```
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
integer ::= ["+"|"-"] digit digit*
```

the last rule can be written as either:

```
integer ::= ["+"|"-"] digit (digit)*
```

or

```
integer ::= ["+"|"-"] (digit)+
```

Online:

See https://docs.python.org/3/reference/simple_stmts.html

Questions: is it legal in Python to write a = b = 0

Also, see the import statement (and all its forms)

EBNF

Large Example:

This is from Section 6.1.3.1 **Format Specification Mini-Language** of the Python Library. Format strings contain “replacement fields” surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.

Grammar for Replacement Field

The grammar for a replacement field is as follows:

```
replacement_field ::=  "{" [field_name] ["!" conversion]
                                   [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name |
                                       "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]“ > +
conversion        ::=  "r" | "s" | "a"
```

Grammar for Replacement Field

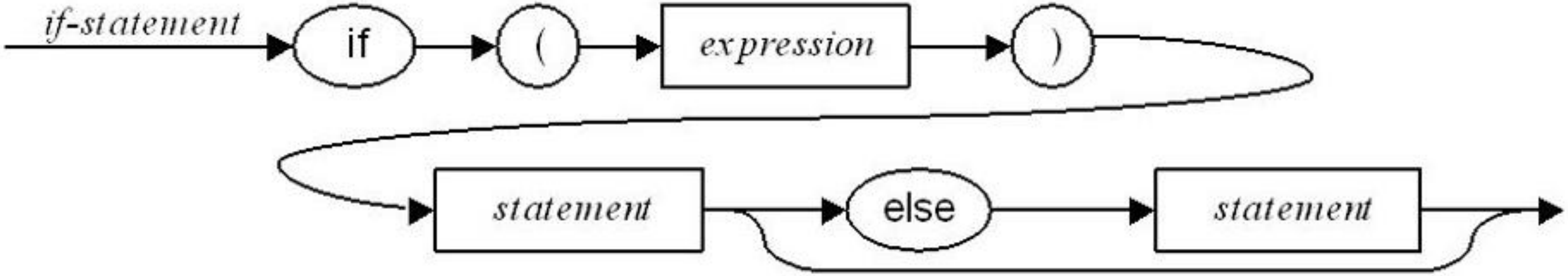
```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill        ::=  <a character other than '{' or '}'>
align       ::=  "<" | ">" | "=" | "^"
sign        ::=  "+" | "-" | " "
width       ::=  integer
precision   ::=  integer
type        ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
                 "n" | "o" | "s" | "x" | "X" | "%"
```

Syntax Diagram

SECTION 12

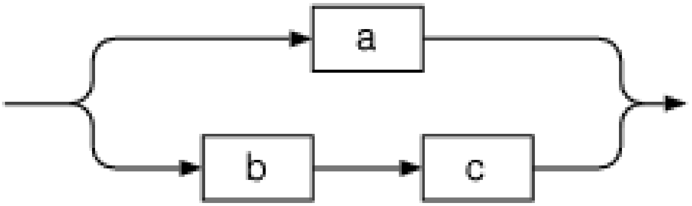
Syntax Diagrams

- An alternative to EBNF.
- Rarely seen any more: EBNF is much more compact.
- Example (if-statement, p. 101):

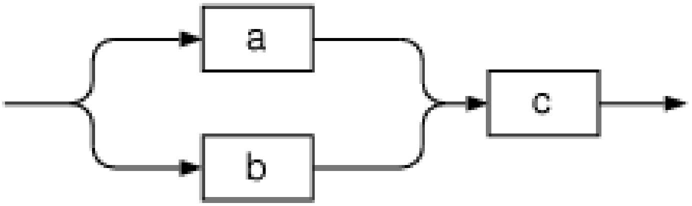


Syntax Diagram

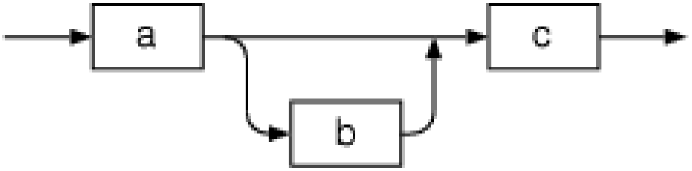
a | b c



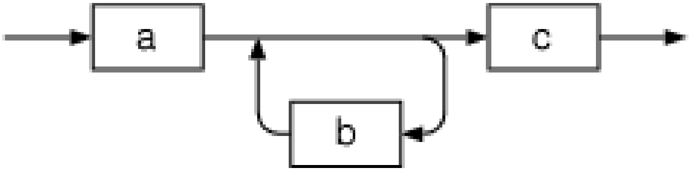
(a | b) c

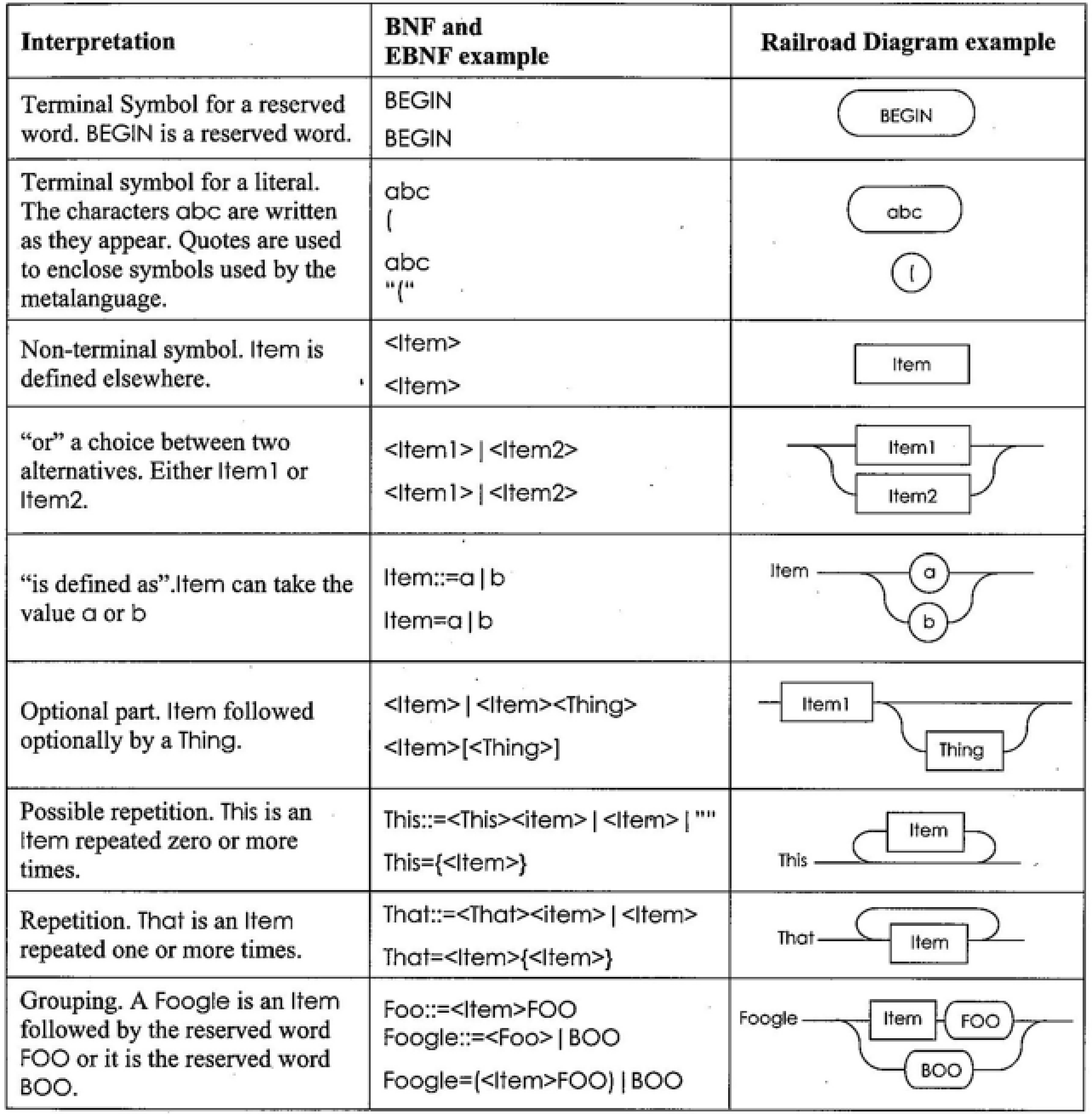


a [b] c

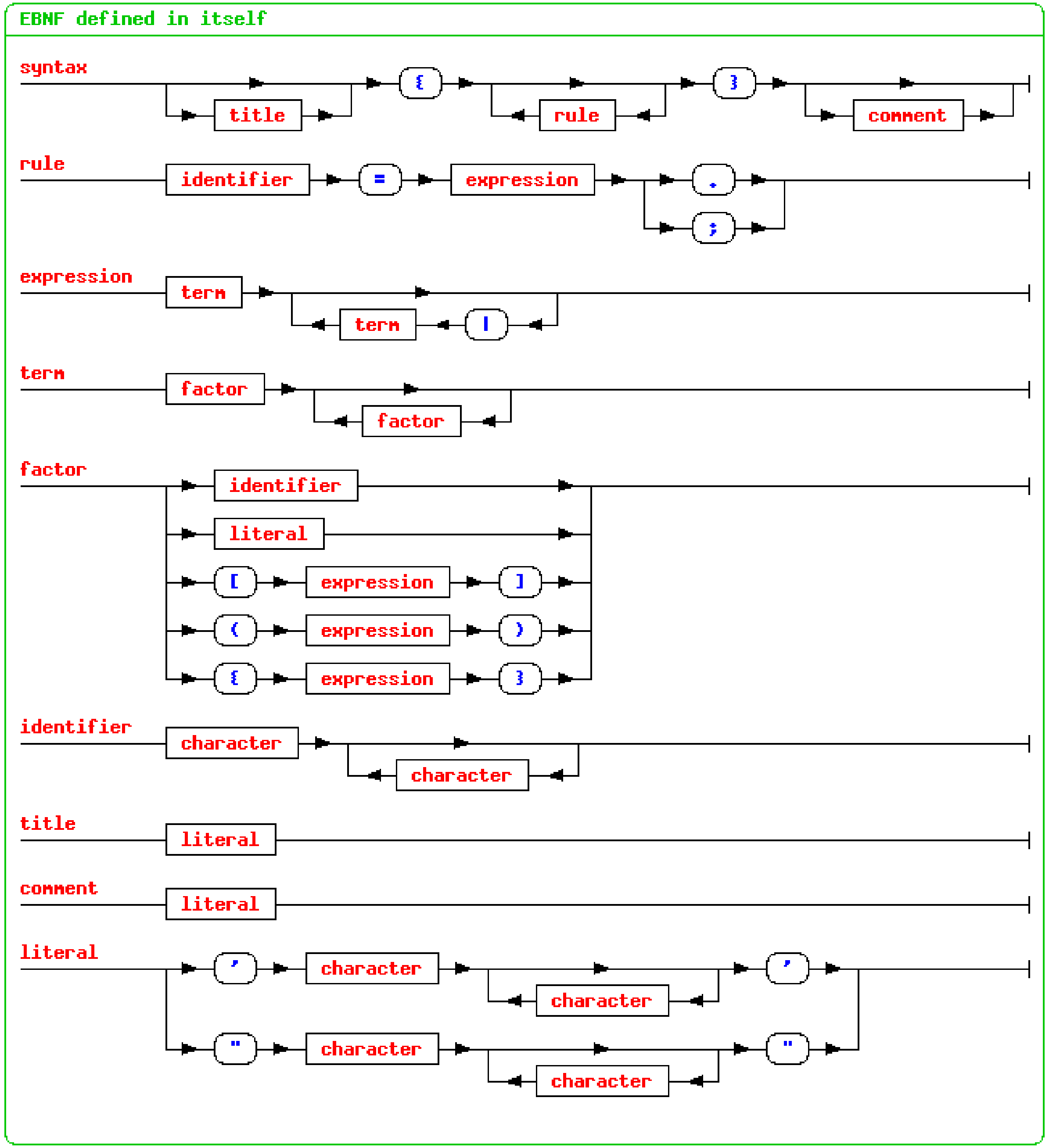


a {b} c

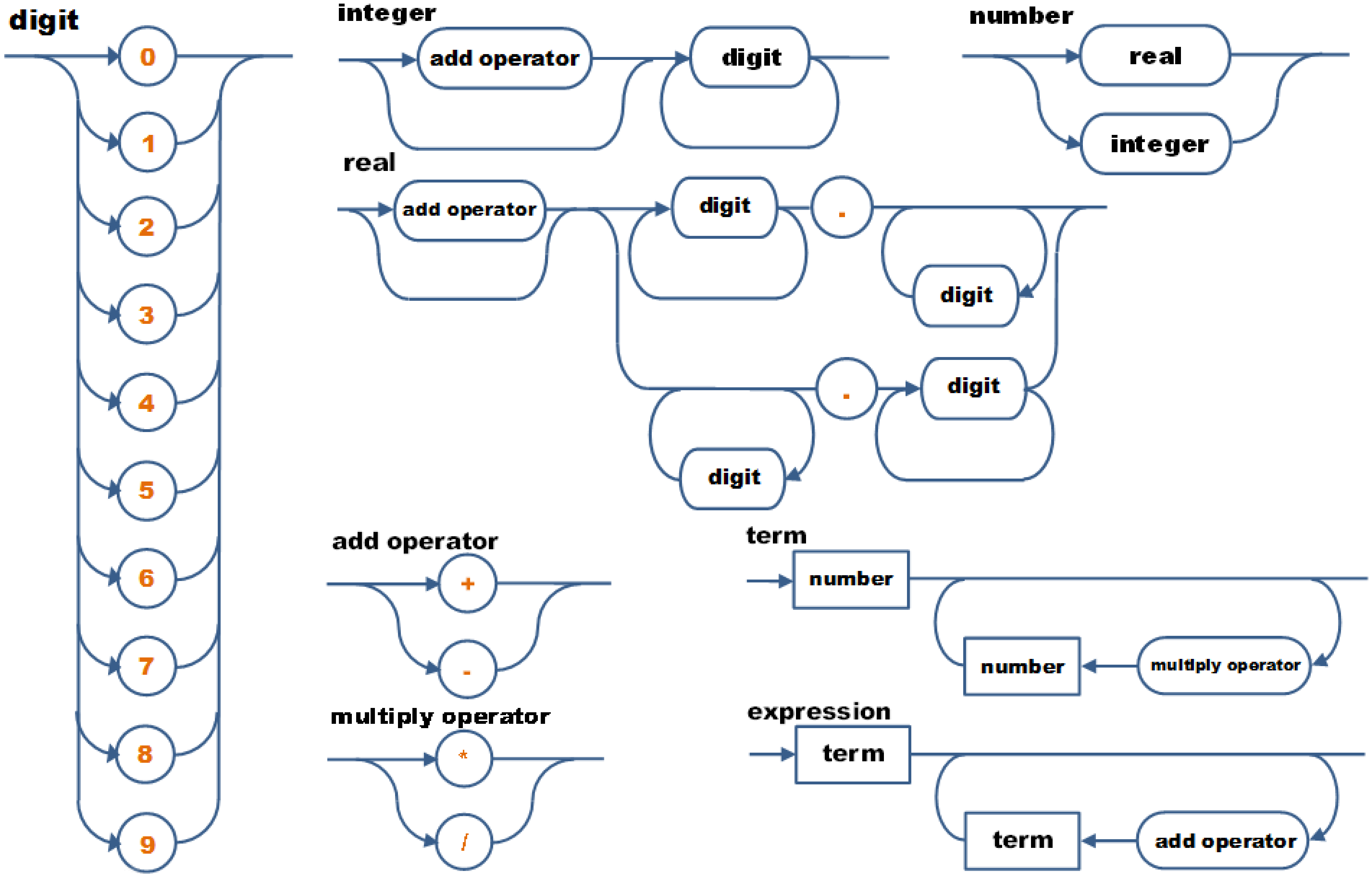




Interpretation	BNF and EBNF example	Railroad Diagram example
Terminal Symbol for a reserved word. BEGIN is a reserved word.	BEGIN BEGIN	BEGIN
Terminal symbol for a literal. The characters abc are written as they appear. Quotes are used to enclose symbols used by the metalanguage.	abc (abc “(“	abc (
Non-terminal symbol. Item is defined elsewhere.	<Item> <Item>	Item
“or” a choice between two alternatives. Either Item1 or Item2.	<Item1> | <Item2> <Item1> | <Item2>	Item1 Item2
“is defined as”.Item can take the value a or b	Item::=a | b Item=a | b	Item a b
Optional part. Item followed optionally by a Thing.	<Item> | <Item><Thing> <Item>[<Thing>]	Item1 Thing
Possible repetition. This is an Item repeated zero or more times.	This::=<This><item> | <Item> | "" This={<Item>}	Item This
Repetition. That is an Item repeated one or more times.	That::=<That><item> | <Item> That=<Item>{<Item>}	That Item
Grouping. A Foogle is an Item followed by the reserved word FOO or it is the reserved word BOO.	Foo::=<Item>FOO Foogle::=<Foo> | BOO Foogle=(<Item>FOO) | BOO	Foogle Item FOO BOO



EBNF defined in itself


digit
0
1
2
3
4
5
6
7
8
9
integer
add operator
digit
number
real
integer
real
add operator
digit
.
digit
.
digit
digit
add operator
+
-
multiply operator
*
/
term
number
number
multiply operator
expression
term
term
add operator

Regular Expressions

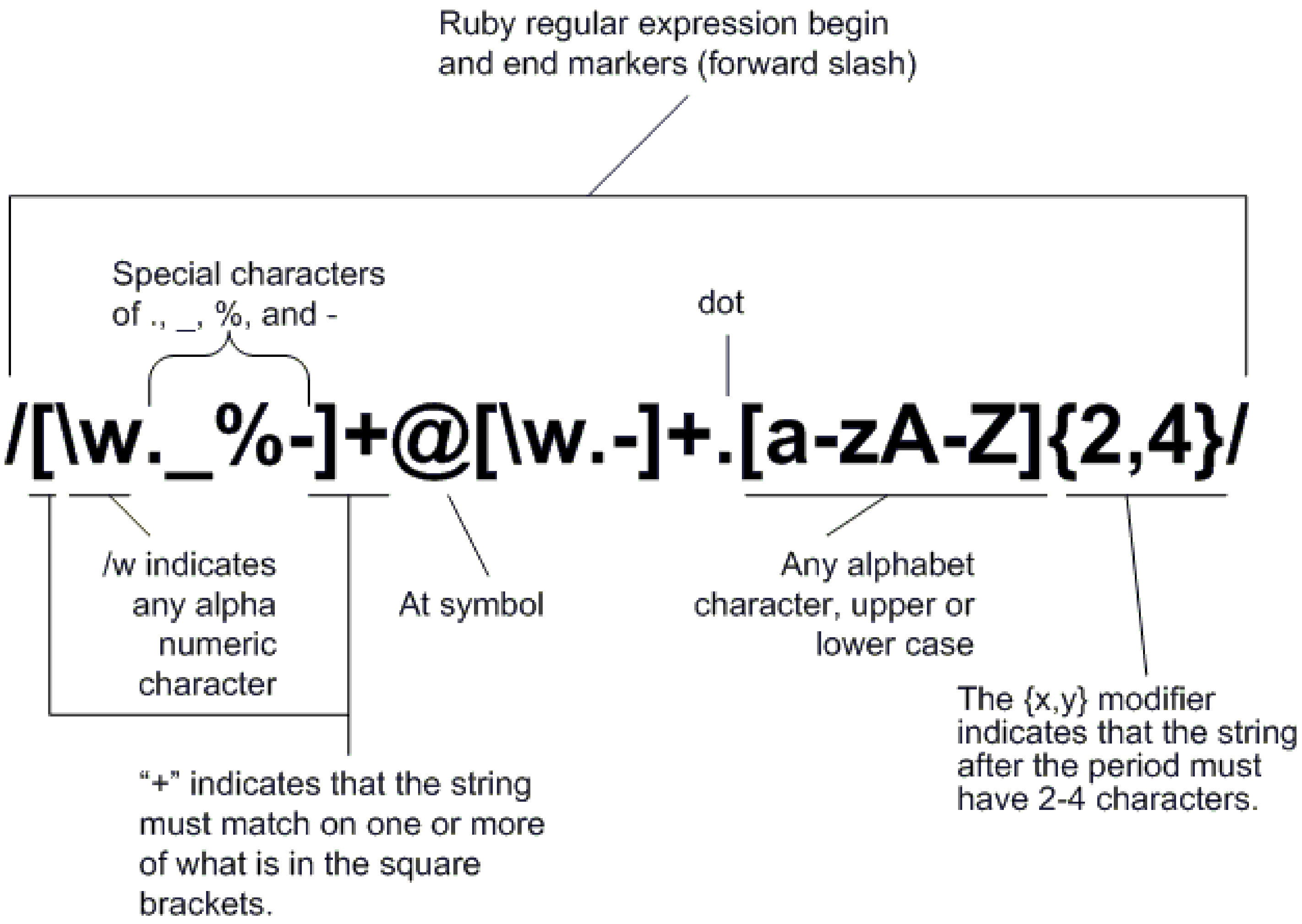
SECTION 13

What is a regular expression?

```
/[a-zA-Z_\-]+@(([a-zA-Z_\-])+\.)+[a-zA-Z]{2,4}/
```

- **regular expression** ("regex"): describes a pattern of text
  - can test whether a string matches the expr's pattern
  - can use a regex to search/replace characters in a string
  - very powerful, but tough to read

- regular expressions occur in many places:
  - text editors (TextPad) allow regexes in search/replace
  - languages: JavaScript; Java `Scanner`, `String split`
  - Unix/Linux/Mac shell commands (`grep`, `sed`, `find`, etc.)



Ruby regular expression begin and end markers (forward slash)

Special characters of ., _, %, and -

dot

/[\w._%-]+@[\w.-]+.[a-zA-Z]{2,4}/

/w indicates any alpha numeric character

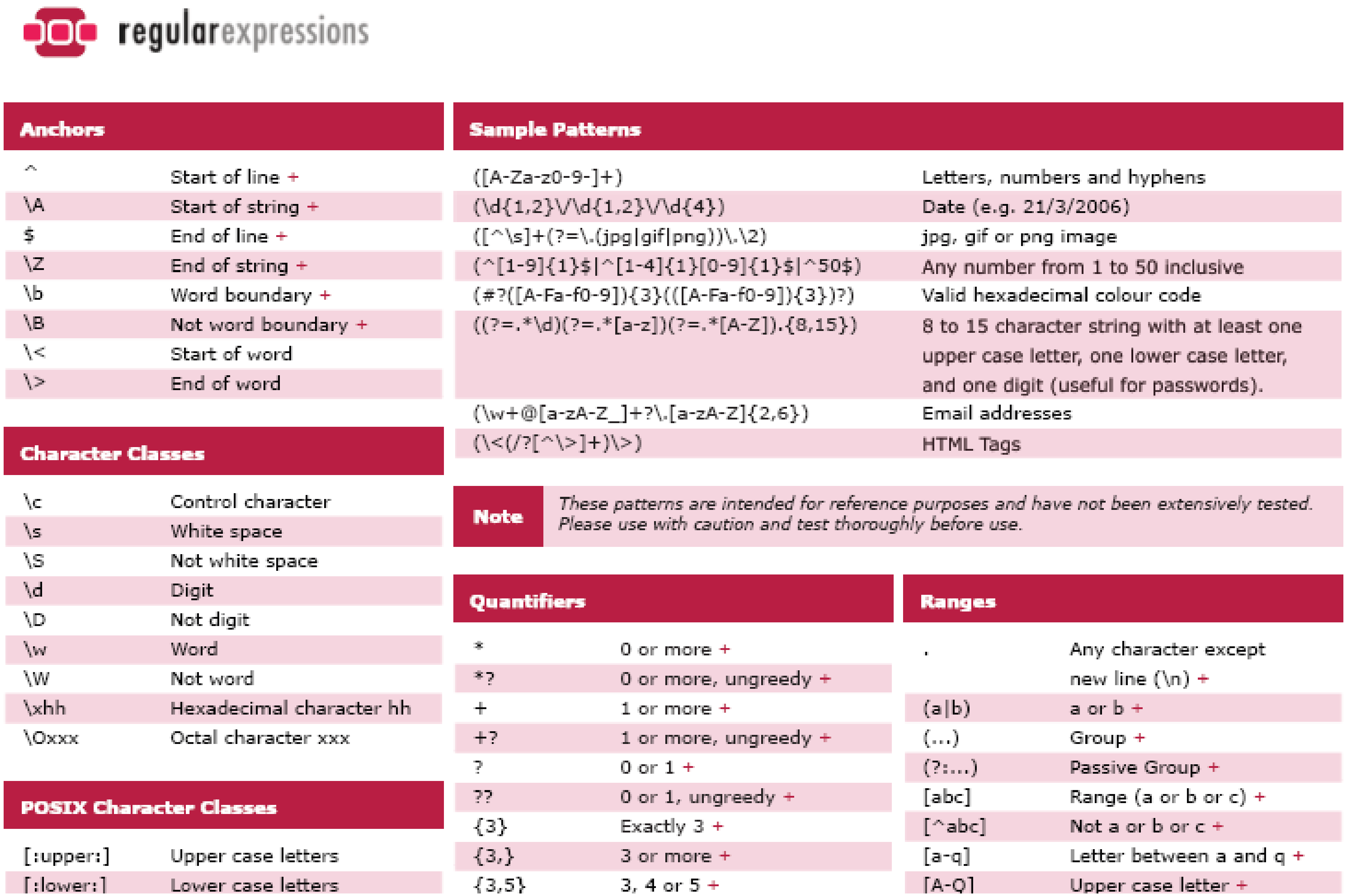
At symbol

Any alphabet character, upper or lower case

The {x,y} modifier indicates that the string after the period must have 2-4 characters.

“+” indicates that the string must match on one or more of what is in the square brackets.

username @ domain . qualifier (com/net/tv/…)



regularexpressions

Anchors

^	Start of line +
\A	Start of string +
$	End of line +
\Z	End of string +
\b	Word boundary +
\B	Not word boundary +
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\xhh	Hexadecimal character hh
\Oxxx	Octal character xxx

POSIX Character Classes

[:upper:]	Upper case letters
[:lower:]	Lower case letters

Sample Patterns

([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note *These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.*

Quantifiers

*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +

Ranges

.	Any character except new line (\n) +
(a|b)	a or b +
(...)	Group +
(?:...)	Passive Group +
[abc]	Range (a or b or c) +
[^abc]	Not a or b or c +
[a-q]	Letter between a and q +
[A-Q]	Upper case letter +

Syntax of Regular Expressions

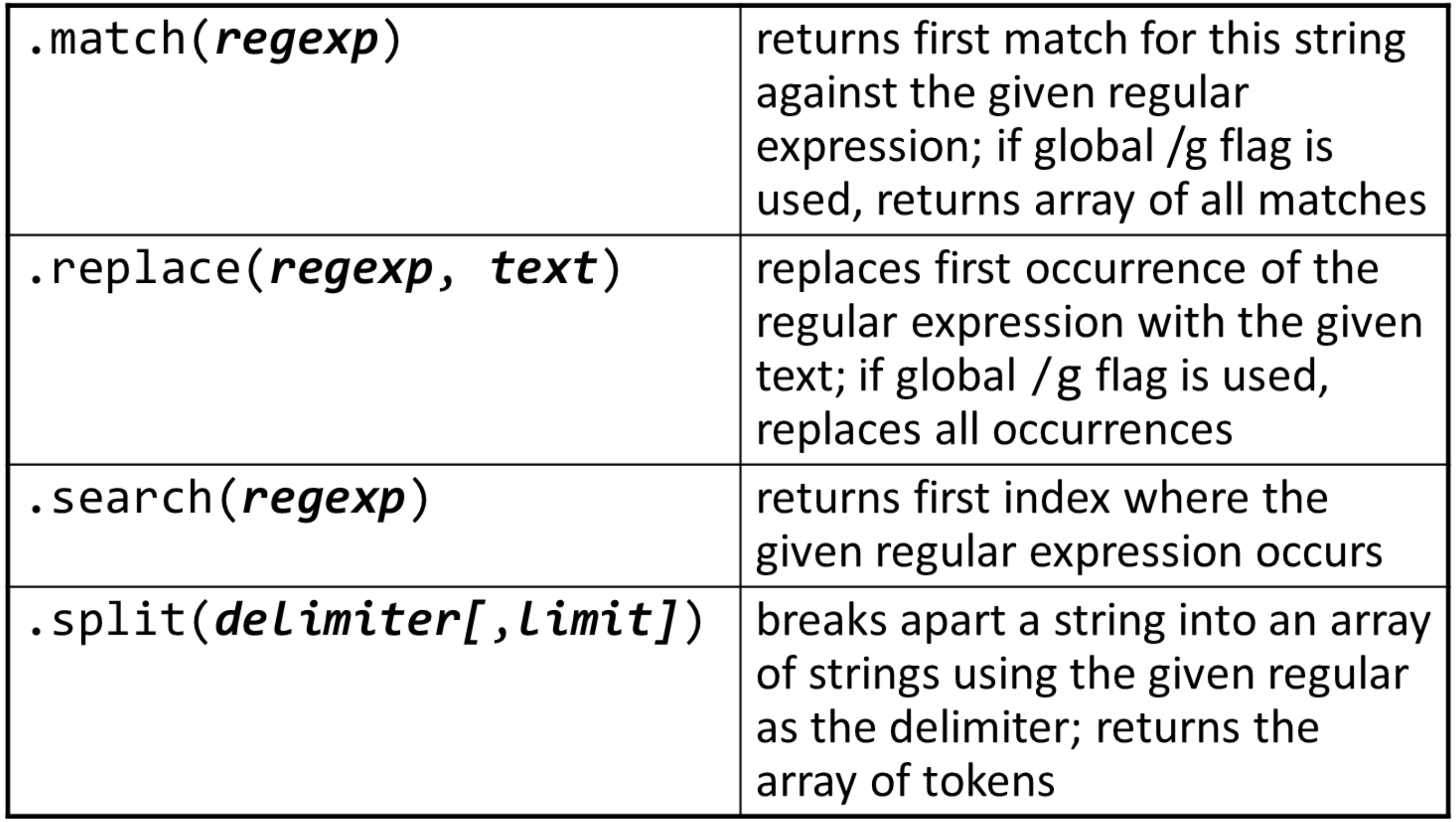
SECTION 14

Influences on Tools/Languages

- **Java**: basic syntax, many type/method names
- **Scheme**: first-class functions, closures, dynamism
- **Self**: prototypal inheritance
- **Perl**: **regular expressions**
- Historic note: *Perl* was a horribly flawed and very useful scripting language, based on Unix shell scripting and C, that helped lead to many other better languages.
  - PHP, Python, Ruby, Lua, ...
  - Perl was excellent for string/file/text processing because it built *regular expressions* directly into the language as a first-class data type. JavaScript wisely stole this idea.

String regexes methods

Javascript String Methods (Used for Text Processing)



`.match(`***regexp***`)`	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
`.replace(`***regexp***`, `***text***`)`	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
`.search(`***regexp***`)`	returns first index where the given regular expression occurs
`.split(`***delimiter[,limit]***`)`	breaks apart a string into an array of strings using the given regular as the delimiter; returns the array of tokens

Basic regexes

/abc/

- a regular expression literal in JS is written `/`***`pattern`***`/`
- the simplest regexes simply match a given substring

- the above regex matches any line containing "abc"
  - *YES* : `"abc"`, `"abcdef"`, `"defabc"`, `".=.abc.=."`
  - *NO* : `"fedcba"`, `"ab c"`, `"AbC"`, `"Bash"`, ...

Wildcards and anchors

`.` (a dot) matches any character except `\n`

- `/.oo.y/` matches `"Doocy"`, `"goofy"`, `"LooPy"`, ...
- use `\.` to literally match a dot `.` character

`^` matches the beginning of a line; `$` the end

- “`^if$`” matches lines that consist entirely of `if`

`\<` demands that pattern is the beginning of a *word*;
`\>` demands that pattern is the end of a word

- “`\<for\>`” matches lines that contain the word `"for"`

String match

***string*.match(*regex*)**

- if string fits pattern, returns matching text; else `null`
  - can be used as a Boolean truthy/falsey test:

    ```
    if (name.match(/[a-z]+/)) { ... }
    ```

- **g** after regex for array of *global* matches
  - `"obama".match(/.a/g)` returns `["ba", "ma"]`

- **i** after regex for case-*insensitive* match
  - `name.match(/Marty/i)` matches `"marty"`, `"MaRtY"`

String replace

***string*.replace(*regex*, "*text*")**

- replaces *first occurrence* of pattern with the given text
  - `var state = "Mississippi";`
    `state.replace(/s/, "x")` returns `"Mixsissippi"`
- **g** after regex to replace *all occurrences*
  - `state.replace(/s/g, "x")` returns `"Mixxixxippi"`
- *returns* the modified string as its result; must be stored
  - `state = state.replace(/s/g, "x");`

Special characters

| means OR

- `/abc|def|g/` matches lines with "`abc`", "`def`", or "`g`"
- precedence: `^Subject|Date:` vs. `^(Subject|Date):`
- There's no AND & symbol. Why not?

( ) are for grouping

- `/(Homer|Marge) Simpson/` matches lines containing "`Homer Simpson`" or "`Marge Simpson`"

\ starts an escape sequence

- many characters must be escaped: `/ \ $ . [ ] ( ) ^ * + ?`
- "`\.\\n`" matches lines containing "`.\n`"

Quantifiers: * + ?

*** means 0 or more occurrences**

- /abc*/ matches "ab", "abc", "abcc", "abccc", ...
- /a(bc)*/" matches "a", "abc", "abcbc", "abcbcbc", ...
- /a.*a/ matches "aa", "aba", "a8qa", "a!?_a", ...

+ means 1 or more occurrences

- /a(bc)+/ matches "abc", "abcbc", "abcbcbc", ...
- /Goo+gle/ matches "Google", "Gooogle", "Goooogle", ...

? means 0 or 1 occurrences

- /Martina?/ matches lines with "Martin" or "Martina"
- /Dan(iel)?/ matches lines with "Dan" or "Daniel"

More quantifiers

`{`***`min`***`,`***`max`***`}` means between ***min*** and ***max*** occurrences

- `/a(bc){2,4}/` matches lines that contain "abcbc", "abcbcbc", or "abcbcbcbc"

- ***min*** or ***max*** may be omitted to specify any number
  - `{2,}` 2 or more
  - `{,6}` up to 6
  - `{3}` exactly 3

Character sets

`[ ]` group characters into a *character set*;
will match any single character from the set

- `/[bcd]art/` matches lines with "bart", "cart", and "dart"
- equivalent to `/(b|c|d)art/` but shorter

- inside `[]`, most modifier keys act as normal characters
  - `/what[.!*?]*/` matches "what", "what.", "what!", "what?**!", ...
    - *Exercise* : Match letter grades e.g. A+, B-, D.

Character ranges

- inside a character set, specify a range of chars with `-`
  - `/[a-z]/` matches any lowercase letter
  - `/[a-zA-Z0-9]/` matches any letter or digit
- an initial `^` inside a character set negates it
  - `/[^abcd]/` matches any character but a, b, c, or d
- inside a character set, `-` must be escaped to be matched
  - `/[\-+]?[0-9]+/` matches optional - or +, followed by at least one digit
    - *Exercise* : Match phone numbers, e.g. 206-685-2181

Built-in character ranges

- `\b` word boundary (e.g. spaces between words)
- `\B` non-word boundary
- `\d` any digit; equivalent to `[0-9]`
- `\D` any non-digit; equivalent to `[^0-9]`
- `\s` any whitespace character; `[ \f\n\r\t\v...]`
- `\S` any non-whitespace character
- `\w` any word character; `[A-Za-z0-9_]`
- `\W` any non-word character
- `\xhh`, `\uhhhh` the given hex/Unicode character
    - `/\w+\s+\w+/` matches two space-separated words

Regex flags

`/pattern/g`	global; match/replace all occurrences
`/pattern/i`	case-insensitive
`/pattern/m`	multi-line mode
`/pattern/y`	"sticky" search, starts from a given index

- flags can be combined:

  `/abc/gi` matches *all* occurrences of abc, AbC, aBc, ABC, ...

Back-references

- text "captured" in `()` is given an internal number; use `\`***`number`*** to refer to it elsewhere in the pattern
  - `\0` is the overall pattern,
  - `\1` is the first parenthetical capture, `\2` the second, ...
  - Example: "A" surrounded by same character: `/(.)A`**`\1`**`/`

  - variations
    - `(?:`***`text`***`)` match ***text*** but don't capture
    - ***`a`***`(?=`***`b`***`)` capture pattern ***b*** but only if preceded by ***a***
    - ***`a`***`(?!`***`b`***`)` capture pattern ***b*** but only if not preceded by ***a***

Replacing with back-references

- you can use back-references when replacing text:
  - refer to captures as ***$number*** in the replacement string
  - Example: to swap a last name with a first name:

```
var name = "Durden,    Tyler";
name = name.replace(/(\w+),\s+(\w+)/, "$2 $1");
// "Tyler Durden"
```

– *Exercise* : Reformat phone numbers from 206-685-2181 format to (206) 685.2181 format.

The RegExp object

```
new RegExp(string)
new RegExp(string, flags)
```

- constructs a regex dynamically based on a given string

`var r = /ab+c/gi;` is equivalent to

`var r = new RegExp("ab+c", "gi");`

  - useful when you don't know regex's pattern until runtime
    - Example: Prompt user for his/her name, then search for it.
    - Example: The empty regex (think about it).

Working with RegExp

- in a regex literal, forward slashes must be \ escaped:

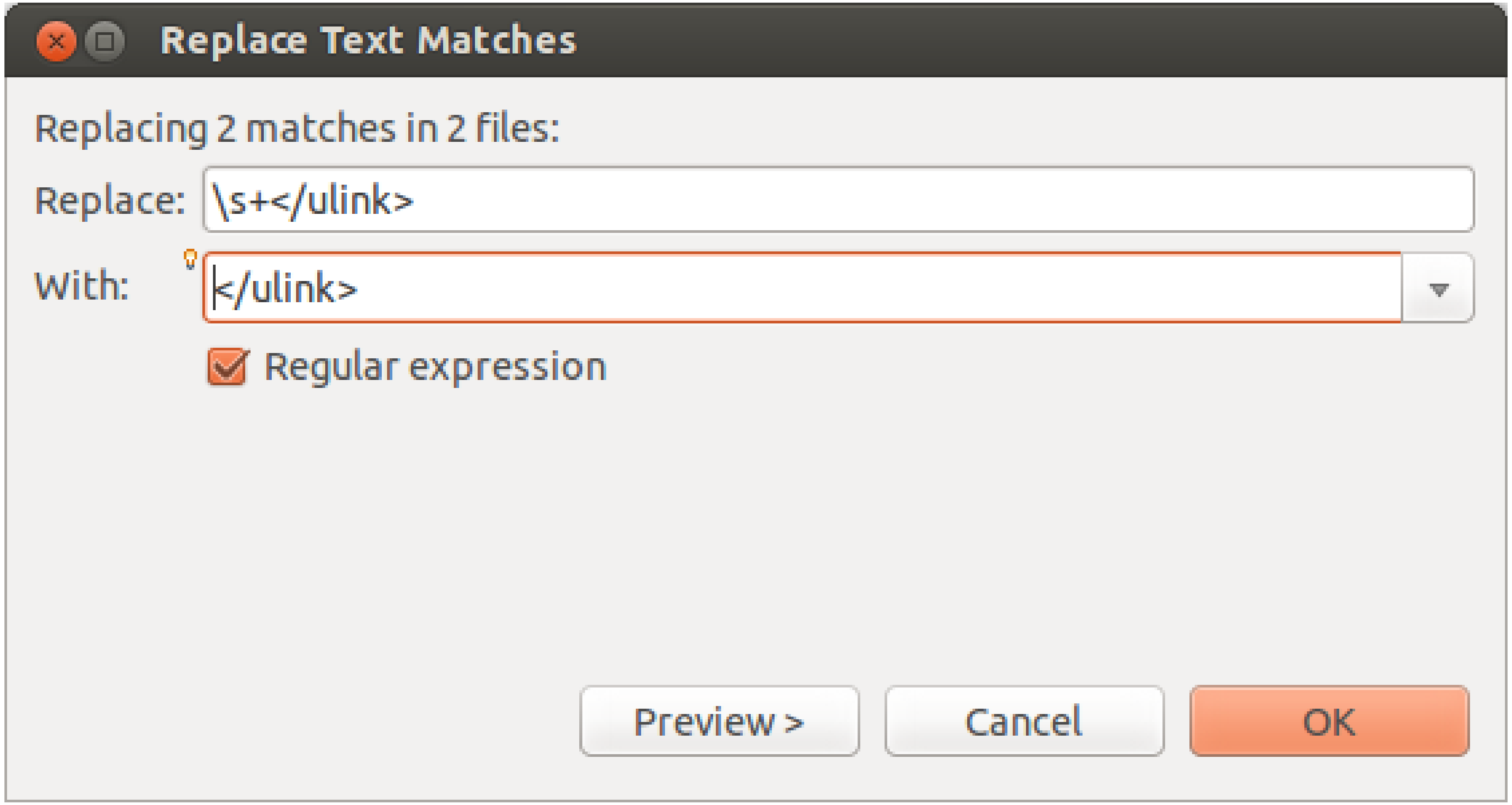
  `/http[s]?:\/\/\w+\.com/`

- in a `new RegExp` object, the pattern is a string, so the usual escapes are necessary (quotes, backslashes, etc.):

  `new RegExp("http[s]?://\\w+\\.com")`

- a `RegExp` object has various properties/methods:
  - properties: `global`, `ignoreCase`, `lastIndex`, `multiline`, `source`, `sticky`; methods: `exec`, `test`

Regular Expressions in Text Editor



Replace Text Matches

Replacing 2 matches in 2 files:

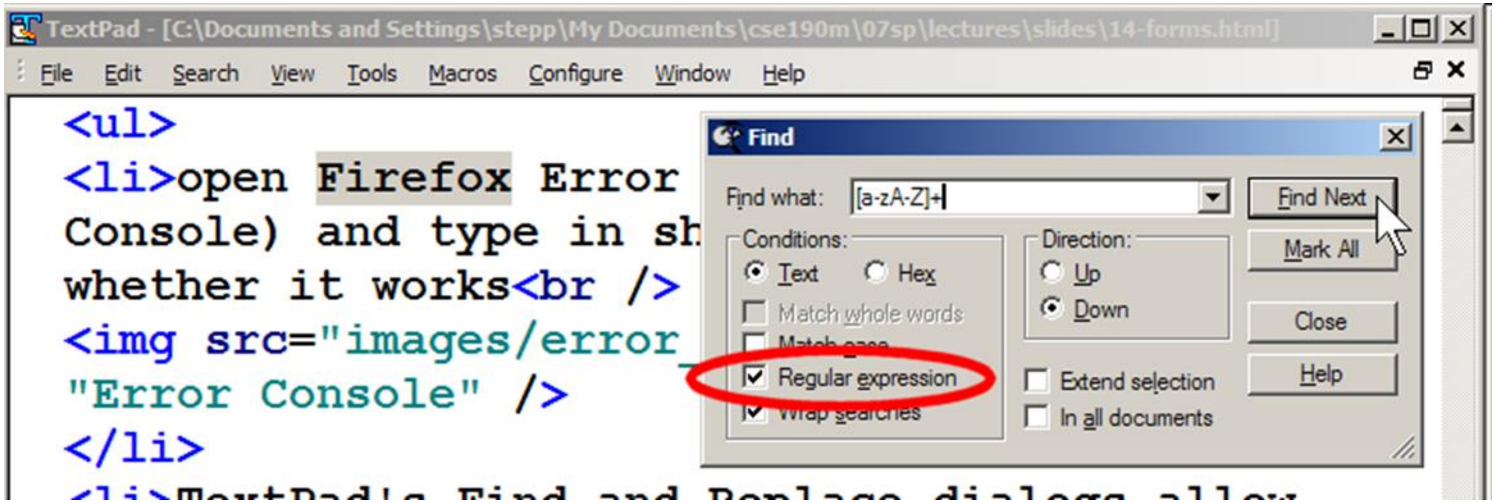
Replace: \s+</ulink>

With: </ulink>

☑ Regular expression

Regexes in editors and tools

- Many editors allow regexes in their Find/Replace feature


TextPad - [C:\Documents and Settings\stepp\My Documents\cse190m\07sp\lectures\slides\14-forms.html]
File Edit Search View Tools Macros Configure Window Help
Find
Find Next
Mark All
Close
Help

```
<ul>
<li>open Firefox Error
Console) and type in sh
whether it works<br />
<img src="images/error
"Error Console" />
</li>
<li>TextPad's Find and Replace dialogs allow
```

Find what: [a-zA-Z]+

Conditions: ● Text ○ Hex

□ Match whole words

□ Match case

☑ Regular expression

☑ Wrap searches

Direction: ○ Up ● Down

□ Extend selection

□ In all documents

- many command-line Linux/Mac tools support regexes

```
grep -e "[pP]hone.*206[0-9]{7}" contacts.txt
```

Summary

SECTION 15

Summary

- Designing a scanner needs to start from the lexical analysis.
- In this lecture, we study how to design the NDFA and DFA
- Then, use FDA to design ad hoc scanner and table-driven scanners.
- In the next lecture, we will be focused on the scanner design based on the regular grammar and regular expression and using automatic scanner generator.

End of Chapter 8A