



# CS49K Programming Languages

Chapter 2: Programming Language  
Syntax - sec. 2.1

LECTURE 2: TOKENIZATION

DR. ERIC CHOU

IEEE SENIOR MEMBER

# Objectives

---

- Overview of Syntax Analysis
- What is tokenization?
- Regular Expression – Backus-Naur Form
- Context Free Grammar
- Tokenization and Flex
- Syntax Analysis and Bison
- Integration of all

# Overview of Programming Language Syntax

SECTION 1

# Formal Languages

## A Computer Term

---

### formal language

noun

1. a language designed for use in situations in which natural language is unsuitable, as for example in mathematics, logic, or computer programming. The symbols and formulas of such languages stand in precisely specified syntactic and semantic relations to one another
2. (**logic**) a logistic system for which an interpretation is provided: distinguished from formal calculus in that the semantics enable it to be regarded as *about* some subject matter

**A Set of Rules**

**A Logistic System**

**Rule (Grammar) – Input (Language) – Machine (States)**

## Grammar

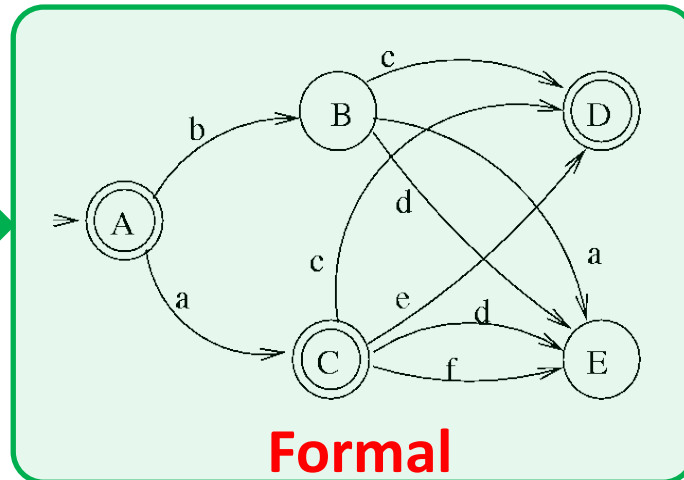
$$\begin{aligned} S &\rightarrow aS \mid bX \\ X &\rightarrow aX \mid bY \\ Y &\rightarrow aY \mid bZ \mid \Lambda \\ Z &\rightarrow aZ \mid \Lambda \end{aligned}$$

**Formal**

## Language Input

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

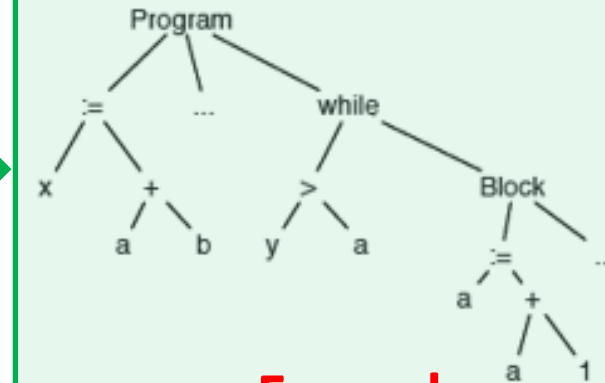
**Informal**



**Formal**

## State Machine

## Validated Tokens Syntax Tree



**Formal**

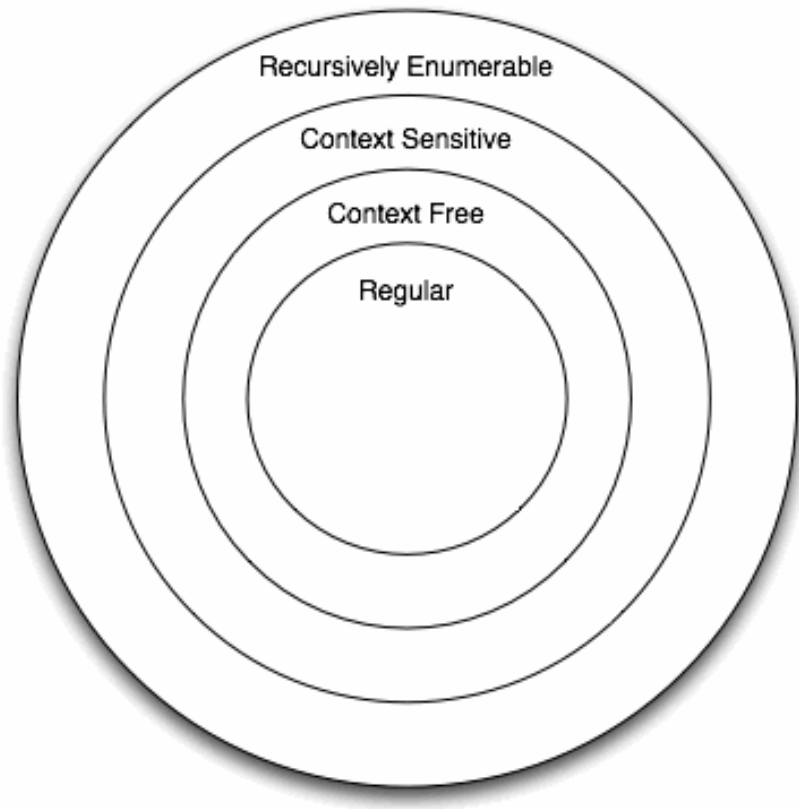
## Target Code

Machine Code  
Assembly  
Byte Code  
Object Code

**Informal**

# Hierarchy of Grammars

---



**Chomsky** defined four types of grammars in the hierarchy, of which, the regular grammars were classified as the most restricted. The other types are context free grammars, context sensitive grammars, and recursively enumerable grammars:

# Hierarchy of Grammars

---

- A **context free grammar** is a grammar where the left hand side of every production rule is a **non-terminal**, and the right hand side is any combination of terminals and non-terminals or the empty string.
- **This is a wider class of grammars than regular grammars**, but it is important to note (based on the hierarchy) that every regular grammar is context free. Chomsky described context free grammars as phrase-structure grammars. While this formalism was originally developed for studying linguistics and natural language, it has had a far greater impact on the development of programming languages through the field of lexical analysis

# Syntax of Arabic numerals

*digit*  $\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

*non\_zero\_digit*  $\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

*natural\_number*  $\rightarrow \text{non\_zero\_digit digit}^*$

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  $\Sigma$  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:  $L_1^* = \{\lambda, 10, 1010, 101010, \dots\}$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}

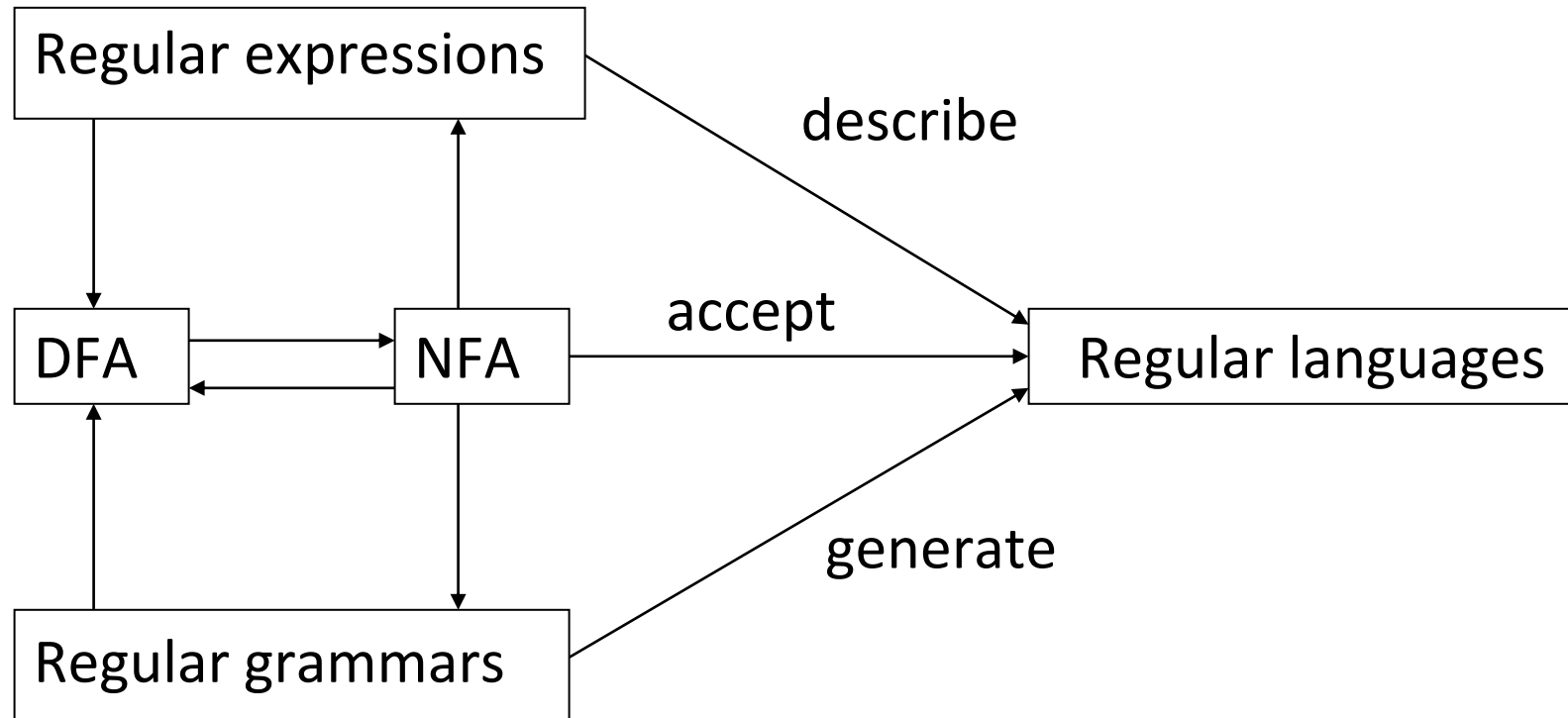
*digit*  $\rightarrow$  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
*non\_zero\_digit*  $\rightarrow$  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
*natural\_number*  $\rightarrow$  *non\_zero\_digit* *digit*\*

# 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





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

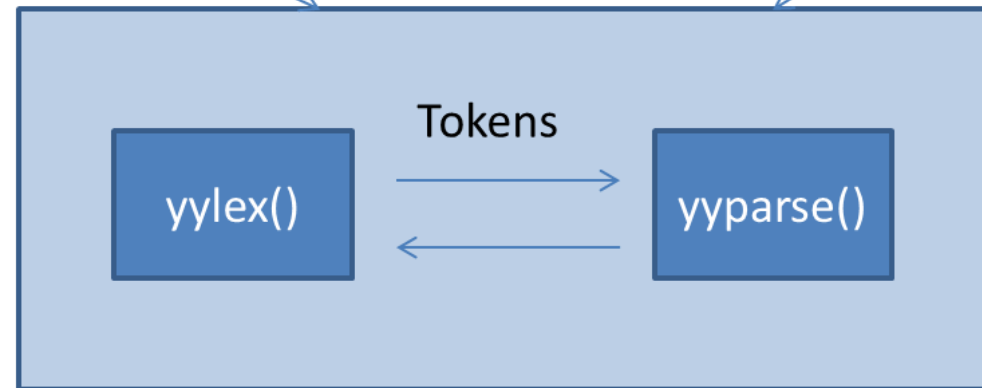
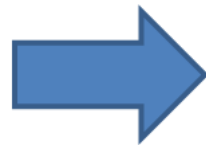
name.tab.c

Parser  
source code

y.tab.c file

yyparse()

Input



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)

# Finite Automata

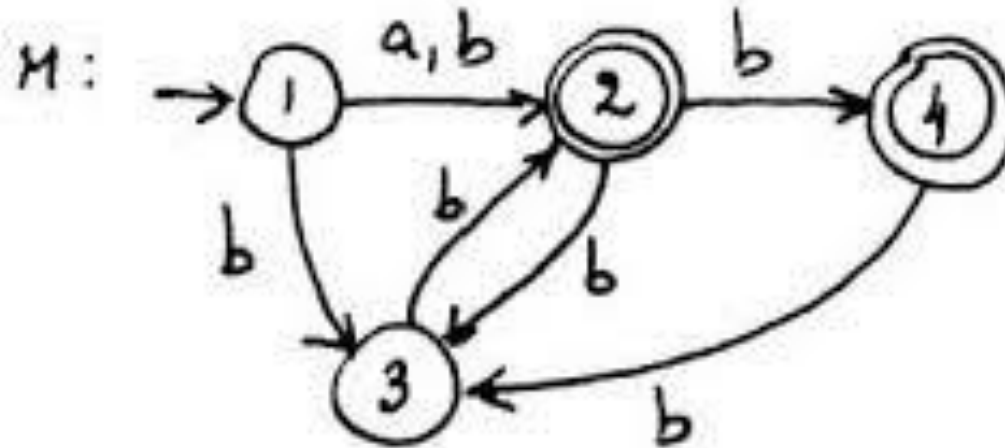
## SECTION 2

# 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.

$R = (a \mid b)(bb \mid b)^*$   
 $G: S \rightarrow aAB \mid bAb$   
 $A \rightarrow bbA \mid b \mid \epsilon$   
 $B \rightarrow bB \mid bAB \mid \epsilon$



# A Formal Definition of Regular Grammars

## Terminal, Non-terminal and Rules

---

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

- $N$  is a nonempty, finite set of *nonterminal symbols*,
- $\Sigma$  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  $A, B \in N$ ,  $a \in \Sigma$ , and  $\varepsilon$  the empty string, and
- $S \in N$  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?).

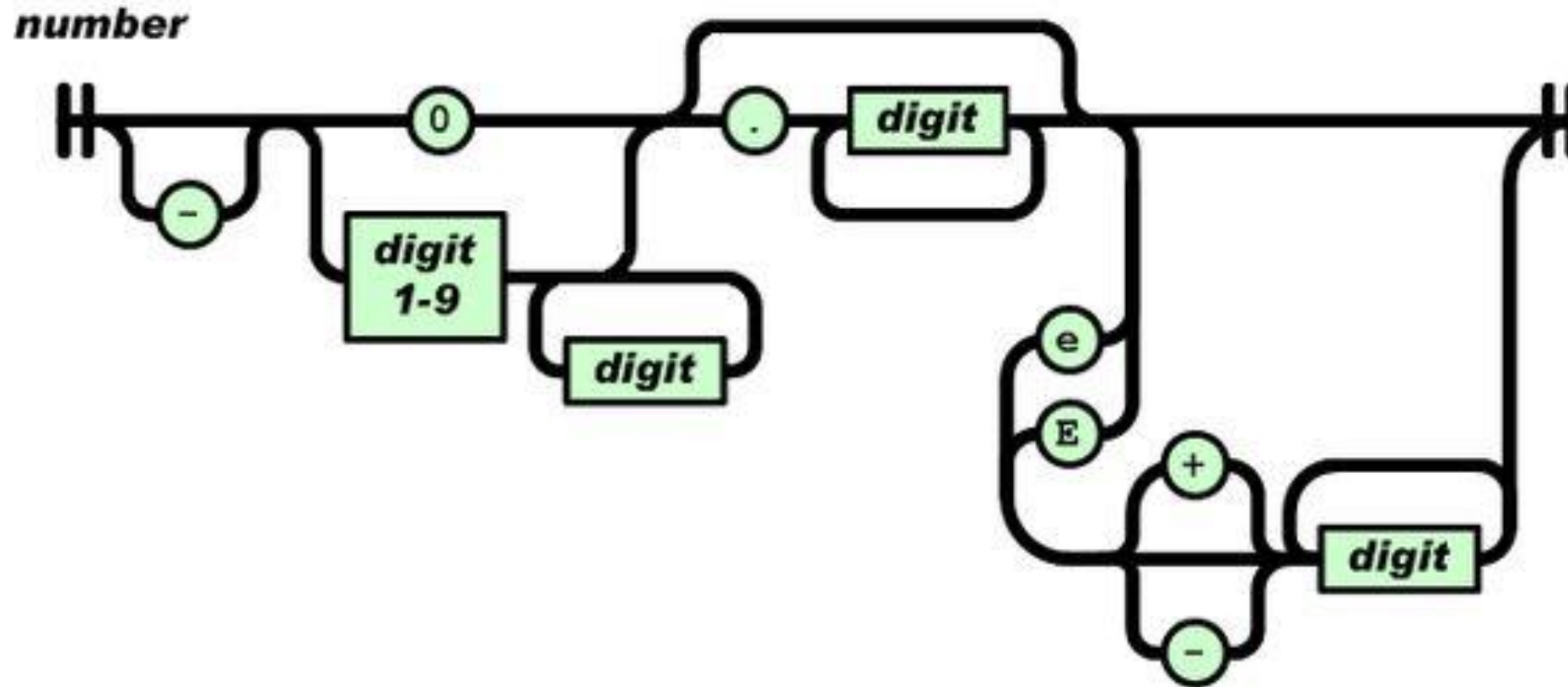


# A regular expression is one of the following

---

- A character
- The empty string, denoted by  $\varepsilon$
- 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

---

$number \longrightarrow integer \mid real$

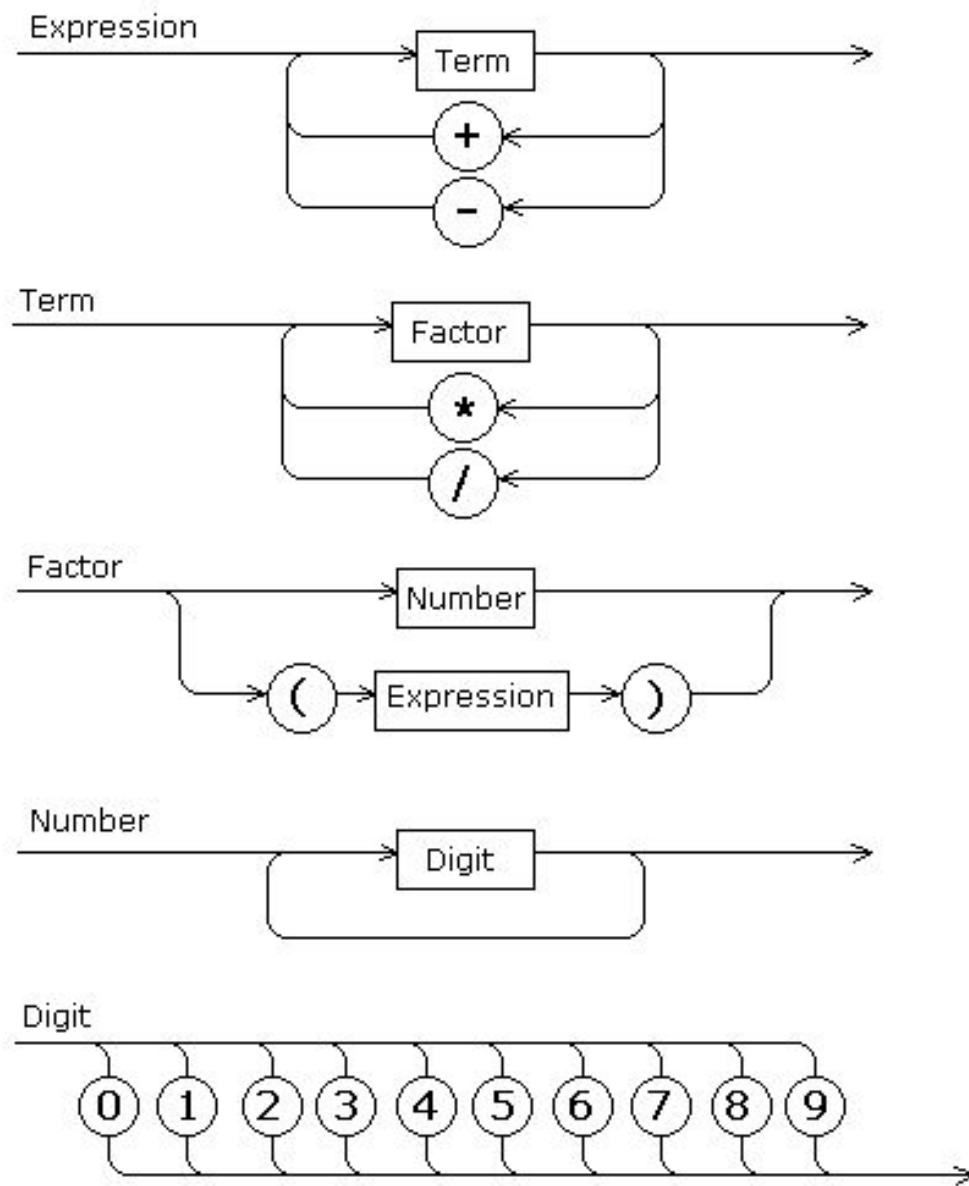
$integer \longrightarrow digit \, digit^*$

$real \longrightarrow integer \, exponent \mid decimal \, ( \, exponent \mid \epsilon \, )$

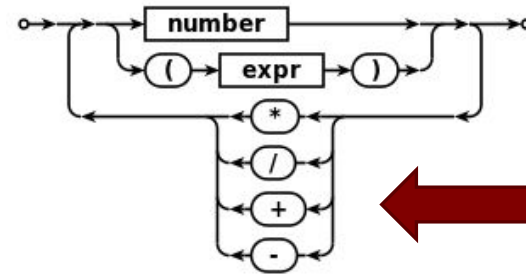
$decimal \longrightarrow digit^* \, ( \, . \, digit \mid digit \, . \, ) \, digit^*$

$exponent \longrightarrow ( \, e \mid E \, ) \, ( \, + \mid - \mid \epsilon \, ) \, integer$

$digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

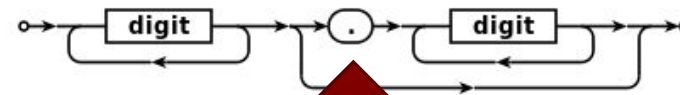


**expr:**

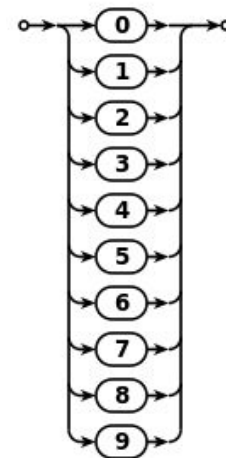


← No precedence

**number:**



**digit:**



↑ With decimal point for floating point and integer

# Regular Expressions

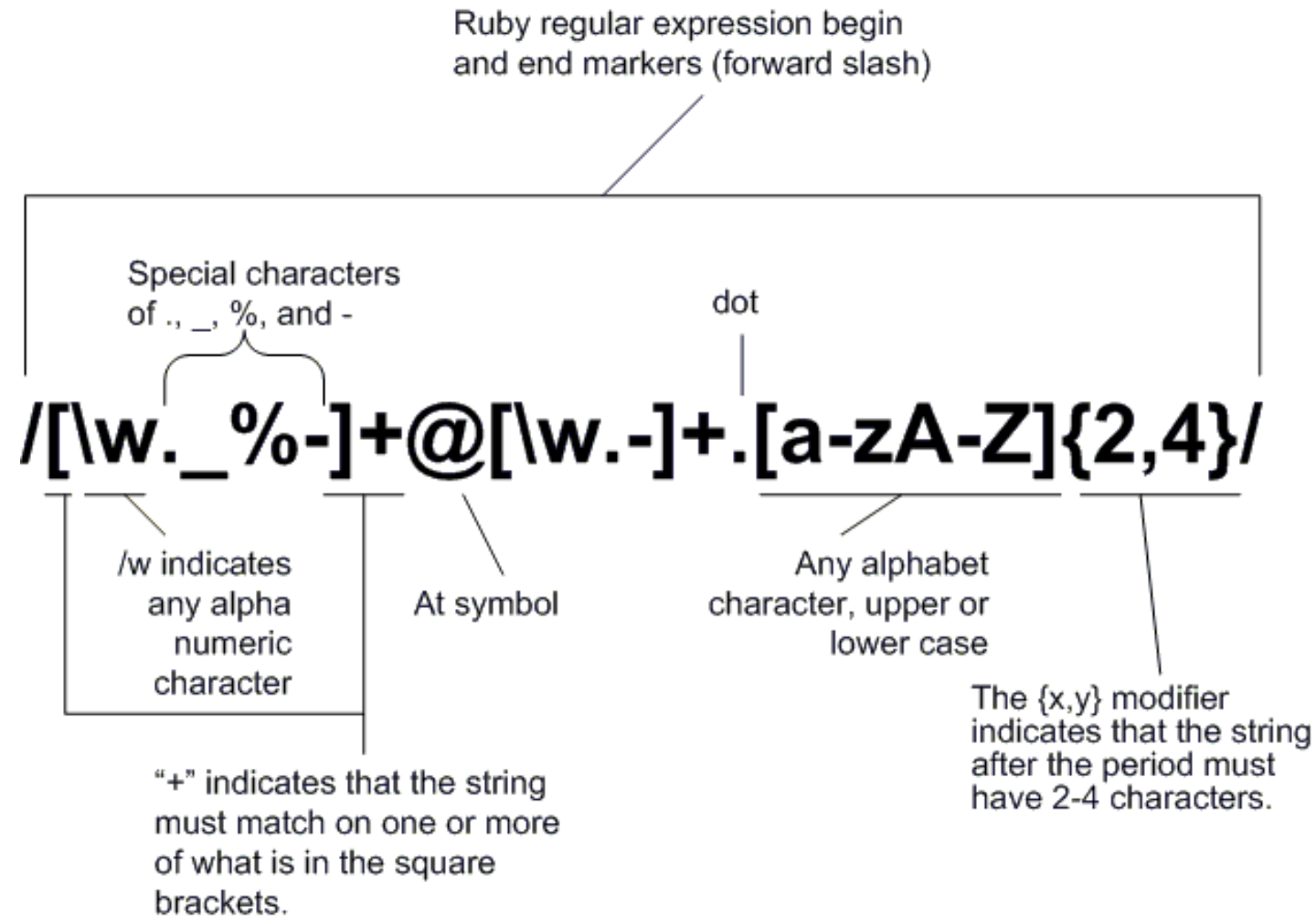
SECTION 3

# 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.)



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



Anchors		Sample Patterns			
^	Start of line +	([A-Za-z0-9-]+)	Letters, numbers and hyphens		
\A	Start of string +	(\d{1,2}\d{1,2}\d{4})	Date (e.g. 21/3/2006)		
\$	End of line +	([^\s]+(?:\.(jpg gif png))\.\2)	jpg, gif or png image		
\Z	End of string +	(^[1-9]{1}\$ ^[1-4]{1}[0-9]{1}\$ ^50\$)	Any number from 1 to 50 inclusive		
\b	Word boundary +	(#?([A-Fa-f0-9]){3}([A-Fa-f0-9]){3})?	Valid hexadecimal colour code		
\B	Not word boundary +	((?=[*\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).		
\<	Start of word	(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses		
\>	End of word	(\<(/?[^\>]+)\>)	HTML Tags		
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				
		Quantifiers	Ranges		
		*	0 or more +	.	Any character except new line (\n) +
		*?	0 or more, ungreedy +	(a b)	a or b +
		+	1 or more +	(...)	Group +
		+	1 or more, ungreedy +	(?:...)	Passive Group +
		?	0 or 1 +	[abc]	Range (a or b or c) +
		??	0 or 1, ungreedy +	[^abc]	Not a or b or c +
		{3}	Exactly 3 +	[a-q]	Letter between a and q +
		{3,}	3 or more +	[A-Q]	Upper case letter +
		{3,5}	3, 4 or 5 +		

Note

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

# Syntax of Regular Expressions

SECTION 4

# 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)

<code>.match(<i>regexp</i>)</code>	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
<code>.replace(<i>regexp</i>, <i>text</i>)</code>	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
<code>.search(<i>regexp</i>)</code>	returns first index where the given regular expression occurs
<code>.split(<i>delimiter</i>[,<i>limit</i>])</code>	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 "Mixxixxiippi"
- *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", "Goooogle", "Goooooogle", ...

## ? 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

# Syntax of Regular Expressions

SECTION 5

# 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 <sup>^</sup> 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:

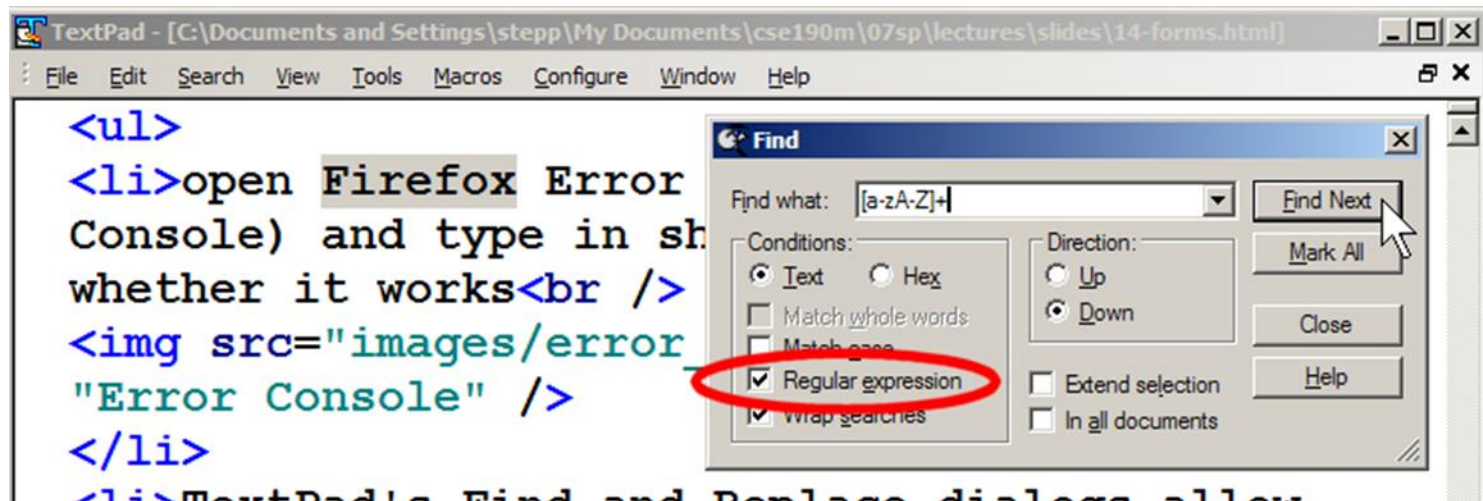
Replace:

With: 

☒ Regular expression

# Regexes in editors and tools

- Many editors allow regexes in their Find/Replace feature



- many command-line Linux/Mac tools support regexes  

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

# Context-Free Grammar

SECTION 6

# Context-Free Grammars

## Expressive Power Similar to BNF

---

- The notation for context-free grammars (**CFG**) is sometimes called Backus-Naur Form (**BNF**)
- A **CFG** consists of
  - A set of *terminals* **T**
  - A set of *non-terminals* **N**
  - A *start symbol* **S** (a non-terminal)
  - A set of *productions* (Rules)

CFG is a set of rules, terminals and non-terminals

Any set of strings that can be defined if we add recursion (to Reg Lang.) is called context-free language (CFL). CFL is generated by CFG.



# Expression grammar with precedence and associativity

---

Regular expressions work well for defining tokens. They are unable to specify nested constructs.

$$\begin{aligned} \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \\ &\quad \mid \text{expr op expr} \\ \text{op} &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

The arrow  $\rightarrow$  means “can have the form” or “goes to”.

# BNF and Extended BNF

Context Free Grammar is also called Backus-Naur Form (BNF)

---

- Each rule in **CFG** is known as a Production.
- The symbols on the left are known as variables or non-terminals.
- In the programming language, the terminals of the context-free grammar are the language's tokens.
- One of the non-terminals, the one on the left-side of first production is called the **Start Symbol**.
- Kleene's star  $*$  is not used and other regular expression is not used.

# Extended BNF

**BNF:** (one or many similar to Kleene notation)

$$id\_list \longrightarrow id ( , id )^*$$

is shorthand for

$$id\_list \longrightarrow id$$

$$id\_list \longrightarrow id\_list , id$$

The parentheses here are meta-symbols.

- Like the Kleene star and parentheses, the vertical bar is in some sense superfluous, though it was provided in the original BNF. [ | , ( ) , \* , and rules (for recursion) ]

The construct

$$op \longrightarrow + \mid - \mid * \mid /$$

can be considered shorthand for

$$op \longrightarrow +$$

$$op \longrightarrow -$$

$$op \longrightarrow *$$

$$op \longrightarrow /$$

which is also sometimes written

$$op \longrightarrow +$$

$$\longrightarrow -$$

$$\longrightarrow *$$

$$\longrightarrow /$$

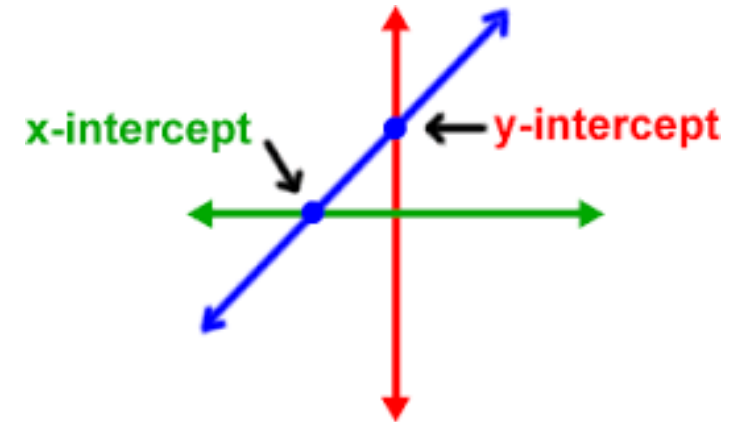
# Derivations and Parse Trees

---

- A context-free grammar shows us how to generate a syntactically valid string of terminals: Begin with the **start** symbol.
- The  $\Rightarrow$  meta-symbol is often pronounced “derives.” It indicates that the right-hand side was obtained by using a production to replace some nonterminal in the left-hand side.
- A series of replacement operations that shows how to derive a string of terminals from the start symbol is called a **derivation**.

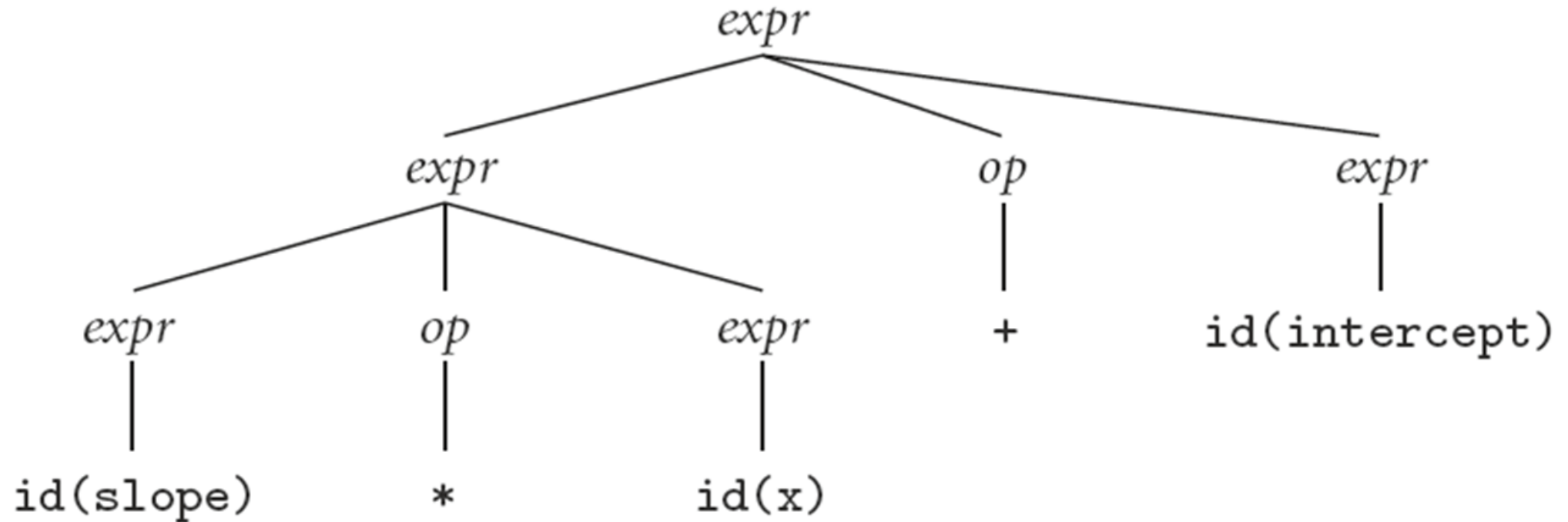
In this grammar, generate the string  
 "slope \* x + intercept"

$$\begin{aligned}
 \text{expr} &\Rightarrow \text{expr op } \underline{\text{expr}} \\
 &\Rightarrow \text{expr } \underline{\text{op}} \text{ id} \\
 &\Rightarrow \underline{\text{expr}} + \text{id} \\
 &\Rightarrow \text{expr op } \underline{\text{expr}} + \text{id} \\
 &\Rightarrow \text{expr } \underline{\text{op}} \text{ id} + \text{id} \\
 &\Rightarrow \underline{\text{expr}} * \text{id} + \text{id} \\
 &\Rightarrow \text{id} * \text{id} + \text{id} \\
 &\quad (\text{slope}) \quad (\text{x}) \quad (\text{intercept})
 \end{aligned}$$

$$\begin{aligned}
 \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \\
 &\quad \mid \text{expr op expr} \\
 \text{op} &\longrightarrow + \mid - \mid * \mid /
 \end{aligned}$$


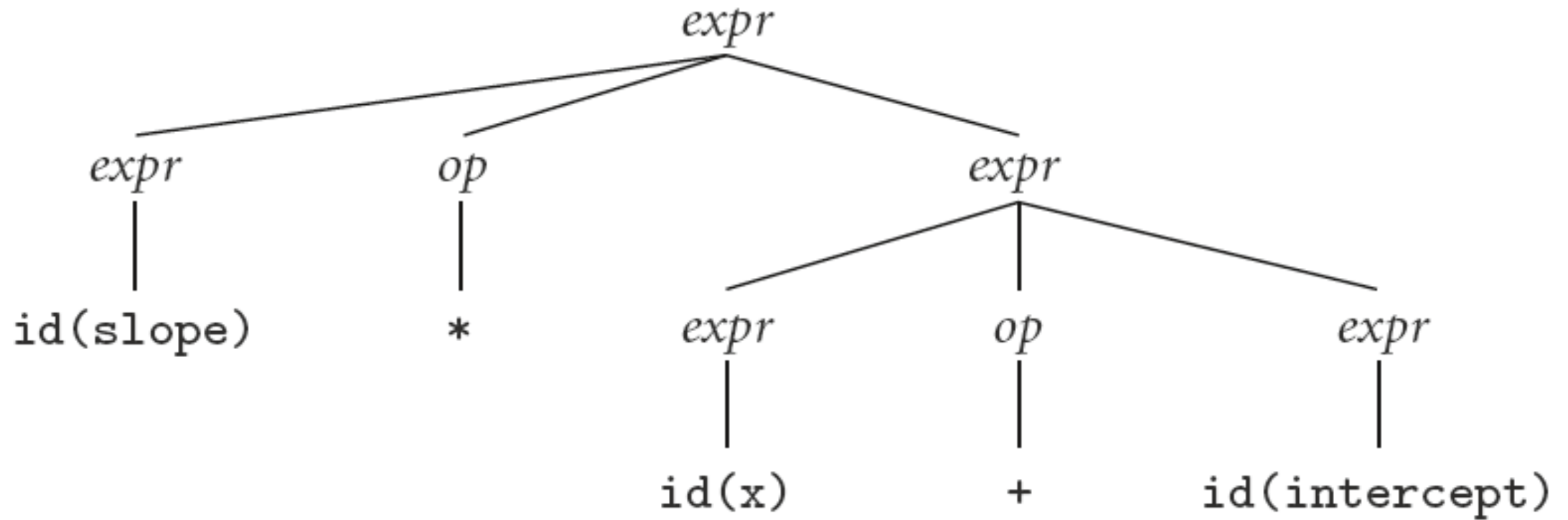
# Parse tree for expression grammar for "slope \* x + intercept"

---



# Alternate (Incorrect) Parse tree for "slope \* x + intercept"

---



Our grammar is ambiguous

A better version because it is unambiguous and captures precedence

---

1.  $expr \longrightarrow term \mid expr \text{ add\_op } term$
2.  $term \longrightarrow factor \mid term \text{ mult\_op } factor$
3.  $factor \longrightarrow id \mid number \mid - factor \mid ( expr )$
4.  $add\_op \longrightarrow + \mid -$
5.  $mult\_op \longrightarrow * \mid /$



# Parse tree for expression grammar (with left associativity) for $3 + 4 * 5$

