Principles of Programming Languages

COMP251: Syntax and Grammars

Prof. Dekai Wu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

Fall 2007

Part I

Language Description

Language Description: Motivation

"Able was I ere I saw Elba." — about Napoléon

How do you know that this is English, and not French or Chinese?

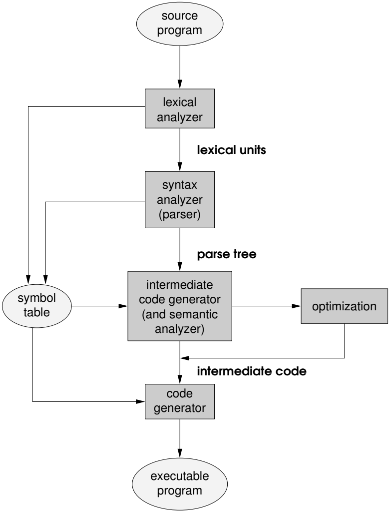
Language Description

A language has 2 parts:

1. **Syntax**
   - lexical syntax
     - describes how a sequence of *symbols* makes up *tokens (lexicon)* of the language
     - checked by a *lexical analyzer*
   - grammar
     - describes how a sequence of *tokens* makes up a valid *program*.
     - checked by a *parser*
2. **Semantics**

   specifies the *meaning* of a program

Compilation



Example 1: English Language

A word = some combination of the 26 letters, a,b,c, ...,z.

One form of a sentence = Subject + Verb + Object.

e.g. The student wrote a great program.

Example 2: Date Format

A date like 06/04/2010 may be written in the general format:

D D / D D / D D D D

where D = 0,1,2,3,4,5,6,7,8,9

But, does 03/09/1998 mean Sept 3rd, or March 9th?

Example 3: Real Numbers (Simplified)

Examples of reals: 0.45 12.3 .98

Examples of non-reals: 2+4i 1a2b 8 <

Informal rules:

- In general, a real number has three parts:
  - an integer part (I)
  - a dot "." symbol (.)
  - a fraction part (F)
- valid forms: $I.F$, $.F$
- I and F are strings of digits
- I may be empty but F cannot
- a digit is one of { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Expression: Examples

$$a + b \qquad 3 * a + b/c$$

$$\frac{-b+\sqrt{b^2-4*a*c}}{2*a} \qquad \frac{a*(1-R^n)}{1-R}$$

```
if (x > 10) then
     x /= 10
else
     x *= 2
```

c.f. "While I was coming to school, I saw a car accident."
The sentence is in the form of: "While E_1, E_2."

Expression Notation: Example 4

Goal: Add a to b.

Abstract Syntax Tree

Infix : $a + b$

Prefix : $+ab$

Postfix : $ab+$

```
  +
 / \
a   b
```

Abstract syntax tree is *independent* of notation.

Expression

- A constant or variable is an expression.
- In general, an expression has the form of a function:

$$E \triangleq \textbf{Op}\ (E_1, E_2, ..., E_k)$$

where **Op** is the operator, and $E_1, E_2, ..., E_k$ are the operands.

- An operator with k operands is said to have an arity of k; and **Op** is an k-ary operator.

unary operator :	$-x$
binary operator :	$x + y$
ternary operator :	$(x > y)\ ?\ x\ :\ y$

Infix, Prefix, Postfix, Mixfix

- Infix : E_1 **Op** E_2 (must be binary operator!)

$$a + b,\ a * b,\ a - b,\ a/b,\ a == b,\ a < b.$$

- Prefix : **Op** E_1 E_2 $\ldots E_k$

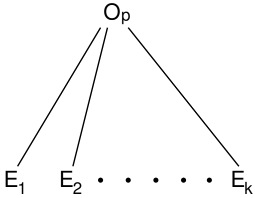
$$+ab,\ *ab,\ -ab,\ /ab,\ == ab,\ < ab.$$

- Postfix : E_1 E_2 $\ldots E_k$ **Op**

$$ab+,\ ab*,\ ab-,\ ab/,\ ab ==,\ ab < .$$

- Mixfix : e.g. **if** E_1 **then** E_2 **else** E_3

Abstract Syntax Tree



Expression Notation: Example 5

abstract syntax tree

infix : $3 * a + b/c$

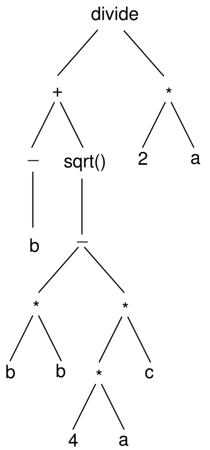
prefix : $+ * 3a/bc$

postfix : $3a * bc/+$

```
        +
       / \
      /   \
     *     /
    / \   / \
   3   a b   c
```

Note: Prefix and postfix notation does <u>not</u> require parentheses.

Expression Notation: Example 6



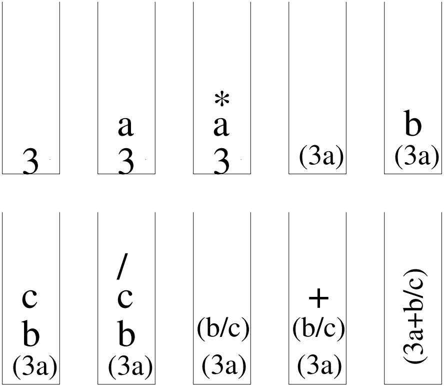
$$\text{infix}: \quad (-b + \sqrt{b^2 - 4 * a * c})/(2 * a)$$

$$\text{prefix}: \quad / + -b \sqrt{} - *bb * *4ac * 2a$$

$$\text{postfix}: \quad b - bb * 4a * c * - \sqrt{} + 2a * /$$

Postfix Evaluation: By a Stack

- infix expression: $3 * a + b/c$.
- postfix expression: $3a * bc/+$.



Precedence and Associativity in C++

Operator	Description	Associativity
[ ] . $\rightarrow$	array element structure member pointer	LEFT
- ++ - - $*$	minus increment decrement indirection	RIGHT
$*$ / %	multiply divide mod	LEFT
+ -	add subtract	LEFT
==	logical equal	LEFT
=	assignment	RIGHT

Precedence

Example: $1/2 + 3 * 4 = (1/2) + (3 * 4)$
because $*$, $/$ has a *higher precedence* over $+$, $-$.

Precedence rules decide which operators run first. In general,

$$x \ P \ y \ Q \ z \quad = \quad x \ P \ (\ y \ Q \ z \ )$$

if operator Q is at a higher precedence level than operator P.

Associativity: Binary Operators

Example: $1 - 2 + 3 - 4 = ((1 - 2) + 3) - 4$
because $+$, $-$ are *left associative*.

Associativity decides the grouping of operands with operators of the *same* level of precedence.
In general, if binary operator P, Q are of the same precedence level:

$$x \; P \; y \; Q \; z \quad = \quad x \; P \; (\; y \; Q \; z \;)$$

if operator P, Q are both right associative;

$$x \; P \; y \; Q \; z \quad = \quad (\; x \; P \;\; y \;) \; Q \; z$$

if operator P, Q are both left associative.
Question : What if $+$ is left while $-$ is right associative?

Associativity: Unary Operators

- Example in C++: $*a++ = *(a++)$
  because all unary operators in C++ are right-associative.
- In Pascal, all operators including unary operators are left-associative.
- In general, unary operators in many languages may be considered as non-associative as it is not important to assign an associativity for them, and their usage and semantics will decide their order of computation.

Question : Which of infix/prefix/postfix notation needs precedence or associative rules?

Summary on Syntax

- $\surd$ Will describe a language by a formal syntax and an informal semantics
- $\surd$ Syntax = lexical syntax + grammar
- $\surd$ Expression notation: infix, prefix, postfix, mixfix
- $\surd$ Abstract syntax tree: independent of notation
- $\surd$ Precedence and associativity of operators decide the order of applying the operators

Part II

Grammar

Grammar: Motivation

What do the following sentences really mean?

- 路不通行不得在此小便
- "I saw a small kid on the beach with a binocular."
- What is the final value of x?

```
x = 15
if (x > 20) then
if (x > 30) then
x = 8
else
x = 9
```

- 楊 乃 武 與 小 白 菜

Ambiguity in semantics is often caused by ambiguous grammar of the language.

A Formal Description: Example 7

$$
\begin{array}{lrcl}
1. & <\textit{real-number}> & ::= & <\textit{integer-part}> \,.\, <\textit{fraction}> \\
2. & <\textit{integer-part}> & ::= & <\text{empty}> \mid <\textit{digit-sequence}> \\
3. & <\textit{fraction}> & ::= & <\textit{digit-sequence}> \\
4. & <\textit{digit-sequence}> & ::= & <\textit{digit}> \mid <\textit{digit}><\textit{digit-sequence}> \\
5. & <\textit{digit}> & ::= & 0|1|2|3|4|5|6|7|8|9
\end{array}
$$

This is the context-free grammar of real numbers written in the Backus-Naur Form.

Context Free Grammar (CFG)

A context-free grammar has **4** components:

1. **A set of tokens or terminals**:
   atomic symbols of the language.

   English : `a, b, c, ...., z`
   Reals : `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .`

2. **A set of nonterminals**:
   variables denoting language constructs.

   English : $< \textit{Noun} >, < \textit{Verb} >, < \textit{Adjective} >, \ldots$
   Reals : $< \textit{real-number} >, < \textit{integer-part} >, < \textit{fraction} >,$
   $< \textit{digit-sequence} >, < \textit{digit} >$

Context Free Grammar ..

3. **A set of rules called productions:**
   for generating expressions of the language.

   nonterminal ::= a string of terminals and nonterminals

English :	$< Sentence > ::= < Noun > < Verb > < Noun >$
Reals :	$< integer\text{-}part > ::= <\text{empty}> \mid < digit\text{-}sequence >$

   Notice that CFGs allow only a single non-terminal on the left-hand side of any production rules.

4. **A nonterminal chosen as the start symbol:**
   represents the main construct of the language.

English :	$< Sentence >$
Reals :	$< real\text{-}number >$

The set of strings that can be generated by a CFG makes up a context-free language.

Backus-Naur Form (BNF)

One way to write context-free grammar.

- Terminals appear as they are.
- Nonterminals are enclosed by $<$ and $>$.
  e.g.: $<$ *real-number* $>$, $<$ *digit* $>$.
- The special empty string is written as $<$empty$>$.
- Productions with a common nonterminal may be abbreviated using the special "or" symbol "$|$".

  e.g. `X ::= W1, X ::= W2, ..., X ::= Wn`

  may be abbreviated as `X ::= W1` $|$ `W2` $| \cdots |$ `Wn`

Top-Down Parsing: Example 8

- A parser checks to see if a given expression or program can be derived from a given grammar.

Check if ".5" is a valid real number by finding from the CFG of Example 6 a leftmost derivation of ".5":

$< \textit{real-number} >$
$=> \; < \textit{integer-part} > . < \textit{fraction} >$ [Production 1]
$=> \; <\texttt{empty}> . < \textit{fraction} >$ [Production 2]
$=> \; . < \textit{fraction} >$ [By definition]
$=> \; . < \textit{digit-sequence} >$ [Production 3]
$=> \; . < \textit{digit} >$ [Production 4]
$=> \; .5$ [Production 5]

Bottom-Up Parsing: Example 9

Check if ".5" is a valid real number by finding from the CFG of Example 6 a rightmost derivation of ".5" in reverse:

$$
\begin{aligned}
\texttt{.5} \quad &= \quad \texttt{<empty>.5} \text{ [By definition]} \\
&=> \quad < \textit{integer-part} > .5 \text{ [Production 2]} \\
&=> \quad < \textit{integer-part} > . < \textit{digit} > \text{ [Production 5]} \\
&=> \quad < \textit{integer-part} > . < \textit{digit-sequence} > \text{ [Production 4]} \\
&=> \quad < \textit{integer-part} > . < \textit{fraction} > \text{ [Production 3]} \\
&=> \quad < \textit{real-number} > \text{ [Production 1]}
\end{aligned}
$$

Parse Tree: Example 10 [Real Numbers]

A parse tree of ".5" generated by the CFG of Example 6.

```
                 <real-number>
                   /    |     \
      <integer-part>    .   <fraction>
            |                    |
        <empty>          <digit-sequence>
                                 |
                              <digit>
                                 |
                                 5
```

Parse Tree

A parse tree shows how a string is generated by a CFG — the concrete syntax in a tree representation.

- Root = start symbol.
- Leaf nodes = terminals or <empty>.
- Non-leaf nodes = nonterminals
- For any subtree, the root is the left-side nonterminal of some production, while its children, if read from left to right, make up the right side of the production.
- The leaf nodes, read from left to right, make up a string of the language defined by the CFG.

Example 11: CFG/BNF [Expression]

$$
\begin{aligned}
<Expr> &::= <Expr><Op><Expr>\\
<Expr> &::= (<Expr>)\\
<Expr> &::= <Id>\\
<Op> &::= \texttt{+} \mid \texttt{-} \mid * \mid \texttt{/} \mid \texttt{=}\\
<Id> &::= \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

1. Terminals: `a, b, c, +, -, *, /, =, (, )`
2. Nonterminals: *Expr*, *Op*, *Id*
3. Start symbol: *Expr*

A parse tree of "$a + b - c$" generated by the CFG of Example 10:

```
                         <Expr>
                 /          |          \
         <Expr>           <Op>      <Expr>
        /   |   \           |          |
 <Expr>  <Op>  <Expr>       -        <Id>
    |      |      |                    |
  <Id>     +    <Id>                   c
    |             |
    a             b
```

Question: What is the difference between a parse tree and an abstract syntax tree?

Ambiguous Grammar: Example 13

A grammar is (syntactically) ambiguous if some string in its language is generated by more than one parse tree.

```
                  <Expr>
              /     |      \
       <Expr>     <Op>      <Expr>
         |          |      /   |   \
        <Id>        +  <Expr> <Op> <Expr>
         |                |     |     |
         a               <Id>   -    <Id>
                          |           |
                          b           c
```

Solution: Rewrite the grammar to make it unambiguous.

Handle Left Associativity: Example 14

CFG of Example 10 cannot handle "$a + b - c$" correctly.
$\Rightarrow$ Add a left recursive production.

$$
\begin{array}{rcl}
< Expr > & ::= & < Expr >< Op >< Term > \\
< Expr > & ::= & < Term > \\
< Term > & ::= & (< Expr >) \mid < Id > \\
< Op > & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \\
< Id > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

Handle Left Associativity ..

Now there is only one parse tree for “$a + b - c$”:

```
                <Expr>
              /    |   \
             /     |    \
       <Expr>     <Op>  <Term>
      /  |  \      |      |
<Expr> <Op> <Term> -     <Id>
  |      |    |           |
<Term>   +   <Id>         c
  |           |
 <Id>         b
  |
  a
```

CFG of Example 10 cannot handle "$a = b = c$" correctly.
$\Rightarrow$ Add a right recursive production.

$$
\begin{array}{rcl}
< \mathit{Assign} > & ::= & < \mathit{Expr} > \texttt{=} < \mathit{Assign} > \\
< \mathit{Assign} > & ::= & < \mathit{Expr} > \\
< \mathit{Expr} > & ::= & < \mathit{Expr} >< \mathit{Op} >< \mathit{Term} > \mid < \mathit{Term} > \\
< \mathit{Term} > & ::= & (< \mathit{Expr} >) \mid < \mathit{Id} > \\
< \mathit{Op} > & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
< \mathit{Id} > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

Question: this grammar will accept strings like " a + b = c - d ". Try to correct it.

Handling Right Associativity ..

Now there is only one parse tree for "$a = b = c$":

```
                <Assign>
               /   |   \
              /    |    \
        <Expr>     =    <Assign>
          |              /  |  \
        <Term>      <Expr> = <Assign>
          |            |         |
         <Id>       <Term>    <Expr>
          |            |         |
          a          <Id>     <Term>
                       |         |
                       b       <Id>
                                 |
                                 c
```

Handling Precedence: Example 16

CFG of Example 10 cannot handle "$a + b * c$" correctly.
⇒ Add one nonterminal (plus appropriate productions) for each precedence level.

$$
\begin{array}{rcl}
<Assign> & ::= & <Expr> = <Assign> \mid <Expr> \\
<Expr> & ::= & <Expr> + <Term> \\
<Expr> & ::= & <Expr> - <Term> \mid <Term> \\
<Term> & ::= & <Term> * <Factor> \\
<Term> & ::= & <Term> / <Factor> \mid <Factor> \\
<Factor> & ::= & (<Expr>) \mid <Id> \\
<Id> & ::= & \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}
\end{array}
$$

Handling Precedence ..

Now there is only one parse tree for "$a + b * c$":

```
           <Assign>
               |
            <Expr>
           /   |   \
          /    |    \
   <Expr>      +     <Term>
     |              /  |   \
   <Term>      <Term> * <Factor>
     |            |         |
  <Factor>     <Factor>    <Id>
     |            |         |
    <Id>         <Id>       c
     |            |
     a            b
```

Tips on Handling Precedence/Associativity

- left associativity $\Rightarrow$ left-recursive production
- right associativity $\Rightarrow$ right-recursive production
- n levels of precedence
  - divide the operators into n groups
  - write productions for each group of operators
  - start with operators with the lowest precedence
- In all cases, introduce new non-terminals whenever necessary.
- In general, one needs a new non-terminal for each new group of operators of different associativity and different precedence.

Dangling-Else: Example 17

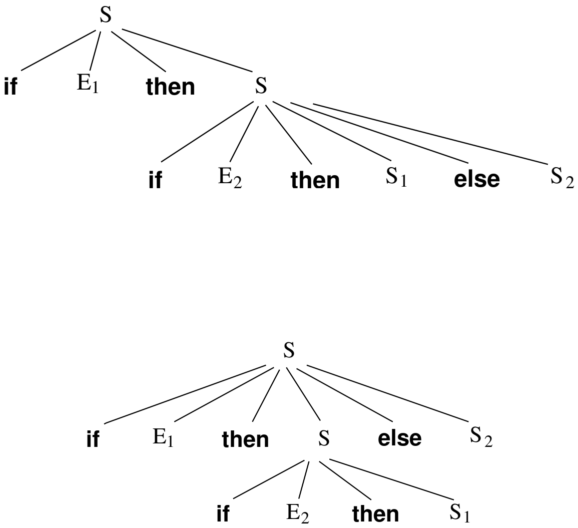
Consider the following grammar:

$$
\begin{array}{lcl}
< S > & ::= & \textbf{if} < E > \textbf{then} < S > \\
< S > & ::= & \textbf{if} < E > \textbf{then} < S > \textbf{else} < S >
\end{array}
$$

- How many parse trees can you find for the statement:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

Dangling-Else ..



Dangling-Else ...

- Ambiguity is often a property of a grammar, not of a language.

Solution: matching an "**else**" with the nearest unmatched "**if**" .
i.e. the first case.

More CFG Examples

1.
$$\begin{array}{lcl} < S > & ::= & < A >< B >< C > \\ < A > & ::= & \mathrm{a}< A > \mid \mathrm{a} \\ < B > & ::= & \mathrm{b}< B > \mid \mathrm{b} \\ < C > & ::= & \mathrm{c}< C > \mid \mathrm{c} \end{array}$$

2.
$$\begin{array}{lcl} < S > & ::= & < A > \mathrm{a} < B > \mathrm{b} \\ < A > & ::= & < A > \mathrm{b} \mid \mathrm{b} \\ < B > & ::= & \mathrm{a}< B > \mid \mathrm{a} \end{array}$$

3.
$$\begin{array}{rcl} <\text{stmts}> & ::= & <\text{empty}> \mid <\text{stmt}> \text{ ; } <\text{stmts}> \\ <\text{stmt}> & ::= & <\text{id}> := <\text{expr}> \\ & & \mid \text{ if } <\text{expr}> \text{ then } <\text{stmt}> \\ & & \mid \text{ if } <\text{expr}> \text{ then } <\text{stmt}> \text{ else } <\text{stmt}> \\ & & \mid \text{ while } <\text{expr}> \text{ do } <\text{stmt}> \\ & & \mid \text{ begin } <\text{stmts}> \text{ end} \end{array}$$

Non-Context Free Grammars: Examples

1.
$$\begin{aligned} <S> &::= <B><A><C> \mid <C><A><B> \\ \texttt{b}<A> &::= \texttt{c}<A><B> \mid <B> \\ \texttt{c}<A> &::= \texttt{b}<A><C> \mid <C> \\ <B> &::= \texttt{b} \\ <C> &::= \texttt{c} \end{aligned}$$

$\Rightarrow L = \{ (cb)^n,\ b(cb)^n,\ (bc)^n,\ c(bc)^n \}$.

2. $L = \{ wcw | w \text{ is a string of } a\text{'s or } b\text{'s} \}$.

   This language abstracts the problem of checking that an identifier is declared before its use in a program.
   The first w = declaration of the identifier, and
   the second w = its use in the program.

Summary on Grammar

- $\surd$ Context-free grammar (CFG) is commonly used to specify most of the syntax of a programming language.
- $\surd$ However, most programming languages are not CFL!
- $\surd$ CFG is commonly written in Backus-Naur Form (BNF).
- $\surd$ CFG = (Terminals, Nonterminals, Productions, Start Symbol)
- $\surd$ A program is valid if we may construct a parse tree, or a derivation from the grammar.
- $\surd$ Associativity and precedence of operations are part of the design of a CFG.
- $\surd$ Avoid ambiguous grammars by rewriting them or imposing parsing rules.

Part III

Regular Grammar, Regular Expression

Regular Grammars

Regular Grammars are a subset of CFGs in which all productions are in one of the following forms:

1. Right-Regular Grammar

```
<A> ::= x
<A> ::= x<B>
```

2. Left-Regular Grammar

```
<A> ::= x
<A> ::= <B>x
```

where A and B are non-terminals and x is a string of terminals.

```
<S> ::= a<A>
<S> ::= b<B>
<S> ::= <empty>
<A> ::= a<S>
<B> ::= bb<S>
```

What is the regular language this RG generates?

Regular Expressions

Regular expressions (RE) are succinct representations of RGs using the following notations.

Sub-Expression	Meaning
x	the single char 'x'
.	any single char except the newline
[abc]	char class consisting of 'a','b', or'c'
[$\wedge$abc]	any char except 'a','b','c'
r*	repeat "r" zero or more times
r+	repeat "r" 1 or more times
r?	zero or 1 occurrence of "r"
rs	concatenation of RE "r" and RE "s"
(r)s	"r" is evaluated and concatenated with "s"
r | s	RE "r" or RE "s"
`\x`	escape sequences for white-spaces and special symbols: `\b` `\n` `\r` `\t`

Precedence of Regular Expression Operators

The following table gives the order of RE operator precedence from the highest precedence to the lowest precedence.

Function	Operator
parenthesis	`( )`
counters	`* + ? { }`
concatenation	
disjunction	`|`

RE Example 2: Regular Expression Notations

RE	Meaning
abc	the string "abc"
a+b+	$\{a^m b^n : m, n \geq 1\}$
a*b*c	$\{a^m b^n c : m, n \geq 0\}$
a*b*c?	$\{a^m b^n c \text{ or } a^m b^n : m, n \geq 0\}$
xy(abc)+	$\{\text{xy(abc)}^n : n \geq 1\}$
xy[abc]	{xya, xyb, xyc}
xy(a|b)	{xya, xyb}

Questions: What are the following REs?

- foo|bar*
- foo|(bar)*
- (foo|bar)*

RE Example 3: Regular Expressions

- REs are commonly used for pattern matching in editors, word processors, commandline interpreters, etc.
- The REs used for searching texts in Unix (vi, emacs, perl, grep), Microsoft Word v.6+, and Word Perfect are almost identical.
- Examples:
  - identifiers in C++:
  - real numbers:
  - email addresses:
  - white spaces:
  - all C++ source or include files:

Summary on Regular Grammars

- √ There are algorithms to prove if a language is regular.
- √ There are algorithms to prove if a language is context-free too.
- √ English is not RL, nor CFL.
- √ REs are commonly used for text search.
- √ Different applications may extend the standard RE notations.