“Able was I ere I saw Elba.” — about Napoléon

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

source program

lexical analyzer

lexical units

syntax analyzer (parser)

parse tree

symbol table

intermediate code generator (and semantic analyzer)

optimization

intermediate code

code generator

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

\[
\text{D D / D D / D D D D}
\]

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

\[ a + b \quad \frac{3 \times a + b}{c} \]

\[ \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \frac{a(1 - R^n)}{1 - R} \]

\[
\text{if} \ (x > 10) \ \text{then} \\
\quad x /= 10 \\
\text{else} \\
\quad x *= 2
\]

c.f. “While I was coming to school, I saw a car accident.”
The sentence is in the form of: “While \( E_1, E_2. \)”
Goal: Add $a$ to $b$.

**Abstract Syntax Tree**

- **Infix**: $a + b$
- **Prefix**: $+ab$
- **Postfix**: $ab+$

Abstract syntax tree is *independent* of notation.
A constant or variable is an expression.

In general, an expression has the form of a function:

\[ E \triangleq \text{Op} \, (E_1, E_2, \ldots, E_k) \]

where \( \text{Op} \) is the operator, and \( E_1, E_2, \ldots, E_k \) are the operands.

An operator with \( k \) operands is said to have an arity of \( k \); and \( \text{Op} \) is an \( k \)-ary operator.

- **Unary operator**: \(-x\)
- **Binary operator**: \(x + y\)
- **Ternary operator**: \((x > y) \, ? \, x \, : \, y\)
Infix, Prefix, Postfix, Mixfix

- **Infix**: $E_1 \text{ Op } E_2$ (must be binary operator!)
  
  \[a + b, \ a \times b, \ a - b, \ a/b, \ a == b, \ a < b.\]

- **Prefix**: \text{Op } E_1 \ E_2 \ldots E_k
  
  \[+ab, \ *ab, \ -ab, \ /ab, \ == ab, \ < ab.\]

- **Postfix**: $E_1 \ E_2 \ldots E_k \text{ Op}$
  
  \[ab+, \ ab*, \ ab-, \ ab/, \ ab ==, \ ab <.\]

- **Mixfix**: e.g. \text{if } E_1 \text{ then } E_2 \text{ else } E_3
Note: Prefix and postfix notation does not require parentheses.
Expression Notation: Example 6

infix : \((−b + \sqrt{b^2 − 4∗a∗c})/(2∗a)\)

prefix : / + −b √ − *bb * *4ac * 2a

postfix : b − bb * 4a * c * − √ + 2a * /
Postfix Evaluation: By a Stack

- **infix** expression: \( 3 \times a + b/c \).
- **postfix** expression: \( 3a \times bc/ + \).

\[
\begin{array}{cccc}
3 & a & \times & a \\
3 & 3 & (3a) & b \\
\hline
c & / & c & (b/c) \\
b & b & (3a) & (3a) \\
\hline
\end{array}
\]
**Precedence and Associativity in C++**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>array element</td>
<td>LEFT</td>
</tr>
<tr>
<td>·</td>
<td>structure member</td>
<td></td>
</tr>
<tr>
<td>→</td>
<td>pointer</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
<td>RIGHT</td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>RIGHT</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>indirection</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>LEFT</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>mod</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>LEFT</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>logical equal</td>
<td>LEFT</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>RIGHT</td>
</tr>
</tbody>
</table>
Example: \( \frac{1}{2} + 3 \times 4 = \left( \frac{1}{2} \right) + (3 \times 4) \) because \( \ast, / \) has a higher precedence over \( +, - \).

**Precedence rules** decide which operators run first. In general,

\[
x P y Q z = x P (y Q z)
\]

if operator \( Q \) is at a higher precedence level than operator \( P \).
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:

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

if operator $P$, $Q$ are both right associative;

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

if operator $P$, $Q$ are both left associative.

**Question:** What if $+$ is left while $-$ is right associative?
Example in C++: $\ast a ++ = \ast (a + +)$
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?
Will describe a language by a formal syntax and an informal semantics

Syntax = lexical syntax + grammar

Expression notation: infix, prefix, postfix, mixfix

Abstract syntax tree: independent of notation

Precedence and associativity of operators decide the order of applying the operators
Part II

Grammar
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?

\[
\begin{align*}
x &= 15 \\
&\text{if } (x > 20) \text{ then} \\
&\quad \text{if } (x > 30) \text{ then} \\
&\quad \quad x = 8 \\
&\quad \text{else} \\
&\quad x = 9
\end{align*}
\]

Ambiguity in semantics is often caused by ambiguous grammar of the language.
This is the context-free grammar of real numbers written in the Backus-Naur Form.

1. `<real-number>` ::= `<integer-part>` . `<fraction>`
2. `<integer-part>` ::= `<empty>` | `<digit-sequence>`
3. `<fraction>` ::= `<digit-sequence>`
4. `<digit-sequence>` ::= `<digit>` | `<digit>` `<digit-sequence>`
5. `<digit>` ::= 0|1|2|3|4|5|6|7|8|9
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:  < Noun >, < Verb >, < Adjective >, . . .
   - Reals:  < real-number >, < integer-part >, < fraction >, < digit-sequence >, < digit >
A set of rules called **productions**:
for generating expressions of the language.

```
nonterminal ::= a string of terminals and nonterminals
```

**English**:

```text
< Sentence > ::= < Noun > < Verb > < Noun >
```

**Reals**:

```text
< integer-part > ::= < empty > | < digit-sequence >
```

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

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

**English**:

```text
< Sentence >
```

**Reals**:

```text
< real-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.  
  
  \[
  X ::= W_1, X ::= W_2, \ldots, X ::= W_n
  \]
  
  may be abbreviated as \[X ::= W_1 \mid W_2 \mid \cdots \mid W_n\]
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”:

\[
\begin{align*}
< \text{real-number} > &
\Rightarrow < \text{integer-part} > . < \text{fraction} > \quad \text{[Production 1]} \\
\Rightarrow < \text{empty} > . < \text{fraction} > \quad \text{[Production 2]} \\
\Rightarrow . < \text{fraction} > \quad \text{[By definition]} \\
\Rightarrow . < \text{digit-sequence} > \quad \text{[Production 3]} \\
\Rightarrow . < \text{digit} > \quad \text{[Production 4]} \\
\Rightarrow .5 \quad \text{[Production 5]}
\end{align*}
\]
Check if “.5” is a valid real number by finding from the CFG of Example 6 a rightmost derivation of “.5” in reverse:

\[
.5 = <\text{empty}>.5 \quad \text{[By definition]}
\]

\[
=> <\text{integer-part}>.5 \quad \text{[Production 2]}
\]

\[
=> <\text{integer-part}>. <\text{digit}> \quad \text{[Production 5]}
\]

\[
=> <\text{integer-part}>. <\text{digit-sequence}> \quad \text{[Production 4]}
\]

\[
=> <\text{integer-part}>. <\text{fraction}> \quad \text{[Production 3]}
\]

\[
=> <\text{real-number}> \quad \text{[Production 1]}
\]
A parse tree of "0.5" generated by the CFG of Example 6.
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]

\[
\begin{align*}
< Expr > & ::= < Expr > < Op > < Expr > \\
< Expr > & ::= ( < Expr > ) \\
< Expr > & ::= < Id > \\
< Op > & ::= + | - | * | / | = \\
< Id > & ::= a | b | c
\end{align*}
\]

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

Solution: Rewrite the grammar to make it unambiguous.
CFG of Example 10 cannot handle “$a + b - c$” correctly.
⇒ Add a left recursive production.

$$
<\text{Expr}> ::= \text{<Expr>\<Op>\<Term>}
\text{<Expr>} ::= \text{<Term>}
\text{<Term>} ::= (\text{<Expr>})|\text{<Id>}
\text{<Op>} ::= +|-|*|/|=\n\text{<Id>} ::= a|b|c
$$
Now there is **one** parse tree for “a + b − c”:
Handling Right Associativity: Example 15

CFG of Example 10 cannot handle “a = b = c” correctly.
⇒ Add a right recursive production.

```plaintext
< Assign > ::= < Expr > = < Assign >
< Assign > ::= < Expr >
< Expr > ::= < Expr > < Op > < Term > | < Term >
< Term > ::= ( < Expr > ) | < Id >
< Op > ::= + | - | * | /
< Id > ::= a | b | c
```

Question: this grammar will accept strings like “a + b = c - d”. Try to correct it.
Now there is only one parse tree for “$a = b = c$”:

\[
\begin{array}{c}
\text{<Assign>} \\
/ & | & \backslash \\
/ & | & \backslash \\
\text{<Expr>} & = & \text{<Assign>} \\
/ & | & \backslash \\
\text{<Term>} & \text{<Expr>} & \text{<Assign>} \\
/ & | & | \\
\text{<Id>} & \text{<Term>} & \text{<Expr>} \\
/ & | & | \\
a & \text{<Id>} & \text{<Term>} \\
/ & | & | \\
b & \text{<Id>} \\
/ & | \\
c
\end{array}
\]
Handling Precedence: Example 16

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

\[
\begin{align*}
\text{< Assign >} & ::= \text{< Expr >} = \text{< Assign >} | \text{< Expr >} \\
\text{< Expr >} & ::= \text{< Expr >} + \text{< Term >} \\
\text{< Expr >} & ::= \text{< Expr >} - \text{< Term >} | \text{< Term >} \\
\text{< Term >} & ::= \text{< Term >} \times \text{< Factor >} \\
\text{< Term >} & ::= \text{< Term >} / \text{< Factor >} | \text{< Factor >} \\
\text{< Factor >} & ::= (\text{< Expr >}) | \text{< Id >} \\
\text{< Id >} & ::= a | b | c
\end{align*}
\]
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 ⇒ **left-recursive** production
- **right** associativity ⇒ **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.
Consider the following grammar:

\[ \langle S \rangle ::= \text{if } \langle E \rangle \text{ then } \langle S \rangle \]

\[ \langle S \rangle ::= \text{if } \langle E \rangle \text{ then } \langle S \rangle \text{ else } \langle S \rangle \]

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

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]
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

\[<S> ::= <A><B><C>\]
\[<A> ::= a<A> | a\]
\[<B> ::= b<B> | b\]
\[<C> ::= c<C> | c\]

\[<S> ::= <A>a <B>b\]
\[<A> ::= <A>b | b\]
\[<B> ::= a<B> | a\]

\[<\text{stmts}> ::= <\text{empty}> | <\text{stmt}> ; <\text{stmts}>\]
\[<\text{stmt}> ::= <\text{id}> ::= <\text{expr}>\]
\[| \text{if <expr> then <stmt>}\]
\[| \text{if <expr> then <stmt> else <stmt>}\]
\[| \text{while <expr> do <stmt>}\]
\[| \text{begin <stmts> end}\]
Non-Context Free Grammars: Examples

\[ S ::= B A C | C A B \]
\[ A ::= b A B | B \]
\[ A ::= c A B | B \]
\[ B ::= b C | C \]
\[ C ::= c B | B \]

\[ L = \{ (cb)^n, b(cb)^n, (bc)^n, c(bc)^n \} \]

1. 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.

2. \[ L = \{ wcw | w \text{ is a string of } a\text{'s or } b\text{'s \} \]
Context-free grammar (CFG) is commonly used to specify most of the syntax of a programming language.

However, most programming languages are not CFL!

CFG is commonly written in Backus-Naur Form (BNF).

CFG = (Terminals, Nonterminals, Productions, Start Symbol)

A program is valid if we may construct a parse tree, or a derivation from the grammar.

Associativity and precedence of operations are part of the design of a CFG.

Avoid ambiguous grammars by rewriting them or imposing parsing rules.
Part III

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

1. **Right-Regular Grammar**

   \[
   \begin{align*}
   \langle A \rangle & \ ::= \ x \\
   \langle A \rangle & \ ::= \ x\langle B \rangle
   \end{align*}
   \]

2. **Left-Regular Grammar**

   \[
   \begin{align*}
   \langle A \rangle & \ ::= \ x \\
   \langle A \rangle & \ ::= \ \langle B \rangle x
   \end{align*}
   \]

where \( A \) and \( B \) are non-terminals and \( x \) is a string of terminals.
RE Example 1: Right-Regular Grammar

\[
\begin{align*}
<S> &::= a<A> \\
<S> &::= b<B> \\
<S> &::= \text{<empty>} \\
<A> &::= a<S> \\
<B> &::= bb<S>
\end{align*}
\]

What is the regular language this RG generates?
Regular expressions (RE) are succinct representations of RGs using the following notations.

<table>
<thead>
<tr>
<th>Sub-Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>the single char ‘x’</td>
</tr>
<tr>
<td>.</td>
<td>any single char except the newline</td>
</tr>
<tr>
<td><code>[abc]</code></td>
<td>char class consisting of ‘a’, ‘b’, or ‘c’</td>
</tr>
<tr>
<td><code>[^abc]</code></td>
<td>any char except ‘a’, ‘b’, ‘c’</td>
</tr>
<tr>
<td><code>r*</code></td>
<td>repeat ”r” zero or more times</td>
</tr>
<tr>
<td><code>r+</code></td>
<td>repeat ”r” 1 or more times</td>
</tr>
<tr>
<td><code>r?</code></td>
<td>zero or 1 occurrence of ”r”</td>
</tr>
<tr>
<td><code>rs</code></td>
<td>concatenation of RE ”r” and RE ”s”</td>
</tr>
<tr>
<td><code>(r)s</code></td>
<td>”r” is evaluated and concatenated with ”s”</td>
</tr>
<tr>
<td>`r</td>
<td>s`</td>
</tr>
<tr>
<td><code>\x</code></td>
<td>escape sequences for white-spaces and special symbols: \b \n \r \t</td>
</tr>
</tbody>
</table>
The following table gives the order of RE operator precedence from the highest precedence to the lowest precedence.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>parenthesis</td>
<td>( )</td>
</tr>
<tr>
<td>counters</td>
<td>* + ? { }</td>
</tr>
<tr>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>abc</td>
<td>the string &quot;abc&quot;</td>
</tr>
<tr>
<td>a+b+</td>
<td>{a^m b^n : m, n \geq 1}</td>
</tr>
<tr>
<td>a<em>b</em>c</td>
<td>{a^m b^n c : m, n \geq 0}</td>
</tr>
<tr>
<td>a<em>b</em>c?</td>
<td>{a^m b^n c \text{ or } a^m b^n : m, n \geq 0}</td>
</tr>
<tr>
<td>xy(abc)+</td>
<td>{xy(abc)^n : n \geq 1}</td>
</tr>
<tr>
<td>xy[abc]</td>
<td>{xya, xyb, xyc}</td>
</tr>
<tr>
<td>xy(a</td>
<td>b)</td>
</tr>
</tbody>
</table>

Questions: What are the following REs?

- foo|bar*
- foo|(bar)*
- (foo|bar)*
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:
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.