Part I

Language Description
“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:

```
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?
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

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

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

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 \(E_1, E_2\).”
Goal: Add $a$ to $b$.

Abstract Syntax Tree

\[
\begin{array}{c}
\text{Infix: } a + b \\
\text{Prefix: } +ab \\
\text{Postfix: } ab+ \\
\end{array}
\]

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

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

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

where \( Op \) 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 \( Op \) 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!)
  
  \( a + b, \ a * b, \ a - b, \ a/b, \ a == b, \ a < b. \)

- **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. if \( E_1 \) then \( E_2 \) else \( E_3 \)
Abstract Syntax Tree

Prof. Dekai Wu, HKUST (dekai@cs.ust.hk)  COMP3031 (Fall 2012, L1)
Expression Notation: Example 5

abstract syntax tree

infix : 3 * a + b/c
prefix : + * 3a/bc
postfix : 3a * bc/+ 

Note: Prefix and postfix notation does not require parentheses.
Expression Notation: Example 6

- **Infix**: \((-b + \sqrt{b^2 - 4 \times a \times c}) / (2 \times a)\)
- **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**: $3a \times bc/+$.
## 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></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: $1/2 + 3 \times 4 = (1/2) + (3 \times 4)$ because $\times$, $/$ has a *higher precedence* over $+$, $\neg$.

**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++: \( *a ++ = *(a++) \)
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?
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
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 $x$?

```plaintext
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.
1. \(< real\text{-}number > \) ::= \(< integer\text{-}part > . < fraction >
2. \(< integer\text{-}part > \) ::= \(< empty > | < digit\text{-}sequence >
3. \(< fraction > \) ::= \(< digit\text{-}sequence >
4. \(< digit\text{-}sequence > \) ::= \(< digit > | < digit > < digit\text{-}sequence >
5. \(< digit > \) ::= 0|1|2|3|4|5|6|7|8|9

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

\[
<\text{Sentence}> ::= <\text{Noun}> <\text{Verb}> <\text{Noun}>
\]

**Reals** :

\[
<\text{integer-part}> ::= <\text{empty}>|<\text{digit-sequence}>
\]

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”:

$$<\text{real-number}>$$

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

\[
.5 = <\text{empty}>.5 \text{ [By definition]}
\Rightarrow <\text{integer-part}>.5 \text{ [Production 2]}
\Rightarrow <\text{integer-part}>. <\text{digit}> \text{ [Production 5]}
\Rightarrow <\text{integer-part}>. <\text{digit-sequence}> \text{ [Production 4]}
\Rightarrow <\text{integer-part}>. <\text{fraction}> \text{ [Production 3]}
\Rightarrow <\text{real-number}> \text{ [Production 1]}
\]
A parse tree of “.5” generated by the CFG of Example 6.

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

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

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

\[
\text{Solution: Rewrite the grammar to make it unambiguous.}
\]

\[
<\text{Expr}>
\quad / \quad | \quad \backslash
\quad <\text{Expr}> \quad <\text{Op}> \quad <\text{Expr}>
\quad | \quad | \quad / \quad | \quad \backslash
\quad <\text{Id}> \quad + \quad <\text{Expr}> \quad <\text{Op}> \quad <\text{Expr}>
\quad | \quad | \quad | \quad | \quad |
\quad a \quad <\text{Id}> \quad - \quad <\text{Id}>
\quad | \quad | \quad |
\quad b \quad c
\]
CFG of Example 10 cannot handle “a + b – c” correctly.
⇒ Add a left recursive production.

\[
\begin{align*}
\langle Expr \rangle &::= \langle Expr \rangle \langle Op \rangle \langle Term \rangle \\
\langle Expr \rangle &::= \langle Term \rangle \\
\langle Term \rangle &::= (\langle Expr \rangle) | \langle Id \rangle \\
\langle Op \rangle &::= + | - | * | / | = \\
\langle Id \rangle &::= a | b | c
\end{align*}
\]
Now there is only one parse tree for “$a + b - c$”:
CFG of Example 10 cannot handle “a = b = c” correctly.
⇒ Add a right recursive production.

\[
< \text{Assign} > ::= < \text{Expr} > = < \text{Assign} > \\
< \text{Assign} > ::= < \text{Expr} > \\
< \text{Expr} > ::= < \text{Expr} > < \text{Op} > < \text{Term} > | < \text{Term} > \\
< \text{Term} > ::= ( < \text{Expr} > ) | < \text{Id} > \\
< \text{Op} > ::= + | - | * | / \\
< \text{Id} > ::= a | b | c
\]

**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”:

```
<Assign>
 / | \
/ | \<Expr> = <Assign>
| / | \<Term> <Expr> = <Assign>
| | |<Id> <Term> <Expr>
| | | a <Id> <Term>
| | | b <Id>
| | c
```
CFG of Example 10 cannot handle “a + b * c” correctly.
⇒ Add one nonterminal (plus appropriate productions) for each precedence level.

\[
\begin{align*}
< \text{Assign} > &::= < \text{Expr} > = < \text{Assign} > | < \text{Expr} > \\
< \text{Expr} > &::= < \text{Expr} > + < \text{Term} > \\
< \text{Expr} > &::= < \text{Expr} > - < \text{Term} > | < \text{Term} > \\
< \text{Term} > &::= < \text{Term} > * < \text{Factor} > \\
< \text{Term} > &::= < \text{Term} > / < \text{Factor} > | < \text{Factor} > \\
< \text{Factor} > &::= ( < \text{Expr} > ) | < \text{Id} > \\
< \text{Id} > &::= \text{a} \mid \text{b} \mid \text{c}
\end{align*}
\]
Now there is only one parse tree for “$a + b \times c$”:

```
<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:

\[
\begin{align*}
< S > &::= \text{if } < E > \text{ then } < S > \\
< S > &::= \text{if } < E > \text{ then } < S > \text{ else } < S >
\end{align*}
\]

How many parse trees can you find for the statement:

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]
Dangling-Else ..

```
if E_1 then S
  if E_2 then S_1 else S_2
  if E_1 then S
    if E_2 then S_1 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

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

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

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

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

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

1. \[ L = \{ wcw | w \text{ is a string of } a\text{'s or } b\text{'s } \}. \]

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.
✓ 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
   \[
   \langle A \rangle \ ::= \ x \\
   \langle A \rangle \ ::= \ x \langle B \rangle
   \]

2. Left-Regular Grammar
   \[
   \langle A \rangle \ ::= \ x \\
   \langle A \rangle \ ::= \ \langle B \rangle x
   \]

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>( x )</td>
<td>the single char ‘x’</td>
</tr>
<tr>
<td>( . )</td>
<td>any single char except the newline</td>
</tr>
<tr>
<td>([abc])</td>
<td>char class consisting of ‘a’, ‘b’, or ‘c’</td>
</tr>
<tr>
<td>([\wedge abc])</td>
<td>any char except ‘a’, ‘b’, ‘c’</td>
</tr>
<tr>
<td>( r^* )</td>
<td>repeat ”r” zero or more times</td>
</tr>
<tr>
<td>( r^+ )</td>
<td>repeat ”r” 1 or more times</td>
</tr>
<tr>
<td>( r? )</td>
<td>zero or 1 occurrence of ”r”</td>
</tr>
<tr>
<td>( rs )</td>
<td>concatenation of RE ”r” and RE ”s”</td>
</tr>
<tr>
<td>((r)s)</td>
<td>”r” is evaluated and concatenated with ”s”</td>
</tr>
<tr>
<td>( r \mid s )</td>
<td>RE ”r” or RE ”s”</td>
</tr>
<tr>
<td>( \backslash x )</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>
</tbody>
</table>
### RE Example 2: Regular Expression Notations

<table>
<thead>
<tr>
<th>RE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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:
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.