Chapter 2. Programming Language Syntax

- Unlike natural languages, ***computer languages*** must be ***precise***. Both their ***form (syntax)*** and ***meaning (semantics)*** must be ***specified without ambiguity***.

- To provide the needed degree of ***precision***, language designers and implementors ***use formal syntactic and semantic notation***.

(ex) digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
natural_number → non_zero_digit digit*

2.1 Specifying Syntax: Regular Expressions and Context-Free Grammars

- Formal specification of ***syntax*** requires ***a set of rules***.

- ***Tokens*** are the ***basic building blocks of programs***- the shortest string of characters ***with individual meaning***.

(ex) keywords, identifiers, symbols, and constants of various types

- ***To specify tokens***, we use the notation of "***regular expressions***".

A regular expression is one of the following:
(1) A ***character***
(2) The ***empty string***, denoted by Ɛ (or λ)
(3) Two regular expressions next to each other (: ***concatenation***)
(4) Two regular expressions separated by a vertical bar (|) (: ***or***)
(5) A regular expression followed by a ***Kleen star***, meaning the concatenation of zero or more strings generated by the expression in front of the star.

(Note) (3), (4) and (5) shows that tokens can be constructed from individual characters using just ***three kinds of formal rules***.

(ex) identifier in C++: (_ | a| … | z | A | … | Z) (_ | a| … | z | A | … | Z | 0 | … | 9)*

(ex) The syntax of numeric constants accepted by a simple hand-held calculator:

number → integer | real
integer → digit digit*
real → integer exponent | decimal (exponent | Ɛ)
decimal → digit* (. digit | digit .) digit*
exponent → (e | E) (+ | – | Ɛ) integer
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Character sets and formatting issues for tokens
  - (1) Case sensitivity
  - (2) Character sets (letters, digits, underscores, or additional characters)
  - (3) Limits on the maximum length of identifiers

- ***Regular expressions*** work well for defining ***tokens***. They are ***unable***, however, ***to specify "nested" constructs***, which are ***central to programming language***.

(ex) expr → id | number | - expr | (expr) | expr op expr
    op → + | - | * | /

(Note) → , production, variables (or nonterminals), starting symbol, terminals

(Note) Chomskey's Hierarchy

Recursively Enumerable Languages

Context Sensitive Languages

Context Free Languages

Regular Languages

- In a programming language, the ***terminals of the context-free grammar*** are the language's ***tokens***.

- The notation for context-free grammars is sometimes called ***Backus-Naur Form (BNF)***.
  (Note) ***EBNF*** with extra operators like +(***Kleen plus***), (,), …

- The ***parser dose not distinguish one identifier from another***. The ***semantic analyzer does*** distinguish them.

- A ***context-free grammar*** shows us ***how to generate a syntactically valid string of terminals***.

(ex) Use the above grammar to generate the string "***slope *x + intercept***"

expr → expr op expr → expr op id → expr + id → expr op expr + id → expr op id + id → expr * id + id

→ id * id + id

(Note) ***left-most derivation*** v.s. ***right-most derivation***

- →(“***drives***”) indicates that the right-hand side was obtained by using a production to replace some nonterminal in the left-hand side.

  (Note) Each string of symbols along the way is called a “***sentential form***”. The final sentential form consist of only terminals.

- →* means “drives after zero or more replacements”.

  (ex) expr →* id * id + id

- A grammar that allows the construction of ***more than one parse tree for some string of terminals*** is said to be “***ambiguous***”.
  requires some extra mechanism

- There are infinitely many context-free grammars for any given context-free language. Some grammars, however, are much ***more useful than others*** (i.e. ***unambiguity***, ***no use of useless symbols***,…).

2.2 Scanning

- Together, the ***scanner*** and ***parser*** for a programming language are responsible for discovering the ***syntactic structure*** of a program.

- “***Syntax analysis***”, is a necessary first step toward translating the program into an equivalent program in the target language.

- The scanner (1) groups input characters into tokens
  (2) removes comments
  (3) saves the text of interesting tokens like identifiers, strings, and numeric literals
  (4) tags tokens with line and column numbers

- See Figure 2.5 (p56) and Figure 2.6 (p57)

- It is more common to build a finite automaton automatically from a set of regular expressions.

(Step1) Converts the ***regular expressions into*** a nondeterministic finite automaton (***NFA***)
(Step2) Translates the ***NFA into*** an equivalent deterministic finite automaton (***DFA***)
(Step3) Generates a final ***DFA with the minimum possible number of states***.