Chapter 3
Syntax Analysis

"Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as *parsing*) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the *syntax tree* of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting *syntax errors*.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called *parser generation*. The notation we use for human manipulation is *context-free grammars*,1 which is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can in some cases be translated almost directly into recursive programs, but it is often more convenient to generate *stack automata*. These are similar to the finite automata used for lexical analysis but they can additionally use a stack, which allows counting and non-local matching of symbols.We shall see two ways of generating such automata. The first of these, LL(1), is relatively simple, but works only for a somewhat restricted class of grammars. The SLR construction,

which we present later, is more complex but accepts a wider class of grammars. Sadly, neither of these work for all context-free grammars. Tools that handle all context-free grammars exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars.

Role of the Parser

- In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.
- The parser returns any syntax error for the source language.

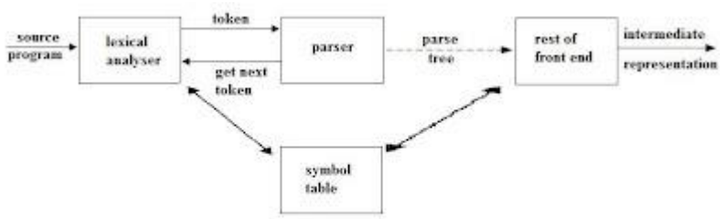

Fig 2.1 Position of parser in compiler model

- There are three general types' parsers for grammars.
- Universal parsing methods such as the *Cocke-Younger-Kasami algorithm* and *Earley's algorithm* can parse any grammar. These methods are too inefficient to use in production compilers.
- The methods commonly used in compilers are classified as either *top-down parsing* or *bottom-up parsing.*
- *Top-down parsers* build parse trees from the *top (root)* to the *bottom (leaves)*.
- *Bottom-up parsers* build parse trees from the *leaves* and work up to the *root*.
- In both case input to the parser is scanned from left to right, one symbol at a time.
- The output of the parser is some representation of the parse tree for the stream of tokens.
- There are number of tasks that might be conducted during parsing. Such as;
  - Collecting information about various tokens into the symbol table.
  - Performing type checking and other kinds of semantic analysis.
  - Generating intermediate code.
- *Syntax Error Handling:*
  - Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.
  - The program can contain errors at many different levels. e.g.,
    - *Lexical* – such as misspelling an identifier, keyword, or operator.
    - *Syntax* – such as an arithmetic expression with unbalanced parenthesis.
    - *Semantic* – such as an operator applied to an incompatible operand.
    - *Logical* – such as an infinitely recursive call.
- Much of the error detection and recovery in a compiler is centered on the syntax analysis phase.
  - One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
  - Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.
- The error handler in a parser has simple goals:

- It should the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

· *Error-Recovery Strategies:*

- There are many different general strategies that a parser can employ to recover from a syntactic error.

§ Panic mode
§ Phrase level
§ Error production
§ Global correction

- *Panic mode:*

§ This is used by most parsing methods.
§ On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens (delimiters; such as; semicolon or end) is found.
§ Panic mode correction often skips a considerable amount of input without checking it for additional errors.
§ It is simple.

- *Phrase-level recovery:*

§ On discovering an error; the parser may perform local correction on the remaining input; i.e., it may replace a prefix of the remaining input by some string that allows the parser to continue.
§ e.g., local correction would be to replace a comma by a semicolon, deleting an extraneous semicolon, or insert a missing semicolon.
§ Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

- *Error productions:*

§ If an error production is used by the parser, can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

- *Global correction:*

§ Given an incorrect input string x and grammar G, the algorithm will find a parse tree for a related string y, such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.

Posted 9th February 2012 by Arun Patrick
Labels: compiler design compiler design notes compiler notes role of parser

Context-Free Languages

Regular languages are inadequate for specifying all but the simplest aspects of programming language syntax. To specify more-complex languages such as

• $L = \{w \in \{a, b\}^* \mid w = a^n b^n \text{ for some } n\}$,
• $L = \{w \in \{(,)\}^* \mid w \text{ is a well-balanced string of parentheses}\}$ and
• the syntax of most programming languages,

we use context-free languages. In this section we define context-free grammars and languages, and their use in describing the syntax of programming languages. This section is intended to provide a foundation for the following sections on parsing and parser construction.

Note Like Section 3, this section contains mainly theoretical definitions; the lectures will cover examples and diagrams illustrating the theory.

4.1.1 Context-Free Grammars

Definition. A context-free grammar is a tuple G = (T,N, S, P) where:

• T is a finite nonempty set of (terminal) symbols (tokens),
• N is a finite nonempty set of (nonterminal) symbols (denoting phrase types) disjoint from T,
• $S \in N$ (the start symbol), and
• P is a set of (context-free) productions (denoting 'rules' for phrase types) of the form $A \to \alpha$ where $A \in N$ and $\alpha \in (T \cup N)^*$.

36

Notation. In what follows we use:

• a, b, c, . . . for members of T,
• A,B,C, . . . for members of N,
• . . . ,X, Y, Z for members of $T \cup N$,
• u, v,w, . . . for members of T^*, and
• α, β, γ, . . . for members of $(T \cup N)^*$.

Examples.

(1) G_1 = (T,N, S, P) where
• T = {a, b},
• N = {S} and
• P = {S → ab, S → aSb}.

(2) G_2 = (T,N, S, P) where
• T = {a, b},
• N = {S,X} and
• P = {S → X, S → aa, S → bb, S → aSa, S → bSb,X → a,X → b}.