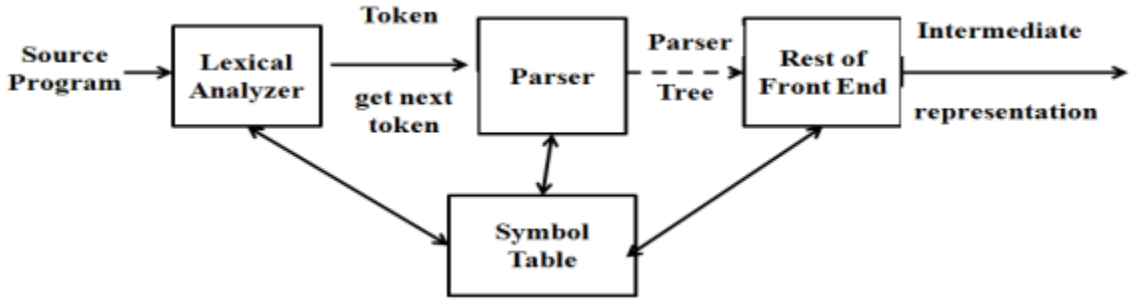
ROLE OF PARSER

- The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.



Role of the Parser:

- Parser builds the parse tree.
- Parser verifies the structure generated by the tokens based on the grammar (performs context free syntax analysis).
- Parser helps to construct intermediate code.
- Parser produces appropriate error messages.
- Parser performs error recovery.

Issues:

- Parser cannot detect errors such as:
- Variable re-declaration
- Variable initialization before use
- Data type mismatch for an operation

GRAMMARS

- Constructs that begin with keywords like while or int, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. Consider the grammar for expressions, which present more of challenge, because of the associativity and precedence of operators.
- In the following expression grammar E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Error Handling:

Programs can contain errors at many different levels.

For example:

- Lexical, such as misspelling an identifier, keyword or operators

- Syntactic, such as misplaced semicolons or extra or missing braces, a case statement without an enclosing switch statement.
- Semantic, such as type mismatches between operators and operands.
- Logical, such as an assignment operator = instead of the comparison operator ==, infinitely recursive call.

Functions of error handler:

- It should report 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:

Once an error is detected, the parser should recover from the error. The following recovery strategies are

- **Panic-mode**: On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or}, whose role in the source program is clear and unambiguous
- **Phrase-level**: When an error is identified, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. 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**: The common errors that might be encountered are anticipated and augment the grammar for the language at hand with productions that generate the erroneous constructs.
- **Global-correction**: A compiler need to make few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms 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. These methods are in general too costly to implement in terms of time and space.

CONTEXT-FREE GRAMMARS

Grammars are used to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable stmt to denote statements and variable expr to denote expressions, the production

stmt → **if** (expr) stmt **else** stmt

It specifies the structure of this form of conditional statement.

The Formal Definition of a Context-Free Grammar

A context-free grammar G is defined by the 4-tuple: G= (V, T, P S) where

1. V is a finite set of non-terminals (variable).
2. T is a finite set of terminals.
3. S is the start symbol (variable S∈V)
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Notational Conventions

These symbols are terminals:

- Lowercase letters early in the alphabet, such as a, b, c.
- Operator symbols such as +, *,- , / and so on.
- Punctuation symbols such as parentheses, comma, and so on.
- The digits 0 , 1 , . . . ,9.
- Boldface strings such as id or if, each of which represents a single terminal symbol

These symbols are non-terminals:

- Uppercase letters early in the alphabet, such as A, B, C.
- The letter S, which, when it appears, is usually the start symbol.
- Lowercase, italic names such as expr or stmt.

Example:

E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | **id**

Start symbol: E

Terminal: +, - , * , / , (,) , id

Non Terminal: E , T , F

Derivations:

`Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example: Consider the following grammar for arithmetic expressions:

E → E+E | E*E | (E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- -(id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called **sentinel** forms.

Types of derivations:

The two types of derivation are:

- Left most derivation: In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- Right most derivation: In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement. Rightmost derivations are sometimes called canonical derivations.

Example: Given grammar G : E → E+E | E*E | (E) | - E | id
Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

Parse Trees and Derivations:

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

Ambiguity:

- A grammar produce more than one parse tree for some sentence is said to be ambiguous. A grammar G is said to be ambiguous if it has more than one parse tree either in LMD or in RMD for at least one string.

Example: Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

Leftmost Derivations -1	***Leftmost Derivations -2***
E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :