ASSIGNMENT NUMBER: B-05

TITLE : Yacc program to run syntatic analysis.

PROBLEM STATEMENT:

Working of a compiler with their phases specifically syntactic analysis
Basics of Java programming
Basics of CFG and regular expressions

OBJECTIVES:

- Analyze source code for variables
- Identify data type and syntax of a variable

OUTCOMES:
The students will be able to

- Create a grammar and match the incoming instructions to their respective syntaxes.

THEORY:

Lex recognizes regular expressions, whereas YACC recognizes entire grammar. Lex divides the input stream into tokens, while YACC uses these tokens and groups them together logically.

The syntax of a YACC file:

```
%{
        declaration section
%}
        rules section
%%
        user defined functions
```

Declaration section:

Here, the definition section is same as that of Lex, where we can define all tokens and include header files. The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Grammar Rules in Yacc:

The rules section defines the context-free grammar to be accepted by the function Yacc generates, and associates with those rules C-language actions and additional precedence information. The grammar is described below, and a formal definition follows.

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

A : BODY ;

The symbol A represents a non-terminal name, and BODY represents a sequence of zero or

more names, literals, and semantic actions that can then be followed by optional precedence rules. Only the names and literals participate in the formation of the grammar; the semantic actions and precedence rules are used in other ways. The colon and the semicolon are Yacc punctuation.

If there are several successive grammar rules with the same left-hand side, the vertical bar '|' can be used to avoid rewriting the left-hand side; in this case the semicolon appears only after the last rule. The BODY part can be empty.

Programs Section

The programs section can include the definition of the lexical analyzer yylex(), and any other functions; for example, those used in the actions specified in the grammar rules. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it shall place them within "%{ ... %}" in the declarations section.

Interface to the Lexical Analyzer

The yylex() function is an integer-valued function that returns a token number representing the kind of token read. If there is a value associated with the token returned by yylex() (see the discussion of tag above), it shall be assigned to the external variable yylval.

Input descriptions:

The format of the grammar rules for Yacc is:

```
name    : names and 'single character's
        | alternatives
        ;
```

Yacc definitions:

%start line	means the whole input should match line
%union	lists all possible types for values associated with parts of the grammar and gives each a field-name
%type	gives an individual type for the values associated with each part of the grammar, using the field-names from the %union declaration
%token	declare each grammar rule used by YACC that is recognised by LEX and give type of value

Yacc does its work using parsing and hence it is also called a parser.

PARSING:

Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not. There are two main kinds of parsers in use, named for the way they build the parse trees:
Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse

to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

CONCLUSION : By performing this assignment we have implemented syntax analysis phase of compiler to recognize simple and compound sentences given in input file.