Assignment no.A4
Roll no.4202

TITLE:

Parser for sample language using YACC..

PROBLEM STATEMENT

Parser for sample language using YACC.

OBJECTIVE

- Be proficient on writing grammars to specify syntax
- Understand the theories behind different parsing strategies-their strengths and limitations
- Understand how the generation of parser can be automated

SOFTWARE AND HARDWARE REQUIREMENT

- PC with the configuration as 64-bit Fedora
- LEX, YACC utility.

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.

Running Lex and Yacc :

- Generate y.tab.c and y.tab.h files using 'yacc -d <filename.y>
- Generate a .c file using 'lex <filename.l>

- Link the two files together using gcc y.tab.c lex.yy.c –lfl

Mathematical Model :

Let S be the solution for the system.

S= { St , E , I , O , F, DD , NDD}
Where,

St is an initial state such that St={Fl,Fy | Fl is lex file and Fy is a yacc file}. At initial state, system consists of two files, each with three sections.

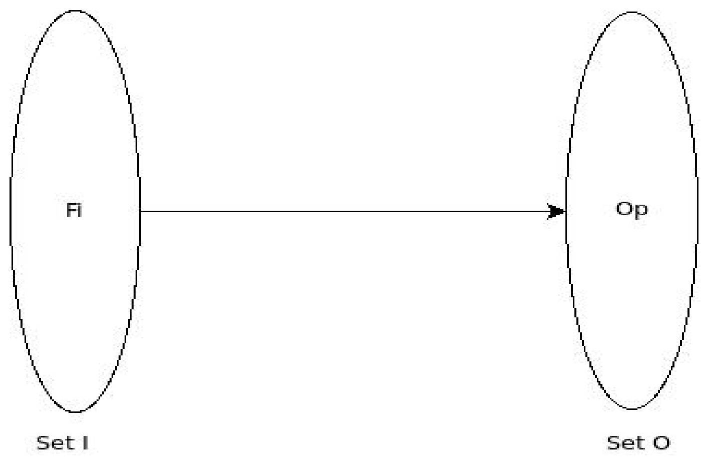
E is a End state.
E={Sc, Fc | Sc = success case which shows that input is syntactically correct and

Fc = failure case which mentions that syntax is syntactically incorrect}.

I= set of inputs
I={Fi | Fi is an input file which consists of High Level Language code}.

O={Op, Ot, Os, Oc | Op is program after preprocessing i.e. after removing comments, white spaces, Ot is lexeme-token table, Os is symbol table, Oc is console output which mention that syntax of statements of input program is correct or not}.

F=A set of Functions.
F={f1,f2,f3,f4 | f1,f2,f3:I → O, function f1 removes comments and white spaces from input program, function f2 generates the lexeme-token table from input program, function f3 generates the symbol table from input program, function f4 checks whether the statements of input program is as per the context free grammar mentioned in Fy or not}



Function f1: I → O

DD=Deterministic Data
DD={x | x is HLL's syntax which includes brackets, keywords, variables, functions definitions etc.}

NDD= Non-Deterministic Data
NDD={y | y is Semantic of the statements of Language}

State Transition Diagram:

$$\begin{tikzcd}
\text{St} \arrow[r, "\text{Fl, Fy}"] & \text{Q1} \arrow[rr, "\text{y.tab.c}"] \arrow[dr, "\text{Fl, y.tab.h}"'] & & \text{Q3} \arrow[r, "\text{a.out}"] & \text{E} \\
& & \text{Q2} \arrow[ur, "\text{lex.yy.c}"'] & &
\end{tikzcd}$$

Algorithm :

1. Write Lex code to separate out the tokens.
2. Write YACC code to check the syntax of the sentence.
3. Accept a input/sentence from the user
4. If input is as per syntax print 'accepted' else 'rejected'

Conclusion:

Hence we have successfully implemented a Parser for sample language using YACC.