Alok Bhawankar - PA06 - Panel 1 - 1032170126

School of CET

System Software and Compiler lab

TY BTech CSE

Assignment No. 8

Title: Parser for sample language using YACC.

Aim: Write a program using LEX and YACC to create Parser for sample l

language. (Design Calculator).

Objective:

1. To understand Yacc Tool.
2. To study how to use Yacc tool for implementing Parser.
3. To understand the compilation and execution of *.y file.

Theory: Write in brief for following:

1. Introduction to Yacc –

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.
YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

2. Study of *.y file(specification of y file)

Input File:
YACC input file is divided in three parts.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Input File: Definition Part:
The definition part includes information about the tokens used in the syntax definition:
%token NUMBER

- %token ID
- Yacc automatically assigns numbers for tokens, but it can be overridden by %token NUMBER 621
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.
- It can also include the specification of the starting symbol in the grammar: %start nonterminal

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:
  #include "lex.yy.c"
- YACC input file generally finishes with: .y

Output Files:
The output of YACC is a file named y.tab.c
If it contains the main() definition, it must be compiled to be executable.
Otherwise, the code can be an external function definition for the function int yyparse()

If called with the –d option in the command line, Yacc produces as output a header file y.tab.h with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).

If called with the –v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

3. Description of Each Section of *.y file with example

Input File: Definition Part:
The definition part includes information about the tokens used in the syntax definition:
%token NUMBER

- %token ID
- Yacc automatically assigns numbers for tokens, but it can be overridden by %token NUMBER 621
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.
- It can also include the specification of the starting symbol in the grammar: %start nonterminal

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:
The auxiliary routines part is only C code.
It includes function definitions for every function needed in rules part.
It can also contain the main() function definition if the parser is going to be run as a program.
The main() function must call the function yyparse().

4. Compilation and Execution Process-

For Compiling YACC Program:

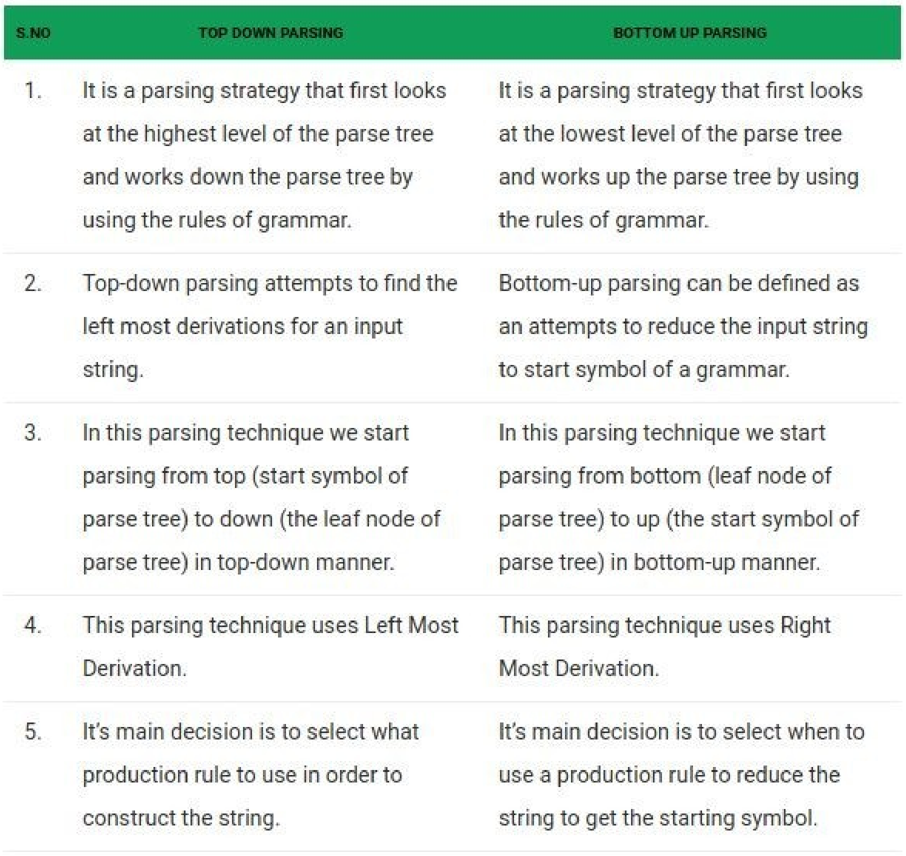
1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. flex file.l
4. yacc -d file.y
5. gcc lex.yy.c y.tab.h -ll
6. ./a.out

Input: Source specification (*.y) file for arithmetic expression statements.

Output: statement is grammatically correct or not.

FAQs:

1. Differentiate between top down and bottom up parsers.



S.NO	TOP DOWN PARSING	BOTTOM UP PARSING
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

2. Explain working of shift-reduce parser.

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.

This parser requires some data structures i.e.

A input buffer for storing the input string.

A stack for storing and accessing the production rules.

Basic Operations:

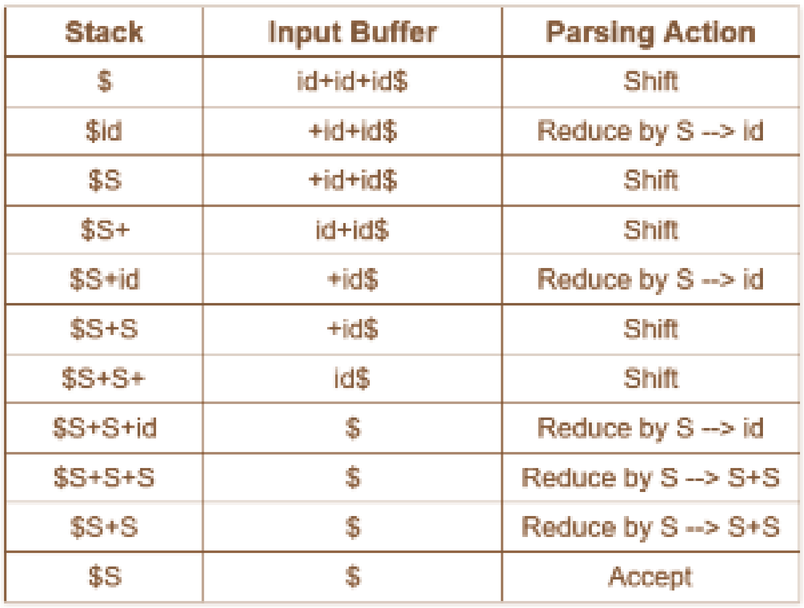
- Shift: This involves moving of symbols from input buffer onto the stack.

- Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
- Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.
- Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example 1 – Consider the grammar

S –> S + S
S –> S * S
S –> id

Perform Shift Reduce parsing for input string "id + id + id".



Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce by S --> id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce by S --> id
$S+S	+id$	Shift
$S+S+	id$	Shift
$S+S+id	$	Reduce by S --> id
$S+S+S	$	Reduce by S --> S+S
$S+S	$	Reduce by S --> S+S
$S	$	Accept

3. Explain how communication between LEX and YACC is carried out.

yacc generates parsers, programs that analyze input to insure that it is syntactically correct.
lex and yacc often work well together for developing compilers.
As noted, a program uses the lex-generated scanner by repeatedly calling the function yylex(). This name is convenient because a yacc-generated parser calls its lexical analyzer with this name.
To use lex to create the lexical analyzer for a compiler, end each lex action with the statement return token, where token is a defined term with an integer value.

The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, called yyparse() by yacc, then resumes control and makes another call to the lexical analyzer to get another token.
To use yacc to generate your parser, insert the following statement in the definitions section of your lex source:

```
#include "y.tab.h"
```

The file y.tab.h, which is created when yacc is invoked with the -d option, provides #define statements that associate token names such as BEGIN and END with the integers of significance to the generated parser.

When using lex with yacc, either can be run first. The following command generates a parser in the file y.tab.c:

```
$ yacc d grammar.y
```

As noted, the -d option creates the file y.tab.h, which contains the #define statements that associate the yacc-assigned integer token values with the user-defined token names. Now you can invoke lex with the following command:

```
$ lex lex.l
```

You can then compile and link the output files with the command:

```
$ cc lex.yy.c y.tab.c -ly -ll
```

The yacc library is loaded with the -ly option before the lex library with the -ll option to insure that the supplied main() calls the yacc parser.
Also, to use yacc with CC, especially when routines like yyback(), yywrap(), and yylook() in .l files are to be extern C functions, the command line must include the following.

```
$ CC -D__EXTERN_C........filename
```

4. How YACC resolves ambiguities within given grammar.

If a grammar is not LALR(1), Yacc will produce one or more multiply defined entries in the parsing table action function. These entries are reported as shift/reduce conflicts or reduce/reduce conflicts.
These conflicts can often be successfully resolved by rewriting the grammar or giving Yacc additional information about the associativities and precedence levels of operators in expressions or leaving them knowing the rules Yacc uses to deal with parsing action conflicts.
We can still use the ambiguous expression grammar above in a yacc specification if we resolve the associativities and relative precedence of the + and * operators by adding statements to the declarations section of the specification.
In general, yacc cannot deal with ambiguous grammars. If you give it an ambiguous grammar (or any non-LALR(1) grammar), it will parse a subset of the language of that grammar, which may or may not be what you want.