System Programming and Operating Systems Lab

ASSIGNMENT 1

Name:Shrirang Mhalgi
Roll No: 322008
Batch: B 1

1 Date of Completion:

02/01/2019

2 Aim:

Write a program using Lex Specification to implement Lexical analysis phase of compiler to generate tokens of subset of java program.

3 Objectives:

To implement basic language translator by using various needed data structures

4 Theory:

Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for ”lexical analyzer generator.” It is intended primarily for Unix-based systems. The code for Lex was originally developed by Eric Schmidt and Mike Lesk.

Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers. Lex can be used with a parser generator to perform lexical analysis. It is easy, for example, to interface Lex and Yacc, an open source program that generates code for the parser in the C programming language.Lex is proprietary but versions based on the original code are available as open source. These include a streamlined version called Flex, an acronym for ”fast lexical analyzer generator,” as well as components of OpenSolaris and Plan 9.

In programs with structured input, two tasks that occur over and over are dividing the input into meaningful units, and then discovering the relation- ship among the units. For a text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that don’t. For a C program, the units are variable names, constants, strings, operators, punctuation, and so forth. This divi- sion into units (which are usually called tokens) is known as lexical analy- sis, or lexing for short. Lex helps you by taking a set of descriptions of pos- sible tokens and producing a C routine, which we call a lexical analyzer, or a lexer, or a scanner for short, that can identify those tokens. The set of descriptions you give

to lex is called a lex specification.

The token descriptions that lex uses are known as regular expressions, extended versions of the familiar patterns used by the grep and egrep com- mands. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A lex lexer is almost always faster than a lexer that you might write in C by hand. As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as parsing and the list of rules that define the relationships that the program understands is a grammar. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules. A yacc parser is generally not as fast as a parser you could write by hand, but the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway. When a task involves dividing the input into units and establishing some relationship among those units, you should think of lex and yacc. (A search program is so simple that it doesn't need to do any parsing so it uses lex but doesn't need yacc. We'll see this again in Chapter 2, where we build several applications using lex but not yacc.) By now, we hope we've whetted your appetite for more details. We do not intend for this chapter to be a complete tutorial on lex and yacc, but rather a gentle introduction to their use.

Running LEX:

>lex filename.l
>gcc lex.yy.c -lfl
>./a.out

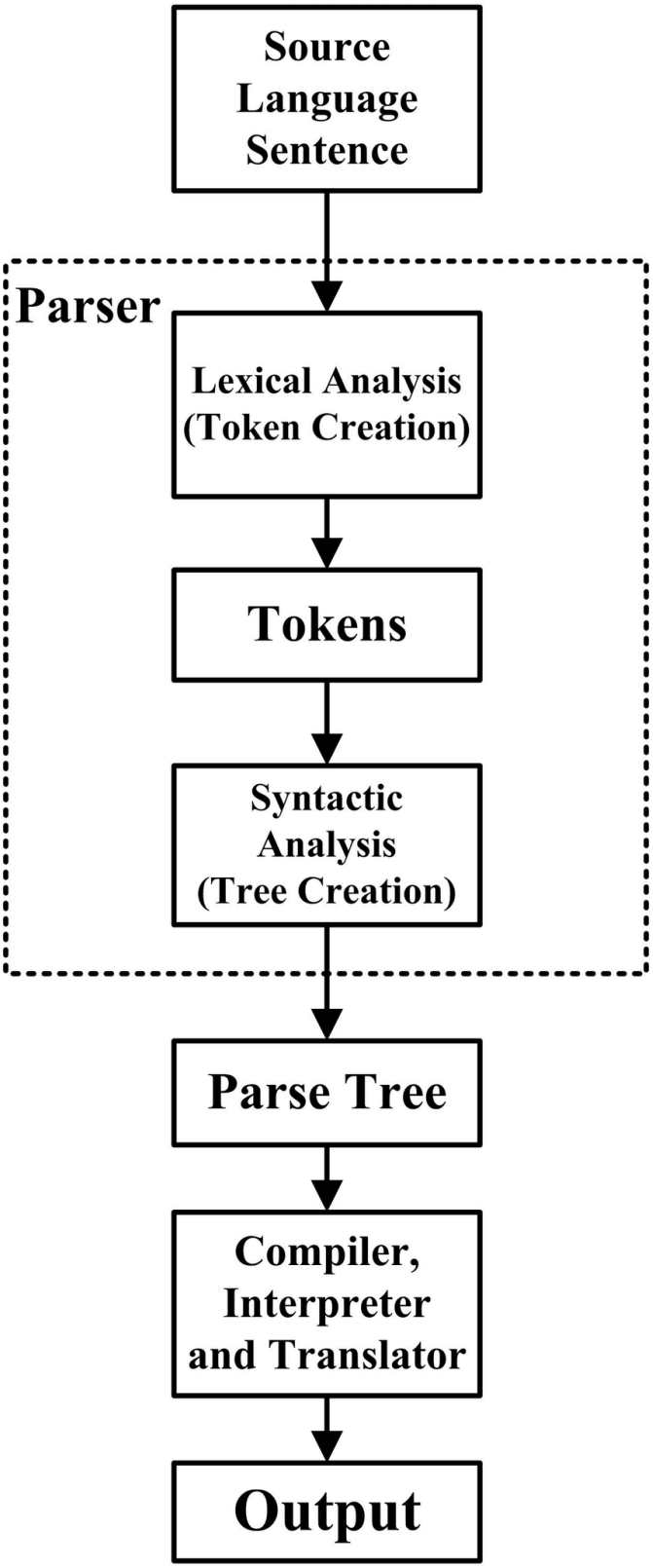
Lex translates the lex specification into a C source file called 1ex.yy.c which we compiled and linked with the lex library -11. We then execute the resulting program to check that it works as we expect.

You first have to go to the directory which the file filename.l is in using cd. Then using lex wordcount.l will make the file lex.yy.c. To the run the program you need compile it with a c compiler such as gcc. With gcc you can compile it using gcc -lfl lex.yy.c. This will create a.out which can be run using ./a.out.

5 Algorithm:

1. Start
2. Open a file file.c in read and include the yylex() tool for input scanning.
3. Define the alphabets and numbers.
4. Print the preprocessor, function, keyword using yytext.lex tool.
5. Print the relational, assignment and all the operator using yytext() tool.
6. Also scan and print where the loop ends and begins.
7. Use yywrap() to enter an error.
8. End

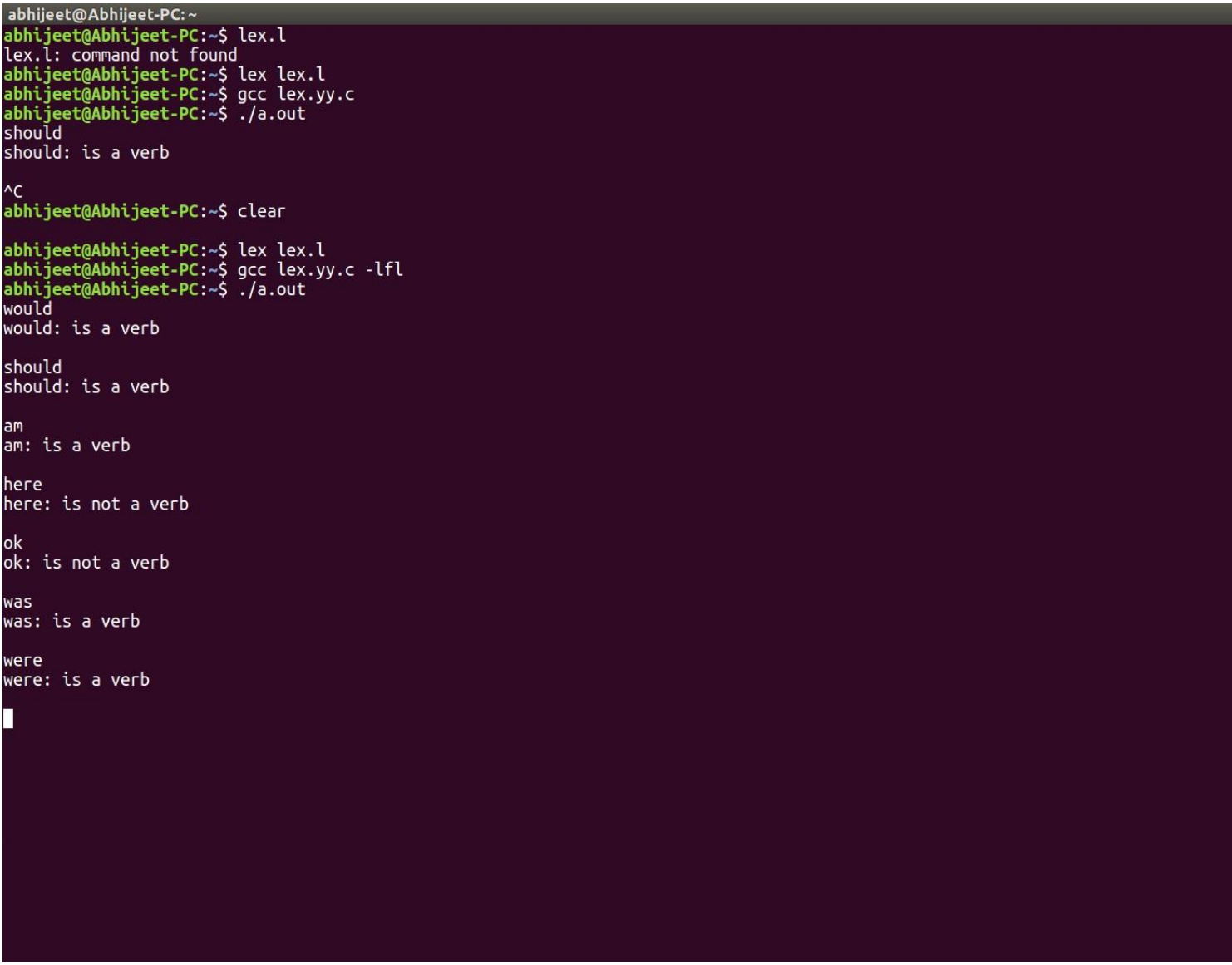
6 Flowchart:


Source Language Sentence
Parser
Lexical Analysis (Token Creation)
Tokens
Syntactic Analysis (Tree Creation)
Parse Tree
Compiler, Interpreter and Translator
Output

7 Code:

```
%{
/*
 * this sample demonstrates (very) s-le recognition:
 * averbhota verb.
 */

%}
%%

[\t ]+          /* ignore whitespace */ ;


is I
am I
are I
were I
was I
be I
being I
 been I
do I
does I
did I
 will I
 would I
 should I
 can I could I
 has I
 have I
had I
go              { printf("%s: is a verb\n", yytext) ; }

[a-zA-Z]+    { printf("%s: is not a verb\nm, yytext); }
 . |\n           { ECHO; /* normal default anyway */ }
%%
main()
{
 yylex() ;
}
```

8 Output:



9 Conclusion:

In this assignment we learn in detail the concept of lex compiler and the generation of tokens of subset.