Introduction to Lex:

Lex is a program that automatically generates code for scanners.Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidtand described in 1975, is the standard lexical analyzer generator on many UNIX systems, and a tool exhibiting its behaviour is specified as part of the POSIX standard. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The UNIX utility lexparses a file of characters. It uses regular expression matching; typically, it is used to 'tokenize' the contents of the file. In that context, it is often used together with the yaccutility. However, there are many other applications possible.

First phase in the compiler development is tokenization. That is, the input or source Program is decomposed into tokens which are also known as lexeme's (thus this phase is Also called as lexical analysis). Each language will be having its own lexical rules. Only After extracting the tokens, syntactical analysis is carried out to test the validity of the expression in terms of that language specific grammatical rules. After this, actual transformation to machine language (via assembly language in the case of C language) takes place. Lex library is widely used for lexical analysis. However, in the recent years, under Free Software Foundation license, Flex (fast lexical analyzer) is distributed along with Gnu compiler package which can be effectively used for lexical analysis purpose. In compiler construction terminology, these SW which are used for lexical analysis are called as scanners. In addition to scanner development, Lex/Flex is also used for some other applications in system administration where text processing is needed.

Structure of lex file:

A lexfile looks like

Definition section

%%

Rules section

%%

User code section

The **Definition section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The **Rules section** associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The **User code section** contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Definition Section:

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions

There are three things that can go in the definitions section:

C code Any indented code between %{and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the

code segment.

Definitions A definition is very much like a #define cpp directive. For example
letter [a-zA-Z]

digit [0-9]

punct [,.:;!?]

nonblank [Нь \t]

These definitions can be used in the rules section: one could start a rule

{letter}+ {...

State definitions If a rule depends on context, it's possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given.

The definitions section can also contain abbreviations for regular expressions to be used in the rules section. The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them. A definition for an ``identifier" would often be placed in this section. Abbreviations must appear after the '%}' delimiter that ends your #include statements and variable declarations (if there are any). The abbreviation appears on the left of the line and its definition appears on the right, separated by one or more spaces. When abbreviations are used in rules, they must be enclosed within curly braces.

Rules section

Each rule consists of a pattern to be searched for in the input, followed on the same line by an action to be performed when the pattern is matched. Because lexical analyzers are often used in conjunction with parsers, as in programming language compilation and interpretation, the patterns can be said to define the classes of *tokens* that may be found in the input. The rules section has a number of pattern-action pairs. The patterns are regular expressions and the actions are either a single C command, or a sequence enclosed in braces. If more than one rule matches the input, the longer match is taken. If two matches are the same length, the earlier one in the list is taken.

Matched text

When a rule matches part of the input, the matched text is available to the programmer as a variable char* yytext of length int yyleng.

To extend the example from the introduction to be able to count words, we would write

```
%{
int charcount=0,linecount=0,wordcount=0;
%}
letter [Нь \t\n]
%%
{letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

State

If the application of a rule depends on context, there are a couple of ways of dealing with this. We distinguish between ‘left state’ and ‘right state’, basically letting a rule depend on what comes before or after the matched token.

See section 8.1 for a good example of the use of state.

1.Left state:

A rule can be prefixed as

<STATE>(some pattern) {...

meaning that the rule will only be evaluated if the specified state holds. Switching between states are done in the action part of the rule:

<STATE>(some pattern) {some action; BEGIN OTHERSTATE;}

where all state names have been defined with %s SOMESTATE statements, as described in section 3. The initial state of *lex* is INITIAL.

2.Right state:

It is also possible to let a rule depend on what follows the matched text. For instance, abc/de {some action} means 'match abc but only when followed by de. This is different from matching on abcde because the de tokens are still in the input stream, and they will be submitted to matching next. It is in fact possible to match on the longer string; in that case the command abcde {yyless(3); .....} pushes back everything after the first 3 characters. The difference with the slash approach is that now the right state tokens are in yytext so they can be inspected.

Regular Expression

Many UNIX utilities have regular expressions of some sort, but unfortunately, they don't all have the same power. In computing, a regular expression provides a concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for "regular expression" include **regex** and **regexp**

Some Examples of Regular Expression:

- **<char> ::=** a the character a
- **<char> ::=** "s" the string s, even if s contains metacharacters
- **<char> ::=** \a the character a when a is a metacharacter
- **<char> ::=** . any character except newline
- **<regExp> ::= [<char>+]** any of the character <char>
- **<regExp> ::= [<char1>-<char2>]** any character from <char1> to <char2>
- **<regExp> ::= [^<char>+]** any character except those <char>

- **<regExp> ::= <regExp1>*** zero or more repetitions of <regExp1>
- **<regExp> ::= <regExp1>+** one or more repetitions of <regExp1>
- **<regExp> ::= <regExp1>?** zero or one repetitions of <regExp1>
- **<regExp> ::= <regExp1>|<regExp2>** <regExp1> or <regExp2>
- **<regExp> ::= <regExp1><regExp2>** <regExp1> followed by <regExp2>
- **<regExp> ::= (<regExp1>)** same as <regExp1>
- **<regExp> ::= {<name>}** the named regular expression in the definitions part

User Code Section

If the *lex* program is used coupled to a yacc program, you obviously do not want a main program: that one will be in the yacc code. In that case, leave this section empty; thanks to some cleverness you will not get the default main if the compiled lex and yacc programs are linked together. The user code section is simply copied to lex.yy.c verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too. The user code section is simply copied to `lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

In the definitions and rules sections, any indented text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{' and '%}' removed). The `%{', and '%}" must appear unindented on lines by themselves.

In the rules section, any indented or %{ } text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well defined and it may well cause compiletime errors. In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

Introduction to Yacc:

Grammars are described by rules using a variant of the Backus Naur Form (BNF)

- Context-free grammars

LALR(1) parse table is generated automatically based on the rules. Actions are added to the rules and executed after each reduction.

Yacc generates the following thing:

- Tables – according to the grammar rules.
- Driver routines – in C programming language.
- y.output – a report file.

Structure of Yacc file:

A Yacc input file consists of three sections:

- Definition
- Rules
- User code

Separate by %%
Similar to lex (actually, lex imitates yacc.)

The **Definition section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
The **Rules section** associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
The **User code section** contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

Definition Section:

There are three things that can go in the definitions section:

C code Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions The definitions section of a lex file was concerned with characters; in yacc this is tokens. These token definitions are written to a .h file when yacc compiles this file.

Associativity rules These handle associativity and priority of operators.

Lex Interaction with yacc:

Conceptually, lex parses a file of characters and outputs a stream of tokens; yacc accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.

If your lex program is supplying a tokenizer, the yacc program will repeatedly call the yylex routine. The lex rules will probably function by calling return everytime they have parsed a token. We will now see the way lex returns information in such a way that yacc can use it for parsing.

The shared header file of return codes

If lex is to return tokens that yacc will process, they have to agree on what tokens there are. This is done as follows.

- The yacc file will have token definitions

%token NUMBER

in the definitions section.

- When the yacc file is translated with yacc -d, a header file y.tab.h is created

that has definitions like

#define NUMBER 258

This file can then be included in both the lex and yacc program.

- The *lex* file can then call return NUMBER, and the yacc program can match on

this token. The return codes that are defined from %TOKEN definitions typically start at around 258, so that single characters can simply be returned as their integer value:

```
/* in the lex program */
[0-9]+ {return NUMBER}
[-+*/] {return *yytext}
/* in the yacc program */
sum : TERMS '+' TERM
```

Return values

In the above, very sketchy example, lex only returned the information that there was a number, not the actual number. For this we need a further mechanism. In addition to specifying the return code, the *lex* parse can return a symbol that is put on top of the stack, so that can access it. This symbol is returned in the variable yylval. By default, this is defined as an int, so the *lex* program would have

```
extern int llval;

%%

[0-9]+ {llval=atoi(yytext); return NUMBER;}
```

If more than just integers need to be returned, the specifications in the yacc code become more complicated. Suppose we want to return double values, and integer indices in a table.

Rules section

The rules section contains the grammar of the language you want to parse. This looks like

```
name1
:   THING something OTHERTHING {action}

    | othersomething THING          {other action}

name2 : .....
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically defines coming from %token definitions in the yacc program or character values.

User Code Section

The minimal main program is

```
int main()

{

yyparse();

return 0;

}
```

Extensions to more ambitious programs should be self-evident.

In addition to the main program, the code section will usually also contain subroutines, to be used either in the yacc or the lex program.