# Assignment No. 03

**Problem Statement:** Write a program to recognize infix expression using LEX and YAAC.

**Objectives:**

1. To understand LEX and YACC Concepts
2. To implement LEX & YACC Program

**Software Requirement:**
**Operating System recommended** :- 64-bit Open source Linux or itsderivative

**Programming tools recommended**: -LEX tool / Eclipse IDE

**Hardware Requirement:**I3 and I5 machines

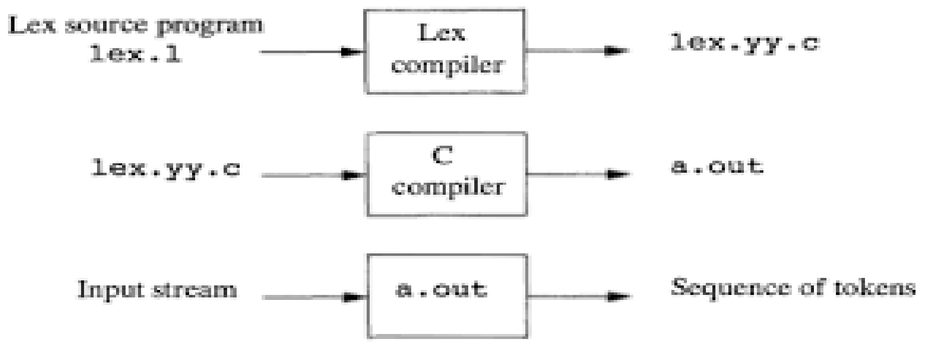
**Theory:**

**Lex:**

Lex stands for Lexical Analyzer. Lex is a tool for generating Scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular Expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it takes input one character at a time and continues until a pattern is matched, then lex performs the associated action (Which may include returning a token). If, on the other hand, no regular expressioncan be matched, further processing stops and Lex displays an error message.

Lex and C are tightly coupled. A .lex file (Files in lex have the extension .lex) is passed through the lex utility, and produces output files in C. These file(s) are coupled to produce an executable version of the lexical analyzer.

Lex turns the user"s expressions and actions into the host general –purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (calledinput in this memo) and perform the specified actions for each expression as it is detected.



**Overview of Lex Tool**

- **Regular Expression in Lex:-**

  A Regular expression is a pattern description using a meta language. An expression is made upof symbols. Normal symbols are characters and numbers, but there are other symbols that havespecial meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

  - **Defining regular expression in Lex:-**

| **Character** | **Meaning** |
|---|---|
| A-Z, 0-9,a-z | Character and numbers that form of the pattern. |
| . | Matches any character except \n. |
| - | Used to denote range. Example: A-Z implies all characters from A to Z. |
| [] | A character class. Matches any character in the brackets. If character is ^ then it indicates a negation pattern. Example: [abc] matches either of a,b and c. |
| * | Matches zero or more occurences of the preceiding pattern. |
| + | Matches one or more occurences of the preceiding pattern. |
| ? | Matches zero or one occurences of the preceidingpattern. |
| $ | Matches end of line as the last character of the pattern. |
| {} | Indicates how many times a pattern can be present. Example: A {1, 3} implies one or three occurences of A may be present. |
| \ | Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table. |
| ^ | Negation |
| \| | Logical OR between expressions. |
| "<some symbols>" | Literal meaning of characters. Meta characters hold. |
| / | Look ahead matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input. |
| () | Groups a series of regular expressions. |

  - **Examples of regular expressions**

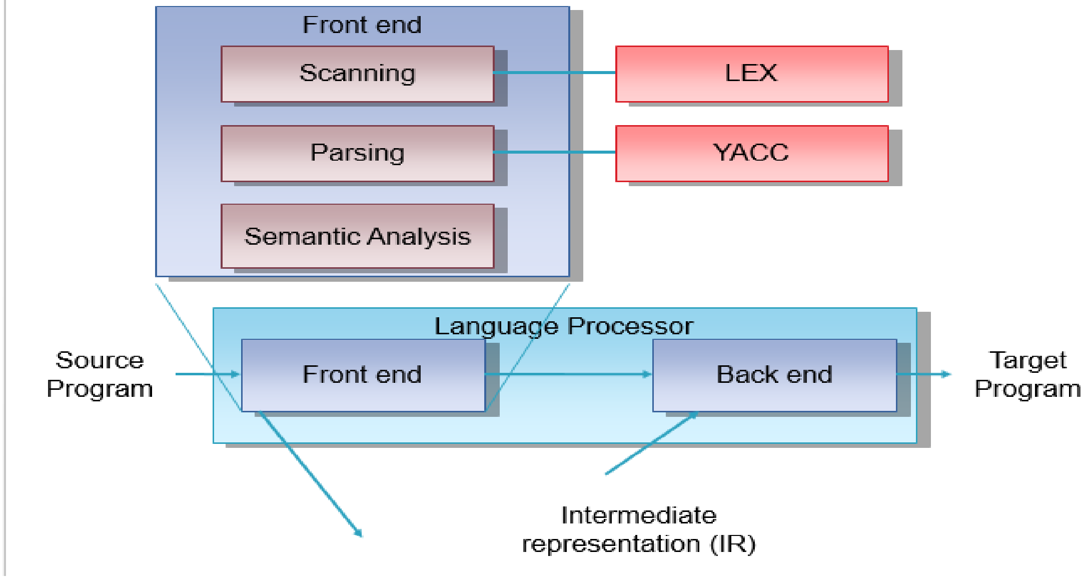
| **Regular expression** | **Meaning** |
|---|---|
| Joke[rs] | Matches either jokes or joker |
| A {1,2}shis+ | Matches Aashis, Ashis, Aashi, Ashi. |
| (A[b-e])+ | Matches zero or one occurrences of A followed by any character from b to e. |

Tokens in Lex are declared like variable names in C.Every token has an associated expression.(Examples of tokens and expression are given in the following table). Using the examples in our tables, we‟ll build a word-counting program. Our first task will be to show how tokens are declared.

- **Examples of token declaration**

| **Token** | **Associated expression** | **Meaning** |
|---|---|---|
| Number | ([0-9])+ | 1 or more occurences of a digit |
| Chars | [A-Za-z] | Any character |
| Blank | "" | A blank space |
| Word | (chars)+ | 1 or more occurences of chars |
| Variable | (chars)+(number)*(chars)*(number)* | |

**Design Architecture:**



➢ **Programming in Lex:-**

Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.
2. Run Lex over this file to generate C code for the scanner.
3. Compile and link the C code to produce the executable scanner.

**Note:** If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should beperformed.

A Lex program is divided into three sections: the first section has global C and Lex declaration, the second section has the patterns (coded in C), and the third section has supplement C functions. Main (), for example, would typically be founding the third section. These sections are delimited by %%.so, to get back to the word to the word-counting Lex program; let‟s look at the composition of the various program sections.

**Table 1: Special Characters**

| Pattern | Matches |
|---|---|
| `.` | any character except newline |
| `\.` | literal . |
| `\n` | Newline |
| `\t` | Tab |
| `^` | beginning of line |
| `$` | end of line |

**Table 2: Operators**

| Pattern | Matches |
|---|---|
| `?` | zero or one copy of the preceding expression |
| `*` | zero or more copies of the preceding expression |
| `+` | one or more copies of the preceding expression |
| `a\|b` | `a` or `b` (alternating) |
| `(ab)+` | one or more copies of `ab` (grouping) |
| `abc` | `Abc` |
| `abc*` | `ab abc abcc abccc ...` |
| `"abc*"` | literal `abc*` |
| `abc+` | `abc abcc abccc abcccc ...` |
| `a(bc)+` | `abc abcbc abcbcbc ...` |
| `a(bc)?` | `a abc` |

**Table 3: Character Class**

| Pattern | Matches |
|---|---|
| `[abc]` | one of: `a b C` |
| `[a-z]` | any letter a through z |
| `[a\-z]` | one of: `a - Z` |
| `[-az]` | one of: `- a Z` |
| `[A-Za-z0-9]+` | one or more alphanumeric characters |
| `[ \t\n]+` | Whitespace |
| `[^ab]` | anything except: `a b` |

| | |
|---|---|
| `[a^b]` | one of: `a ^ B` |
| **[a\|b]** | one of: **a \| b** |

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen ("- ") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

*... definitions ...*

**%%**

.
*.. rules ...*

**%%**

*... subroutines ...*

Input to Lex is divided into three sections with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

**%%**

Input is copied to output one character at a time. The first **%%** is always required as there must always be a rules section. However if we don‟t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively.

**Table 4: Lex Predefined Variables**

| Name | Function |
|---|---|
| **int yylex(void)** | call to invoke lexer, returns token |
| **char *yytext** | pointer to matched string |
| **yyleng** | length of matched string |
| **yylval** | value associated with token |
| **int yywrap(void)** | wrapup, return 1 if done, 0 if not done |
| **FILE *yyout** | output file |
| **FILE *yyin** | input file |
| **INITIAL** | initial start condition |
| **BEGIN condition** | switch start condition |
| **ECHO** | write matched string |

**YACC :**

**Yacc** (**Yet Another Compiler-Compiler**) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

Yacc is one of the automatic tools for generating the parser program. Basically Yacc is a LALR parser generator. The Yacc can report conflicts or ambiguities (if at all) in the form of error messages. LEX and Yacc work together to analyse the program syntactically.

Yacc is officially known as a “parser”. Its job is to analyze the structure of the input stream, and operate of the “big picture”. In the course of it’s normal work, the parser also verifies that the input is syntactically sound.

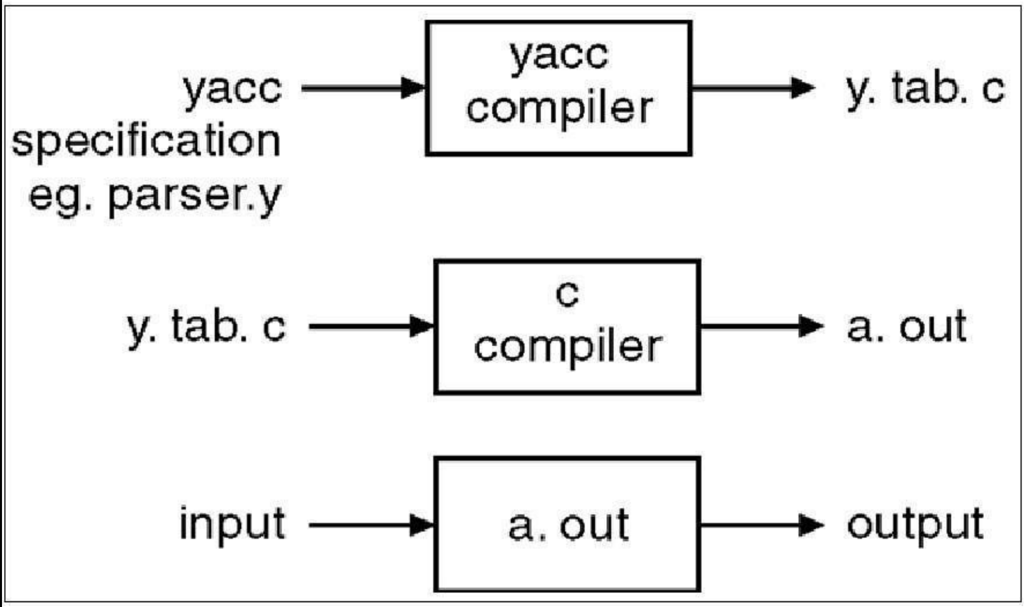


**Fig:-YACC: Parser Generator Model**

**Structure of a yacc file:**

A yacc file looks much like a lex file:

```
...definitions..
%%
...rules...
%%
...code...
```

Definitions As with lex, all code between %{ and %} is copied to the beginning of the resulting C file. Rules As with lex, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the 3 case with lex code. Thiscan be very Elaborate, but the main ingredient is the call to yyparse, the grammatical parse.

Input to yacc is divided into three sections. The definitions section consists of token declarations andC code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

**Infix Expression:**

Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between ( A + B )*C and A + ( B * C ) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. **Polish notation (prefix notation) –**
   It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e., +AB
2. **Reverse Polish notation(postfix notation) –**
   It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e., AB+

There are 3 levels of precedence for 5 binary operators as given below:

```
Highest: Exponentiation (^)
Next highest: Multiplication (*) and division (/)
Lowest: Addition (+) and Subtraction (-)
```

**For example –**

```
Infix notation: (A-B)*[C/(D+E)+F]
Post-fix notation: AB- CDE +/F +*
```

**Input:**

**Output:**

**Conclusion:**

Learn about Lex and yacc tools. Programming with lex and yacc and infix expression concepts.

**Frequently asked questions:**

1. What is Lex and Yacc?
2. What is Used of Lex and Yacc?
3. What is Infix, Postfix and Prefix expression?