**Practical No-2 (a)**

**Aim :**  Study of Lexical Analyser Generator- Lex and YACC tools

**Theory :**

A compiler or interpreter for a programming language is often decomposed into two parts:

* Read the source program and discover its structure.
* Process this structure, e.g. to generate the target program.

Lex and Yacc can generate program fragments that solve the first task.

The task of discovering the source structure again is decomposed into subtasks:

* Split the source file into tokens (Lex).
* Find the hierarchical structure of the program (Yacc).

**Introduction:-**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

**+-------+**

**Source -> | Lex | -> yylex**

**+-------+**

**+-------+**

**Input -> | yylex | -> Output**

**+-------+**

**Figure 1 An overview of Lex**

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

**%%**

**[ \t] + $ ;**

The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ``one or more ...''; and the $ indicates ``end of line,'' as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

%%

[ \t] + $ ;

[ \t] + printf(" ");

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs. Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc . Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

lexical grammar

rules rules

| |

v v

+---------+ +---------+

| Lex | | Yacc |

+---------+ +---------+

| |

v v

+---------+ +---------+

Input -> | yylex | -> | yyparse | -> Parsed input

+---------+ +---------+

**Figure 2 Lex with Yacc**

Yacc users will realize that the name yylex is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing. Lex generates a deterministic finite automaton from the regular expressions in the source . The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex. In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine. Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, Lex will recognize ab and leave the input pointer just before cd. . . Such backup is more costly than the processing of simpler languages.

**The general form of a Lex source file is:**

**{definitions}**

**%%**

**{rules}**

**%%**

**{user subroutines}**

The definitions section contains a combination of

1) Definitions, in the form ``name space translation''.

2) Included code, in the form ``space code''.

3) Included code, in the form

%{

code

%}

4) Start conditions, given in the form

%S name1 name2 ...

5) Character set tables, in the form

%T

number space character-string

...

%T

6) Changes to internal array sizes, in the form

%x nnn

where nnn is a decimal integer representing an array size and x selects the parameter as follows:

|  |  |
| --- | --- |
| Letter | Parameter |
| p | Positions |
| n | States |
| e | Tree nodes |
| a | Transitions |
| k | Packed character class |
| o | Output array size |

Lines in the rules section have the form ``expression action'' where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

|  |  |
| --- | --- |
| Operator | Meaning |
| x | The character "x" |
| "x" | An "x", even if x is an operator |
| [xy] | The character x or y |
| [x-z] | The characters x, y or z |
| [^x] | Any character but x |
| . | Any character but newline |
| ^x | An x at the beginning of a line |
| <y>x | An x when Lex is in start condition y |
| x$ | An x at the end of a line |
| x? | An optional x |
| x\* | 0,1,2, ... instances of x |
| x+ | 1,2,3, ... instances of x |
| x|y | An x or a y |
| (x) | An x |
| x/y | An x but only if followed by y |
| {xx} | The translation of xx from the definitions section |
| x{m,n} | m through n occurrences of x |

**Example**: consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

%%

int k;

[0-9]+ {

k = atoi(yytext);

if (k%7 == 0)

printf("%d", k+3);

else

printf("%d",k);

}

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

%%

int k;

-?[0-9]+ {

k = atoi(yytext);

printf("%d",

k%7 == 0 ? k+3 : k);

}

-?[0-9.]+ ECHO;

[A-Za-z][A-Za-z0-9]+ ECHO;

**Compilation /Running and Debugging the Solution**

1. Open Command prompt and switch to your working directory where you have stored your lex file (“.l“) and yacc file (“.y“)
2. Let your lex and yacc files be “hello.l” and “hello.y“. Now, follow the preceding steps to compile and run your program.
   1. For Compiling Lex file only:
      1. flex hello.l
      2. gcc lex.yy.c
   2. For Executing the Program

Double click on a.exe

**Testing the Solution**

Suppose the “myfile.txt” is as follows:-

**Yacc: Yet Another Compiler-Compiler**

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month\_name day ',' year ;

Here, date, month\_name, day, and year represent structures of interest in the input process; presumably, month\_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

month\_name : 'J' 'a' 'n' ;

month\_name : 'F' 'e' 'b' ;

. . .

month\_name : 'D' 'e' 'c' ;

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month\_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month\_name was seen; in this case, month\_name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

date : month '/' day '/' year ;

allowing 7 / 4 / 1776

as a synonym for July 4, 1776

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development. Yacc has been extensively used in numerous practical applications, including lint, the Portable C Compiler, and a system for typesetting mathematics

**Basic Specifications**

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

**declarations**

**%%**

**rules**

**%%**

**programs**

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

**%%**

**rules**

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /\* . . . \*/, as in C and PL/I. he rules section is made up of one or more grammar rules. A grammar rule has the form:

**A : BODY ;**

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``\_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

|  |  |
| --- | --- |
| '\n' | newline |
| '\r' | return |
| '\'' | single quote ``''' |
| '\\' | backslash ``\'' |
| '\t' | tab |
| '\b' | backspace |
| '\f' | form feed |
| '\xxx' | ``xxx'' in octal |

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

A : B C D ;

A : E F ;

A : G ;

can be given to Yacc as

A : B C D

| E F

| G

;

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

empty : ;

Names representing tokens must be declared; this is most simply done by writing

%token name1 name2 . . .

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

**Compilation /Running and Debugging the Solution**

* 1. For Compiling **Lex & Yacc** file both:
     1. flex hello.l
     2. bison -dy hello.y
     3. gcc lex.yy.c y.tab.c
  2. For **Executing** the Program

Double click on a.exe

**Program :**

%{

#include<stdio.h>

%}

%%

"if"|"else"|"while"|"do"|"switch"|"case"|"int" {printf("Keyword");}

[a-z|A-Z][a-z|0-9]\* {printf("Identifier");}

[0-9]\* {printf("Number");}

"!"|"@"|"\*"|"&"|"^"|"%"|"$"|"#"|";" {printf("Special Character");}

%%

int yywrap()

{

return 1;

}

main()

{

printf("Enter a string of data\n");

yylex();

}

**Installing procedure:**

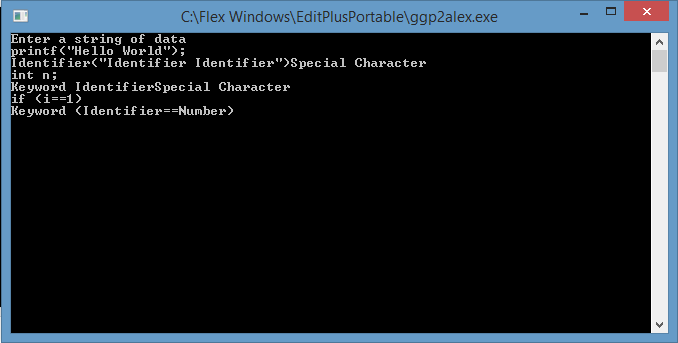
**How to install flex:**

1. Download flex-2.5.4a-1 from” https://sourceforge.net › Browse › Business & Enterprise › Office/Business” and install.
2. Download minGW from“https://sourceforge.net › Browse › Development › Compilers”  
   and install.
3. Add the Environment variable path for both flex and minGW’s bin folder.

**How to install Bison-2.4.1:**

1. Download Bison-2.4.1 from” https://sourceforge.net › Browse › Business & Enterprise › Office/Business” and install.
2. Download minGW from“https://sourceforge.net › Browse › Development › Compilers”  
    and install.
3. Add the Environment variable path for both flex and minGW’s bin folder.

**Output :**

****

**Conclusion :** The tools Flex and YACC have been studied and sample programs have been performed and tokens have been generated from the program.