MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LEX AND YACC

Outline

- Introduction on LEX
- RegEx
- Structure of LEX program
- Introduction to Yacc
- Structure of YACC program
- Integration of LEX and YACC
- References

Introduction to LEX

- LEX and YACC are tools designed for writers of compilers and interpreters.
- LEX is a tool for automatically generating a lexer or scanner given a lex specification (.l file)
- A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens.
- For example, consider breaking a text file up into individual words.
- In short, instead of writing a lexical analyzer from scratch, LEX will construct a lexical analyzer for you.
- Newer version of LEX is FLEX.

Lex file execution

1. LEX reads a specification of a scanner either from an input file and it generates as output a C source file "lex.yy.c".
2. "lex.yy.c" is compiled and linked with the lex library to produce an executable "a.out".
3. "a.out" analyzes its input stream and transforms it into a sequence of tokens.

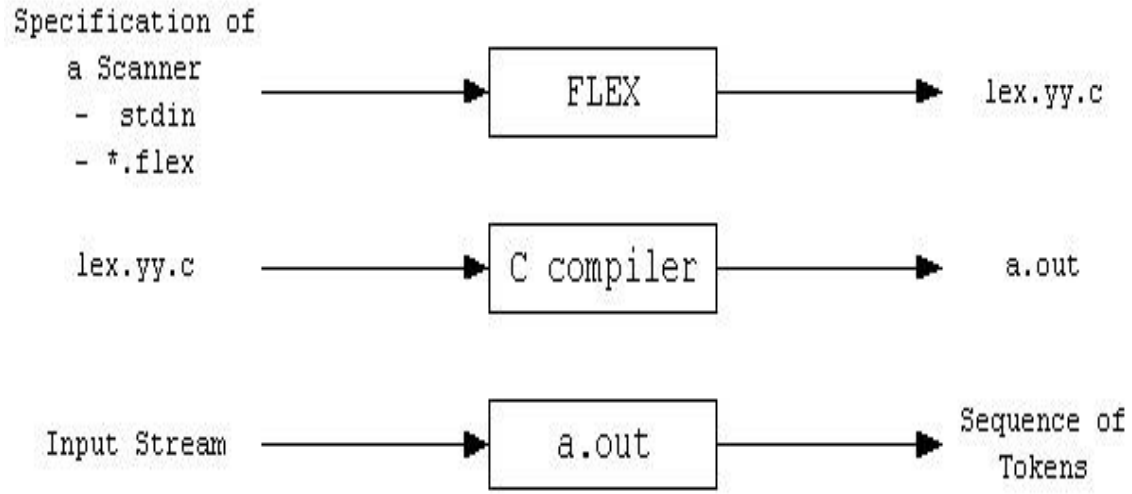


Fig. 1 Steps involved in generating Lexical Analyzer using Flex

Lex file execution[contd..]

- *.l is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine *yylex()* that uses the specification to recognize tokens.
- a.out is actually the scanner.

Regular expressions(RegEx)

- Scanners generally work by looking for patterns of characters in the input.
- For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter or an underscore followed by zero or more letters, underscores or digits, and the various operators are single characters or pairs of characters.
- A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp.
- A lex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions.
- A lex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.
- Lex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously.

Regular Expression Basics

. : matches any single character except \n

* : matches 0 or more instances of the preceding regular expression

+ : matches 1 or more instances of the preceding regular expression

? : matches 0 or 1 of the preceding regular expression

| : matches the preceding or following regular expression

[ ] : defines a character class

() : groups enclosed regular expression into a new regular expression

"…": matches everything within the " " literally

Regular Expression Basics[contd]

x|**y**	**x** or **y**
{**i**}	definition of **i**
x/**y**	**x**, only if followed by **y** (**y** not removed from input)
x{*m*,*n*}	*m* to *n* occurrences of **x**
∧ **x**	**x**, but only at beginning of line
x$	**x**, but only at end of line
"**s**"	exactly what is in the quotes (except for "\" and following character)

A regular expression finishes with a space, tab or newline

Meta-characters

- meta-characters (do not match themselves, because they are used in the preceding reg exps):
  - ( ) [ ] { } < > + / , ^ * | . \ " $? - %
- to match a meta-character, prefix with "\“[Escaping a character]
- to match a backslash, tab or newline, use \\, \t, or \n

Regular Expression Examples

- an integer: 12345
  - [1-9][0-9]*
- a word: cat
  - [a-zA-Z]+
- a (possibly) signed integer: 12345 or -12345
  - [-+]?[1-9][0-9]*
- a floating point number: 1.2345
  - [0-9]+"."[0-9]+

Lex Regular Expressions

Lex uses an extended form of regular expression:

(c: character, x,y: regular expressions, s: string, *m*,*n* integers and i: identifier).

1. c any character except meta-characters (see below)
2. [...]the list of enclosed chars (may be a range)
3. [∧...] the list of chars not enclosed
4. . any ASCII char except newline
5. xy concatenation of x and y
6. x* same as x*
7. x+ same as x^+ (i.e. x* but not $\in$)
8. x? an optional x (same as x+ $\in$)

Structure of a lex specification (.l file)

Filename.l → *.c is generated after running

%{

< C global variables, prototypes, comments > → This part will be embedded into *.c

%}

[DEFINITION SECTION] → substitutions, code; will be copied into *.c

%%

[RULES SECTION] → define how to scan and what action to take for each token

%%

< C auxiliary subroutines> → any user code. For example, a main function to call the scanning function yylex().

Definition section

- Declaration of variables and constants can be done in this section.
- This section introduces any initial C program code we want to get copied into the final program.
- This is especially important if, for example, we have header files that must be included for code later in the file to work.
- We surround the C code with the special delimiters "%{"and "%}."
- Lex copies the material between "%{" and "%}" directly to the generated C file, so we may write any valid C code here.
- The %% marks the end of this section.

Rules section

%%

[RULES SECTION]

<pattern> { <action to take when matched> }

<pattern> { <action to take when matched> }

…

%%

Patterns are specified by *regular expressions*.

For example:

%%

[A-Za-z]* { printf("this is a word"); }

%%

Rule section[contd..]

- Each rule is made up of two parts: a pattern and an action, separated by whitespace.
- The lexer that lex generates will execute the action when it recognizes the pattern.
- These patterns are UNIX style regular expressions.
- Each pattern is at the beginning of a line (since flex considers any line that starts with whitespace to be code to be copied into the generated C program.), followed by the C code to execute when the pattern matches.
- The C code can be one statement or possibly a multiline block in braces, { }.

User Subroutines section

- This is the final section which consists of any legal C code.
- This section has functions namely main( ) and *yywrap( )*.
- The function yylex( ) is defined in lex.yy.c file and is called from main( ).
- Unless the actions contain explicit return statements, yylex() won't return until it has processed the entire input.
- The function yywrap( ) is called when EOF is encountered. If this function returns 1, the parsing stops. If the function returns 0, then the scanner continues scanning.

Lex variables and special Functions

yytext - where text matched most recently is stored

yyleng -number of characters in text most recently matched

yylval-associated value of current token

yyin - Points to the input file.

yyout - Points to the output file.

yymore() - append next string matched to current contents of yytexts

yyless(n) - remove from yytext all but the first n characters

unput(c) - return character c to input stream

yywrap() - may be replaced by user. The yywrap method is called by the lexical analyser whenever it inputs an EOF as the first character when trying to match a regular expression

Lex program to count number of words,lines and characters(count.l)

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
 %%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }
%%
```

```
int main()
{
 yylex();
 printf("%d%d%d\n", lines, words, chars);
}
   int yywrap()
  {
return 1;
}
```

Steps to execute lex file

1. Type Flex program and save it using .l extension(as count.l in this ex)
2. Compile the flex code using

   $ flex filename.l ($ flex count.l)
3. Compile the generated C file using

   $ gcc lex.yy.c - o output

   This gives an executable output
4. Run the executable using **$ ./output**

Handling ambiguous patterns

Most flex programs are quite ambiguous, with multiple patterns that can match the same input.

Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

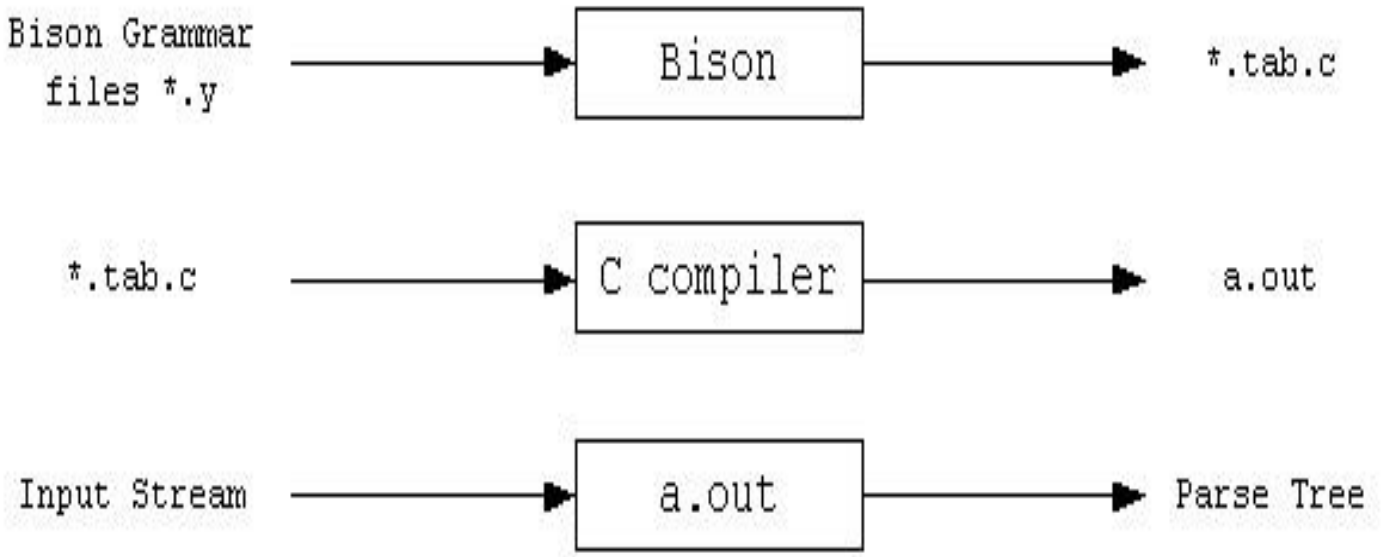
YACC

YET ANOTHER C COMPILER

Introduction to YACC

- A tool for automatically generating a parser given a grammar written in a yacc specification (.y file)
- A grammar specifies a set of production rules, which define a language.
- A production rule specifies a sequence of symbols, sentences, which are legal in the language.
- Newer version is called Bison.
- Bison is a general-purpose parser generator that converts a grammar description (Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar.
- The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

Working of Bison



Structure of a yacc specification (.y file)

Filename.y → *.c is generated after running

%{

< C global variables, prototypes, comments > → This part will be embedded into *.c

%}

[DEFINITION SECTION] → contains token declarations. Tokens are recognized in lexer.

%%

[PRODUCTION RULES SECTION] → define how to “understand” the input language, and what actions to take for each “sentence”.

%%

< C auxiliary subroutines> → any user code. For example, a main function to call the parser function yyparse()

Structure of Yacc[Contd]

- A bison specification has the same three-part structure as a flex specification. (Flex copied its structure from the earlier lex, which copied its structure from yacc, the predecessor of bison.)
- The first section, the definition section, handles control information for the parser and generally sets up the execution environment in which the parser will operate.
- The second section contains the rules for the parser.
- The third section is C code copied verbatim into the generated C program.

```
… definition section …
%%
… rules section …
%%
… user subroutines section …
```

Definition section

- The declarations here include C code to be copied to the beginning of the generated C parser, again enclosed in %{ and %}.
- Following that are %token token declarations, telling bison the names of the symbols in the parser that are tokens. By convention, tokens have uppercase names, although bison doesn't require it. For Example : %token IDENTIFIER,NUMBER.
- Any symbols not declared as tokens have to appear on the left side of at least one rule in the program. i.e these symbols will be treated as Non terminals.

The Production Rules Section

- The second section contains the rules in simplified BNF.
- Bison uses **:** in place of ➡ in each production, and since line boundaries are not significant, a semicolon marks the end of a rule.
- Again, like flex, the C action code goes in braces at the end of each rule.
- Bison creates the C program by plugging pieces into a standard skeleton file.
- The rules are compiled into arrays that represent the state machine that matches the input tokens.
- Each symbol in a bison rule has a value; the value of the target symbol (the one to the left of the colon) is called $$ in the action code,
- The values on the right are numbered $1, $2, and so forth, up to the number of symbols in the rule.
- The values of tokens are whatever was in yylval when the scanner returned the token; the values of other symbols are set in rules in the parser.

The Production Rules Section

Syntax

```
%%
production :  symbol1 symbol2 …   { action }
           |  symbol3 symbol4 …    { action }
           |  …

production:  symbol1 symbol2       { action }
%%
```

Example

```
%%
statement : expression  { printf (“ = %g\n”, $1); }
expression : expression ‘+’ expression { $$ = $1 + $3; }
           |  expression ‘-’ expression { $$ = $1 - $3; }
           |  NUMBER  { $$ = $1; }
%%
```

Precedence and Associativity

- Precedence and Associativity can be specified using following statements

%right ‘=‘

%left '-' '+'

%left '*' '/'

%right '^'

- Appears after definition and before rules section.
- It is optional

Write a Bison program to check the syntax of a simple expression involving operators +, -, * and /

exp.y

```
%{
        #include<stdio.h>
        #include<stdlib.h>
%}

 %token NUMBER ID NL
%left '+'
%left '*'

%%
stmt : exp NL  { printf("Valid Expression"); exit(0);}
        ;
exp : exp '+' term
      | term
term: term '*' factor
        |factor
factor:  ID
      | NUMBER
      ;
%%
int yyerror(char *msg)
{printf("Invalid Expression\n");
        exit(0);}

void main ()
{
        printf("Enter the expression\n");
        yyparse();
}
```

Flex Part

```
%{
        #include “exp.tab.h”        //exp.tab.h : here both flex and bison
%}

%%
[0-9]+ {return NUMBER; }
\n {return NL ;}
[a-zA-Z][a-zA-Z0-9_]* {return ID; }
. {return yytext[0]; }
%%
```

Steps to execute

1. Type Flex program and save it using .l extension.

2. Type the bison program and save it using .y extension.

3. Compile the bison code using

$ bison –d filename.y

The option **-d** Generates the file exp.tab.h with the #define statements that associate the yacc user-assigned “token codes" with the user-declared "token names." This association allows source files other than exp.tab.c to access the token codes.

This command generates two files

filename.tab.h and filename.tab.c

4.Compile the flex code using

$ flex filename.l

5. Compile the generated C file using

$ gcc lex.yy.c filename.tab.c - o output

This gives an executable output

6. Run the executable using **$ ./output**

References

Flex and Bison, O'reilly ,https://www.oreilly.com/library/view/flex-bison/9780596805418/

Download Link

http://web.iitd.ac.in/~sumeet/flex__bison.pdf