Introduction to LEX

What is lex?

- Lex is a tool for automatically generating a lexical analyzers or scanner given a lex specification (.l file)
- Lexical analyzers tokenize input streams.
- Tokens are the terminals of a language.
- Regular expressions define tokens.

Work Flow

Lex source program → Lex → lex.yy.c

lex.yy.c → C compiler→ a.out

Input → a.out →tokens

General Format

```
%{
< C global variables,
    prototypes, comments>
%}
[DEFINITION SECTION]
%%
[RULES SECTION]
%%
<C auxillary subroutines>
```

→ This part will be embedded into *.c

→ This defines how to scan and what action to take for each token

→ E.g. main function that calls the scanning function yylex()

General Format

- Input specification file is divided in three parts:
  - Definitions: Declarations
  - Rules: Token Descriptions and actions
  - Subroutines: User-Written code
- These three parts are separated by %%
- The first %% is always required as there must be a rules section
- If any rule is not specified, then by default everything on input will be copied to output
- Defaults for input and output are stdin and stdout

General Format

- The patterns are specified in the rules section.
- Each pattern must begin in column one.\
- This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern.
- The action may be a single C statement, or multiple C statements, enclosed in braces.
- Anything not starting in column one is copied as it is to the generated C file.

How to compile and run?

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

Sample Program: To read letters

```
%{
%}
letter      [A-Za-z]
%%
 /* match letters */
{letter}+ { printf("Letter Read");}
 %%
int main(void) {
     yylex();
     printf("Program ends\n");
     return 0;
}
```

Sample Program: To count lines, words and characters

```
%option noyywrap
%{
#include<stdio.h>
int lines=0, words=0;
%}
%%
[^ \t\n]+                words++;
\n                       {lines++; words++;}
%%
int main(){
    yyin= fopen("Noname.txt","r");
    yylex();
    printf("\n%d", lines);
    printf("\n%d", words);
    return 0;
}
```

Sample Program: To show use of REJECT

```
%option noyywrap
%{
    #include<stdio.h>
    int s=0;
%}

%%
she  {s++; REJECT;};
he   {s++;}
%%
int main(int argc, char *argv)
{
yylex();
printf("%d\n",s);
return 0;
}
```

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal "`a+b`" (C escapes still work)
`[]`	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-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
a|b	one of: a, b

Table 2: Pattern Matching Examples

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

Table 3: Lex Predefined Variables

Meta-Characters

- Meta-characters (do not match themselves)
  ( ) [ ] { } < > + / , ^ * | . \ " $? - %
- To match a meta-character, prefix with "\"
- To match a backslash, tab or new line, use \\, \t, or \n