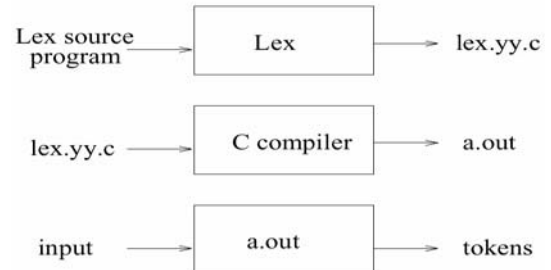


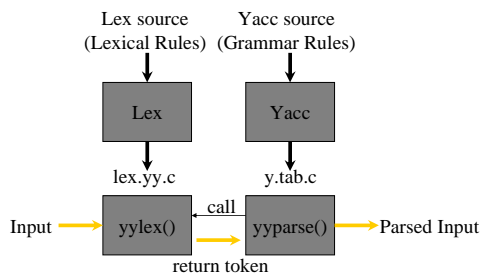
Lex Overview

- Lex is a tool for creating lexical analyzers.
- Lexical analyzers *tokenize* input streams.
- Tokens are the *terminals* of a language.
- Regular expressions define *tokens* .

Usage Paradigm of Lex



To Use Lex and Yacc Together



Lex Internals Mechanism

- Converts regular expressions into DFAs.
- DFAs are implemented as table driven state machines.

lex.yy.c : What it produces

```

# define YYTYPE unsigned char
struct yywork { YYTYPE verify, advance; } yycrank[] = {
0,0, 0,0, 1,3, 0,0,
0,0, 0,0, 0,0, 0,0,
...

struct yysvf yysvec[] = {
0, 0, 0,
yycrank+1, 0, yyvstop+1,
yycrank+3, yysvec+1, yyvstop+3,
yycrank+0, 0, yyvstop+5,
...

unsigned char yymatch[] = {
00 ,01 ,01 ,01 ,01 ,01 ,01 ,01 ,
01 ,01 ,012 ,01 ,01 ,01 ,01 ,01 ,
...
  
```

Running Lex

- To run lex on a source file, use the command:
lex source.l
- This produces the file *lex.yy.c* which is the C source for the lexical analyzer.
- To compile this, use:
cc -o prog -O lex.yy.c -ll

Versions and Reference Books

- AT&T lex, GNU flex, and Win32 version
- lex & yacc ,2/e by John R. Levine, Tony Mason & Doug Brown, O'Reilly
- Mastering Regular Expressions, by Jeffrey E.F. Friedl, O'Reilly

General Format of Lex Source

```
%{  
    C Declarations and includes  
%}  
  
<name> <regexp>  
<name> <regexp>  
...  
  
%%  
  
<regexp> <action>  
<regexp> <action>  
...  
  
%%  
  
User Subroutines (C code)
```

- Input specification file is in 3 parts
 - Declarations: Definitions
 - Rules: Token Descriptions and actions
 - Auxiliary Procedures: User-Written code
- Three parts are separated by %%
- Tips: ***The first part defines patterns, the third part defines actions, the second part puts together to express “If we see some pattern, then we do some action”.***

Regular Policy of Non-translated Source

- Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is **copied** into the generated program.
 - Any line which is not part of a Lex rule or action which begins with a **blank** or **tab**
 - Anything included between lines containing only **%{** and **%}**
 - Anything after the **second %%** delimiter

Position of Copied Source

- source input prior to the first %%
 - external to any function in the generated code
- after the first %% and prior to the second %%
 - appropriate place for declarations in the function generated by Lex which contains the actions
- after the second %%
 - after the Lex generated output

Regular Policy of Translated Source

- Various variables or tables whose name prefixed by yy
 - yyleng, yysvec[], yywork[]
- Various functions whose name prefixed by yy
 - yylex(), yymore(), yywarp(), yylex()...
- Various definition whose name are capital
 - BEGIN, INITIAL...

Default Rules and Actions

- The first and second part must exist, but may be empty, the third part and the second %% are optional.
- If the third part does not contain a main(), -ll will link a default main() which calls yylex() then exits.
- Unmatched patterns will perform a default action, which consists of copying the input to the output

Default Input and Output

- If you don't write your own main() to deal with the input and the output of yylex(), the default input of default main() is **stdin** and the default output of default main() is **stdout**.
 - stdin usually is to be keyboard input
 - stdout usually is to be screen output
 - cs20: %./a.out < inputfile > outputfile

Some Simple Lex Source Examples

- A minimum lex program:

```
%%
```

It only copies the input to the output unchanged.
- A trivial program to delete three spacing characters:

```
%%  
[ \t\n];
```
- Another trivial example:

```
%%  
[ \t]+$;
```

It deletes from the input all blanks or tabs at the ends of lines.

A General Lex Source Example

```
#{  
/*  
 * Example lex source file  
 * This first section contains necessary  
 * C declarations and includes  
 * to use throughout the lex specifications.  
 */  
#include <stdio.h>  
#}  
bin_digit [01]  
#%
```

```
{bin_digit}* {  
/* match all strings of 0's and 1's */  
/* Print out message with matching  
 * text  
 */  
printf("BINARY: %s\n", yytext);  
}  
([ab]*aa[ab]*bb[ab]*)|([ab]*bb[ab]*aa[ab]*) {  
/* match all strings over  
 * (a,b) containing aa and bb  
 */  
printf("AABB\n");  
}  
\n ; /* ignore newlines */
```

```
%%  
/*  
 * Now this is where you want your main  
 * program  
 */  
int main(int argc, char *argv[]) {  
/*  
 * call yylex to use the generated lexer  
 */  
yylex();  
/*  
 * make sure everything was printed  
 */  
fflush(yyout);  
exit(0);  
}
```

Token Definitions (Extended Regular Expression)

- Elementary Operations
 - single characters
 - except "\. \$ ^ [] - ? * + | () / { } % < >
 - concatenation (put characters together)
 - alternation (a|b|c)
 - [ab] == a|b
 - [a-k] == a|b|c|...|i|j|k
 - [a-z0-9] == any letter or digit
 - [^a] == any character but a

- Elementary Operations (cont.)
 - NOTE: . matches any character except the newline
 - * -- Kleene Closure
 - + -- Positive Closure
- Examples:
 - [0-9]+ "." [0-9]+
 - note: without the quotes it could be any character
 - [\t]+ -- is whitespace
 - (except CR).
 - There is a blank space character before the \t

- Special Characters:
 - . -- matches any single character (except newline)
 - " and \ -- quote the part as text
 - \t -- tab
 - \n -- newline
 - \b -- backspace
 - \" -- double quote
 - \\ -- \
 - ? -- this means the preceding was optional
 - ab? == a|ab
 - (ab)? == ab|ε

- Special Characters (cont.)
 - ^ -- means at the beginning of the line (unless it is inside of a [])
 - \$ means at the end of the line, same as ^\n
 - [^] -- means anything except
 - \"[^\"]*" is a double quoted string
 - {n,m} – m through n occurrences
 - a{1,3} is a or aa or aaa
 - {definition} – translation from definition
 - / -- matches only if followed by right part of /
 - 0/1 means the 0 of 01 but not 02 or 03 or ...
 - () -- grouping

Definitions

- NAME REG_EXPR
 - digs [0-9]+
 - integer {digs}
 - plainreal {digs} "." {digs}
 - expreal {digs} "." {digs} [Ee][+-]? {digs}
 - real {plainreal} {expreal}
- NAME must be a valid C identifier
- {NAME} is replaced by prior REG_EXPR

- The definitions can also contain variables and other declarations used by the Code generated by Lex.
 - These usually go at the start of this section, marked by %{ at the beginning and %} at the end or the line which begins with a blank or tab .
 - Includes usually go here.
 - It is usually convenient to maintain a line counter so that error messages can be keyed to the lines in which the errors are found.
 - %{
 - int linecount = 1;
 - %}

User Written Code

- The actions associated with any given token are normally specified using statements in C. But occasionally the actions are complicated enough that it is better to describe them with a function call, and define the function elsewhere.
- Definitions of this sort go in the last section of the Lex input.

More Example 1

```
int lengs[100];
%%
[a-z]+ lengs[yyleng]++;
. |
\n ;
%%
yywrap()
{
    int i;
    printf("Length No. words\n");
    for(i=0; i<100; i++)
        if (lengs[i] > 0)
            printf("%5d%10d\n",i,lengs[i]); return(1);
}
```

More Example 2

```
%{
int charCount=0, wordCount=0, lineCount=0;
%}
word [^ \t\n]+
%%
{word} {wordCount++; charCount += yyleng; }
[\n]   {charCount++; lineCount++; }
.      {charCount++; }
%%
main() {
    yylex();
    printf("Characters: %d Words: %d Lines:%d\n",
        charCount,wordCount,lineCount); }
```

Using yacc with lex

- yacc will call `yylex()` to get the token from the input so that each lex rule should end with:
 `return(token);`
 where the appropriate token value is returned.
- An easy way is placing the line:
 `#include "lex.yy.c"`
 in the last section of yacc input.