Compiler Design

Lex Syntax and Example

Lex is short for "lexical analysis". Lex takes an input file containing a set of lexical analysis rules or regular expressions. For output, Lex produces a C function which when invoked, finds the next match in the input stream.

1. **Format of lex input:**

   declarations
   %%
   token-rules (or) Translation rules
   %%
   Auxiliary procedures

2. **Declarations:**

a) string sets;	*name character-class*
b) standard C;	**%{** *-- c declarations --*
	%}

3. **Token rules:** *regular-expression* **{** *optional C-code* **}**

   a) if the expression includes a reference to a character class, enclose the class name in brackets { }
   b) regular expression operators;

** , +*	*--closure, positive closure*
*" " or *	*--protection of special chars*
|	*--or*
^	*--beginning-of-line anchor*
()	*--grouping*
$	*--end-of-line anchor*
?	*--zero or one*
.	*--any char (except \n)*
{ref}	*--reference to a named character class (a definition)*
[ ]	*--character class*
[^]	*--not-character class*

4. **Match rules:** Longest match is preferred. If two matches are equal length, the first match is preferred. Remember, lex partitions, it does not attempt to find nested matches. Once a character becomes part of a match, it is no longer considered for other matches.%{

5. **Built-in variables:** **yytext** -- ptr to the matching lexeme. (**char *yytext**;)
   yylen *-- length of matching lexeme (***yytext***). Note: some systems use* **yyleng**

6. **Aux Procedures:** C functions may be defined and called from the C-code of token rules or from other functions. Each lex file should also have a **yyerror()** function to be called when lex encounters an error condition.

7. **Execution of lex:** (to generate the *yylex()* function file and then compile a user program)

   (MS) **c:> flex** *rulefile* *(Linux)* **$ lex** *rulefile*
   flex produces *lexyy.c* **lex** produces *lex.yy.c*

   The produced .c file contains this function: **int yylex()**

8. Procedure to compile and execute LEX Program :

```
vi filename.l
lex filename.l
cc lex.yy.c -ll
./a.out
```

(or)

```
vi filename.l
lex filename.l
cc –o filename lex.yy.c –ll      (or)     cc lex.yy.c -ll -o filename –ll
./filename
```

Input given in a file and saved as Eg: file1.dat or   file1.txt

```
./filename < file1.dat
```

Compiling (f)lex

Create your lexical source in the file **filename.l** and then compile it with the command

- **flex filename.l**

The output of flex is a C source file **lex.yy.c** which you then must compile with the compiler of your choice

- **cc lex.yy.c –lfl**

f/lex can be used as a standalone program generator and does not have to be part of a larger compiler system as the diagram above shows.
lex.cc.y can be set to another filename within flex as can be the input file name (we use *scanner.specs*)
The key function **yylex()** can be generated and combined with other code instead of being connected to the standard executable **a.out**

`` `-lfl' ``
    library with which scanners must be linked.

`` `lex.yy.c' ``
    generated scanner (called `` `lexyy.c' `` on some systems).

`` `lex.yy.cc' ``
    generated C++ scanner class, when using `` `-+' ``.

`` `<FlexLexer.h>' ``
    header file defining the C++ scanner base class, `FlexLexer`, and its derived class, `yyFlexLexer`.

`` `flex.skl' ``
    skeleton scanner. This file is only used when building flex, not when flex executes.

`` `lex.backup' ``
    backing-up information for `` `-b' `` flag (called `` `lex.bck' `` on some systems).

Table 1: Pattern Matching Primitives

Meta character	**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
\x	the special character x, e.g. \$ or \? (prefix unary)
|	either the preceding expression or the following one (infix binary)

/	conditional: match the preceding expression only if followed by the following expression; useful for lookahead situations (binary)
"..."	exactly what's inside the quotes
$	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
[xyz]	any character from the string of characters; can use **-** for ranges of characters, e.g.[Ii] [0-9] [_ \n\t]
[^xyz]	any character not from the string of characters; can use **-** for ranges
{name}	a named regular expression reference, e.g. {digit}
{n,m}	minimum of n to a maximum of m repeats (postfix unary), e.g. {digit}{1,3}
( )	grouping

Table 2: Pattern Matching Examples

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 3: Lex Predefined Variables

Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yytext	Character string of matched lexeme
yylineno	Number of current input line
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 4: Pattern-Matching Modifiers

Name	**Function**
`/i`	Ignore alphabetic case (locale-aware)
`/x`	Ignore most whitespace in pattern and permit comments
`/g`	Global - match/substitute as often as possible
`/gc`	Don't reset search position on failed match
`/s`	Let `.` match newline; also, ignore deprecated `$*`
`/m`	Let `^` and `$` match next to embedded `\n`
`/o`	Compile pattern once only
`/e`	Righthand side of a `s///` is code to `eval`
`/ee`	Righthand side of a `s///` is a string to `eval`, then run as code, and its return value `eval` 'led again.

Regular expressions

```
delim  [ \t\n]
ws     {delim}+
letter [A-Za-z]
digit  [0-9]
id     {letter}({letter}|{digit})*
unum   {digits}+
rnum   (unum)(.{unum})?([Ee][+\-]?{unum})?
```

Translation rules

Translation rules are constructed as follows

r.e.$_1$ {action$_1$}
r.e.$_2$ {action$_2$}
....
r.e.$_n$ {action$_n$}

The actions$_i$ are C code to be carried out when the regular expression matches the input. Think event-driven programming.

For example:

```
{ws}        {/* nothing */}
[Ii][Ff]    {return(IF);}
{id}        {yylval = storeId(yytext,yyleng);
                 return(ID);}
{snum}      {yylval = storeNum(yytext,yyleng,atoi(yytext) ),INTEGER);
                 return(CON);}
{rnum}      {. . . . }
"<"         {yylval = LESS; return(RELOP);}
"<="        {yylval = LESSEQ; return(RELOP);}
```

Printing a Matched String

To find out what text matched an expression in the rules section of the specification file, you can include a C language **printf** subroutine call as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts the matched string into the external character (**char**) and wide character (**wchar_t**) arrays, called **yytext** and **yywtext,** respectively. For example, you can use the following rule to print the matched string:

```
[a-z]+            printf("%s",yytext);
```

The C language **printf** subroutine accepts a format argument and data to be printed. In this example the arguments to the **printf** subroutine have the following meanings:

%s	A symbol that converts the data to type string before printing
%S	A symbol that converts the data to wide character string (**wchar_t**) before printing
yytext	The name of the array containing the data to be printed
yywtext	The name of the array containing the multibyte type (**wchar_t**) data to be printed.

The **lex** command defines **ECHO**; as a special action to print out the contents of **yytext**. For example, the following two rules are equilvalent:

```
[a-z]+       ECHO;
[a-z]+       printf("%s",yytext);
```

You can change the representation of **yytext** by using either **%array** or **%pointer** in the definitions section of the **lex** specification file.

%array	Defines **yytext** as a null-terminated character array. This is the default action.
%pointer	Defines **yytext** as a pointer to a null-terminated character string.

Finding the Length of a Matched String

To find the number of characters that the lexical analyzer matched for a particular extended regular expression, use the **yyleng** or the **yywleng** external variables.

yyleng	Tracks the number of bytes that are matched.
yywleng	Tracks the number of wide characters in the matched string. Multibyte characters have a length greater than 1.

To count both the number of words and the number of characters in words in the input, use the following action:

```
[a-zA-Z]+       {words++;chars += yyleng;}
```

This action totals the number of characters in the words matched and puts that number in `chars`.

The following expression finds the last character in the string matched:

```
yytext[yyleng-1]
```

Matching Strings within Strings

The **lex** command partitions the input stream and does not search for all possible matches of each expression. Each character is accounted for only once. To override this choice and search for items that may overlap or include each other, use the**REJECT** directive. For example, to count all instance of she and he, including the instances of he that are included in she, use the following action:

```
she              {s++; REJECT;}
he               {h++}
\n               |
.                ;
```

After counting the occurrences of she, the **lex** command rejects the input string and then counts the occurrences of he. Because he does not include she, a **REJECT** action is not necessary on he.

Getting More Input

Normally, the next string from the input stream overwrites the current entry in the **yytext** array. If you use the **yymore** subroutine, the next string from the input stream is added to the end of the current entry in the **yytext** array.
For example, the following lexical analyser looks for strings:

```
%s instring
%%
<INITIAL>\"     {  /* start of string */
         BEGIN instring;
         yymore();
        }
<instring>\"    {  /* end of string */
         printf("matched %s\n", yytext);
         BEGIN INITIAL;
        }
<instring>.     {
         yymore();
        }
<instring>\n    {
         printf("Error, new line in string\n");
         BEGIN INITIAL;
        }
```

Even though a string may be recognized by matching several rules, repeated calls to the **yymore** subroutine ensure that the **yytext** array will contain the entire string.

Putting Characters Back

To return characters to the input stream, use the call:

```
yyless(n)
```

where n is the number of characters of the current string to keep. Characters in the string beyond this number are returned to the input stream. The **yyless** subroutine provides the same type of look-ahead that the / (slash) operator uses, but it allows more control over its usage.
Use the **yyless** subroutine to process text more than once. For example, when parsing a C language program, an expression such asx=-a is difficult to understand. Does it mean x *is equal to minus* a, or is it an older representation of x -= a which means *decrease* x *by the*

value of `a`? To treat this expression as `x` *is equal to minus* `a`, but print a warning message, use a rule such as:

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-1);
                ... action for = ...
                }
```

Input/Output Subroutines

The **lex** program allows a program to use the following input/output (I/O) subroutines:

input()	Returns the next input character.
output(c)	Writes the character c on the output.
unput(c)	Pushes the character c back onto the input stream to be read later by the **input** subroutine.
winput()	Returns the next multibyte input character.
woutput(C)	Writes the multibyte character C back onto the output stream.
wunput(C)	Pushes the multibyte character C back onto the input stream to be read by the **winput** subroutine.

lex provides these subroutines as macro definitions. The subroutines are coded in the **lex.yy.c** file. You can override them and provide other versions.

The **winput**, **wunput**, and **woutput** macros are defined to use the **yywinput**, **yywunput**, and **yywoutput** subroutines. For compatibility, the **yy** subroutines subsequently use the **input**, **unput**, and **output** subroutine to read, replace, and write the necessary number of bytes in a complete multibyte character.

These subroutines define the relationship between external files and internal characters. If you change the subroutines, change them all in the same way. They should follow these rules:

- All subroutines must use the same character set.
- The **input** subroutine must return a value of 0 to indicate end of file.
- Do not change the relationship of the **unput** subroutine to the **input** subroutine or the look-ahead functions will not work.

The **lex.yy.c** file allows the lexical analyzer to back up a maximum of 200 characters.

To read a file containing nulls, create a different version of the **input** subroutine. In the normal version of the **input** subroutine, the returned value of 0 (from the null characters) indicates the end of file and ends the input.

End-of-File Processing

When the lexical analyzer reaches the end of a file, it calls the **yywrap** library subroutine.

yywrap	Returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up at the end of input

However, if the lexical analyzer receives input from more than one source, change the **yywrap** subroutine. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates the program should continue processing.

You can also include code to print summary reports and tables when the lexical analyzer ends in a new version of the **yywrap** subroutine. The **yywrap** subroutine is the only way to force the **yylex** subroutine to recognize the end of input.

YYMORE & YYLESS:

lex Library

The **lex** library contains the following subroutines:

main()	Invokes the lexical analyser by calling the **yylex** subroutine.
yywrap()	Returns the value 1 when the end of input occurs.
yymore()	Appends the next matched string to the current value of the **yytext** array rather than replacing the contents of the **yytext** array.
yyless(int *n***)**	Retains *n* initial characters in the **yytext** array and returns the remaining characters to the input stream.
yyreject()	Allows the lexical analyser to match multiple rules for the same input string. (**yyreject** is called when the special action **REJECT** is used.)

Some of the **lex** subroutines can be substituted by user-supplied routines. For example, **lex** supports user-supplied versions of the **main** and **yywrap** subroutines. The library versions of these routines, provided as a base, are as follows:

main

```
#include <stdio.h>
#include <locale.h>
main() {
     setlocale(LC_ALL, "");
     yylex();
     exit(0);
}
```

yywrap

```
yywrap() {
       return(1);
}
```

yymore, **yyless**, and **yyreject** subroutines are available only through the **lex** library; however, these subroutines are required only when used in **lex** actions.

YACC :

For running Yacc program alone

vi filename.y
yacc filename.y
cc y.tab.c –ly
./a.out

Using lex and yacc together

Once again, there are a number of different ways you can build your lex and yacc specification files with a varying degree of customization. Here are a few notes:

- ***Do not*** include `%option main` in your lex specification file. This will cause a default main to be created only for lex and the yacc source will be ignored.
- Since you will not be using the default routines provided by `%option main`, you have to include the definition of `yywrap()` in your **lex specification** file. A simple example would look like this:

```
...definitions...
%%
...rules...
%%
...additional user code...

int yywrap() { return 1; }
```

- Since on some Linux systems, default libraries are not provided for yacc, we recommend that you define your own `main()` and `yyerror()` functions in your **yacc specification** file. So, the following would be a simple yacc specification file which does this:

```
...declarations...
%%
...rules...
%%
...additional user code...

main() {
  return yyparse();
}
int yyerror( char *s ) { fprintf( stderr, "%s\n", s); }
```

- For large projects, you could compile `lex.yy.c` and `y.tab.c` and link them afterwards. However, for simple cases (and most likely your projects), you could directly include `lex.yy.c` in the user code section of your **yacc specification**. Your yacc specification would then look something like the following:

```
...declarations...
%%
...rules...
%%
#include "lex.yy.c"

...additional user code...

main() {
  return yyparse();
}
int yyerror( char *s ) { fprintf( stderr, "%s\n", s); }
```

Procedure to compile and execute Lex and Yacc Program :
lex first.l
yacc second.y
cc y.yab.c –ly –ll

```
./a.out
```

- These custom definitions eliminate the need to link any external libraries. So, assuming your have the specification files **example.l** and **example.y**, the following commands would get you a nice executable which hopefully does what you want:

```
lex filename.l
yacc –d filename.y
cc lex.yy.c y.tab.c –o ex –lfl    or gcc y.yab.c –ll –ly     or  cc lex.yy.c y.tab.h - ll
./a.out  < file.txt
```

Input given in a file and saved as Eg: file1.dat or file1.txt

./filename < file1.dat

OR

```
lex example.l
yacc example.y
gcc -o example y.tab.c
./example
```

OR

```
lex example.l
yacc -d example.y
gcc lex.yy.c y.tab.c -ll -ly -o example
./example
```