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:**

(beginning in col. 1)	*declarations or definitions*
	%%
	token-rules (or) Translation rules
	%%
	Auxiliary procedures or subroutines

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. Example header file: tokens.h

```
#define NUM     1          // define constants used by lexyy.c
#define ID      2          // could be defined in the lex rule file
#define PLUS    3
#define MULT    4
#define ASGN    5
#define SEMI    6
```

7. Example lex file

```
D    [0-9]                          /* note these lines begin in col. 1 */
A    [a-zA-Z]
%{
#include “tokens.h”
%}
%%
```

```
{D}+                    return (NUM);           /* match integer numbers */
{A}({A}|{D})*           return (ID);            /* match identifiers */
"+"                     return (PLUS);          /* match the plus sign  (note protection) */
"*"                     return (MULT);    /* match the mult sign  (note protection again) */
:=                      return (ASGN);          /* match the assignment string */
;                       return (SEMI);          /* match the semi colon */
.                       ;                       /* ignore any unmatched chars */
%%

void yyerror ()                                 /* default action in case of error in yylex() */
  {   printf (" error\n");
      exit(0);
  }

void yywrap () { }              /* usually only needed for some Linux systems */
```

8. **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()**

9. **User program:**

(The above scanner file must be linked into the project)

```
#include <stdio.h>
#include "tokens.h"
int yylex ();                           // scanner prototype
extern char* yytext;

main ()
{     int n;
      while ( n = yylex() )                 // call scanner until it returns 0 for EOF
          printf (" %d  %s\n", n, yytext);   // output the token code and lexeme string
}
```

Table 1: Pattern Matching Primitives

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
\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
main()	Invokes the lexical analyser by calling the yylex subroutine.
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yytext	Character string of matched lexeme
yyleng	length of matched string
yywleng	Tracks the number of wide characters in the matched string. Multibyte characters have a length greater than 1.
yylval	value associated with token
yyval	local variable
yylineno	number of the current input line
int yywrap(void)	wrapup, called by lex when input is exhausted (EOF) and return 1 if done, 0 if not done
yymore()	Appends the next matched string to the current value of the yytext array rather than replacing the contents of the yytext array.
yyless(k)	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.)
yyparse()	It parses (builds the parse tree) of lexeme
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Regular expressions:

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

Translation rules

Translation rules are constructed as follows
r.e.1 {action1}
r.e.2 {action2}
....
r.e.n {actionn}

The actionsi 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);}
```

Compiling (f)lex
Create your lexical source in the file lex.l and then compile it with the command

- flex lex.l

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

- gcc 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).

```
%{
#include <stdio.h>
%}
%option noyywrap

%%
   [0-9]+  {
      printf("Saw an integer: %s\n", yytext);  }
.|\n    {   }
%%
int main(void)
{      yylex();
   return 0;
}
```