1 Lexical Analysis

What is Lexical Analysis?

Process of breaking input into tokens. Sometimes called “scanning” or “tokenizing.”

Lexical Analyzer scans the input stream and converts sequences of characters into tokens. Identifies tokens in input string (stream).

1.1 `lex`

- Lex (Lesk & Schmidt[Lesk75]) was developed in the early to mid-’70’s.
- `lex` reads a specification file containing regular expressions (rules and actions) to generate a C routine that performs lexical analysis.
- `lex` is a rule-based language. [Prolog later.]

1.2 `lex` File Format

```
definitions

%%

rules (regular expressions and associated actions)

%%

user routines (C functions)
```

`%%` is used to separate the sections.

1.2.1 definitions section

```
%{
C code
%}

%definition(s)
```

1.2.2 rules section

```
pattern action
pattern action
pattern action
```

1.2.3 user routines section

```

    C functions

```

1.2.4 Details

The first part, **definitions**, is optional. It can contain

- Lines controlling the dimensions of certain tables internal to lex
- Definitions for text replacements
- Global C code preceded by a line beginning with `%{` and followed by a line beginning with `%}`

The second part, **rules**, contains a table of patterns and actions. This part is line-oriented! A pattern starts at the beginning of a line and extends to the first non-escaped white space. An action is specified after an arbitrary amount of white space.

The third part, **user-defined functions**, and the separator preceding it are optional. This part contains C code (local functions) which are used in the **rules** part.

Note: Do not leave extra spaces and/or empty lines at the end of the `lex` specification file.
What about comments? C comments can be placed in the definition or user routines section.

1.3 Building an application with `lex`

General commands to create an application using `lex`.

```
lex lexFile.l
```

generates (outputs) a C file `lex.yy.c` and constructs a C function `yylex()`.

Compile and link with `lex` library.

```
cc -o scanner lex.yy.c -ll
```

1.4 Example 1

Note when `start` and `stop` commands are encountered in a file.

1.4.1 `lex` file: `example1.l`

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

%%
stop    printf( "Stop command received" );
start   printf( "Start command received" );
%%
```

`lex example1.l`
output: `lex.yy.c` (1500^+ lines)
`cc -o example1 lex.yy.c -ll`

1.4.2 example1.dat

```
start
stop
start
rest
stop
go
start start start
rest
stop  stop  stop
```

1.4.3 Output

```
% ./example1 < example1.dat

Start command received
Stop command received
Start command received
rest
Stop command received
go
Start command received Start command received Start command received
rest
Stop command received Stop command received Stop command received
```

1.5 Special `lex` Variables

yytext is an external variable that contains the matched string.

yyleng is an external variable that contains the length (the number of characters) of the matched string.

yylineno is an external variable that contains the number of the current input line (version dependent).

Study the `man` page for `lex` for more detail.

1.6 Example 2: Name Matching

Find a name and bracket it with `<` and `>` if it is encountered.

1.6.1 `nameMatch.l`

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

%%
Dexter|DeeDee  printf( "<%s>", yytext );
%%
```

1.6.2 Input: cartoon.dat

```
Animaniacs:  Yakko, Wakko, Dot
Bugs Bunny:  Bugs Bunny
Dexter's Laboratory: Dexter, DeeDee
Speed Racer:  Speed, ChimChim
Spongebob Squarepants:  Spongebob
```

1.6.3 Output

```
% ./example2 < cartoon.dat

Animaniacs:  Yakko, Wakko, Dot
Bugs Bunny:  Bugs Bunny
<Dexter>'s Laboratory: <Dexter>, <DeeDee>
Speed Racer:  Speed, ChimChim
Spongebob Squarepants:  Spongebob
```

1.7 Pattern Resolution Rules

`lex` patterns tend to be ambiguous. The following rules are used by `lex`:

1. chars are only matched once
2. longest match wins (action of longest match is used)
3. same length: first pattern wins
4. no pattern matches character(s) are printed

1.8 Example 3: Word Count (`wc`)

```
%{
/* wc.l
   lex file for word count
 */

#include <stdio.h>

int nL = 0;
int nW = 0;
int nC = 0;
%}
%%
[a-zA-Z]+  { nW++;  nC += yyleng; }
\n         { nL++;  nC++; }
.          { nC++; }
%%

int main()
{
   yylex();

   printf( "%d\n%d\n%d\n", nL, nW, nC );
}
```

1.8.1 Build and Execute

```
% lex wc.l
% cc -o wcount lex.yy.c -ll
% ./wcount < cartoon.dat
5
20
155
```

1.9 Example 4: Yet Another Word Count

```
%{
/* yawc.lex

   Yet another word count program.
   Derived from "Jumpstart your Yacc...and Lex too!"
   IBM developerWorks
   http://www-106.ibm.com/developerworks/linux/library/
      l-lex.html?dwzone=linux

   Bruce M. Bolden                     February 20, 2003
 */
int wordCount = 0;
%}

chars      [a-zA-Z\_\'\.\"\,\:\?]
numbers    ([0-9])+
delim      [" "\n\t]
whitespace {delim}+
words      {chars}+
%%

{words}      { wordCount++; }
{whitespace} { /* do nothing */ }
{numbers}    { wordCount++; }

%%

int main()
{
   yylex();

   printf( "words: %d\n", wordCount );

   return 0;
}

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

1.9.1 Build and Execute

```
% lex yawc.lex
% cc -o yawc lex.yy.c -ll
% ./yawc < cartoon.dat
words: 15
```

1.10 Example 5: Line Numbering

The following examples are derived from the line numbering examples in Schreiner and Friedman [S&FB85].

```
%{
/*  lineNum.l

    Line numbering
 */
%}

%%

\n    ECHO;
^.*$  printf( "%d\t%s", yylineno, yytext );
```

```
% lex lineNum.l
% cc -o lineNum lex.yy.c -ll
lineNum.l: In function ‘yylex’:
lineNum.l:11: ‘yylineno’ undeclared (first use in this function)
lineNum.l:11: (Each undeclared identifier is reported only once
lineNum.l:11: for each function it appears in.)
```

1.10.1 Blank lines not numbered

```
% lex lineNum1.l
% cc -o lineNum lex.yy.c -ll
% ./lineNum < lineNum1.l
0       %{
1       /*  lineNum1.l

3           Line numbering (modified version)
4           yylineno not defined in flex.
5        */

7       int lineNum = 0;
8       %}
```

```
10      %%

12      \n    { ECHO; lineNum++; }
13      ^.*$  printf( "%d\t%s", lineNum, yytext );
```

1.10.2 Blank lines numbered

```
% lex lineNum2.l
% cc -o lineNum lex.yy.c -ll
% ./lineNum < lineNum2.l
1       %{
2       /*  lineNum2.l
3
4           Line numbering --- blank lines also
5        */
6
7       int lineNum = 0;
8       %}
9
10      %%
11
12      ^.*\n  printf( "%d\t%s", ++lineNum, yytext );
```

1.10.3 `yylineno` fix—`lex` compatability

```
%{
/*  lineNum.l

    Line numbering
 */
%}

%option lex-compat

%%

\n    ECHO;
```

```
^.*$  printf( "%d\t%s", yylineno, yytext );
```

1.11 `lex` command line options

`lex` supports numerous command line options. For example
  -l turn on compatibility with the original AT&T lex.
  -p generates a performance report to stderr.

See the `man` page for `lex` for more command line option information.

1.12 Comparing Lexical Analyzers

Why use `lex`? Consider a handwritten lexer with `lex` description for a simple command language. The `lex` description is about one third the size of the C source file!

```
% wc lexer.[lc]
      64     183    1116 lexer.c
      26      71     499 lexer.l
```

Note: The source files are not displayed since they were displayed in class. The point is that `lex` is a very powerful tool and can save development time. Naive approach to keyword recognition:

```
%%
char   printf( "<b>char</b>" );
int    printf( "<b>int</b>" );
double printf( "<b>double</b>" );
if     printf( "<b>if</b>" );
else   printf( "<b>else</b>" );
/* <
   >
*/
\n  printf( "<br>\n" );
[ ] printf( "&nbsp" );
%%
```

Robust keyword recognition:

```
%{
#define MAX_NAME  32

struct keytable
{
    char  name[MAX_NAME];
    int   nOccurences;
};

typedef struct keytable KeyTable;

KeyTable keys[] =
{
   "char",   0,
   "int",    0,
   "double", 0,
   "if",     0
};
%}

%%
[A-Za-z][A-Za-z0-9]*
         { if( IsKeyword( yytext ) )
            {
                printf( "<b>%s</b>", yytext );
            }
            else
            {
                printf( "%s", yytext );
            }
         }
\n  printf( "<br>\n" );
[ ] printf( "&nbsp" );
%%

int IsKeyword( char* s )
{
    //  search keyword table
}
```

References

[Lesk75] M. Lesk, E. Schmidt, "Lex-A Lexical Analyzer Generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1975.

[LM&B90] Levine, Mason & Brown, *lex & yacc*, O'Reilly & Associates, 1990.

[S&FB85] Schreiner and Friedman, *Introduction to Compiler Construction with Unix*, Prentice-Hall, 1985.

[LnYP03] The Lex & Yacc Page, `http://dinosaur.compilertools.net/`