LEX:
LEXICAL ANALYZER GENERATOR

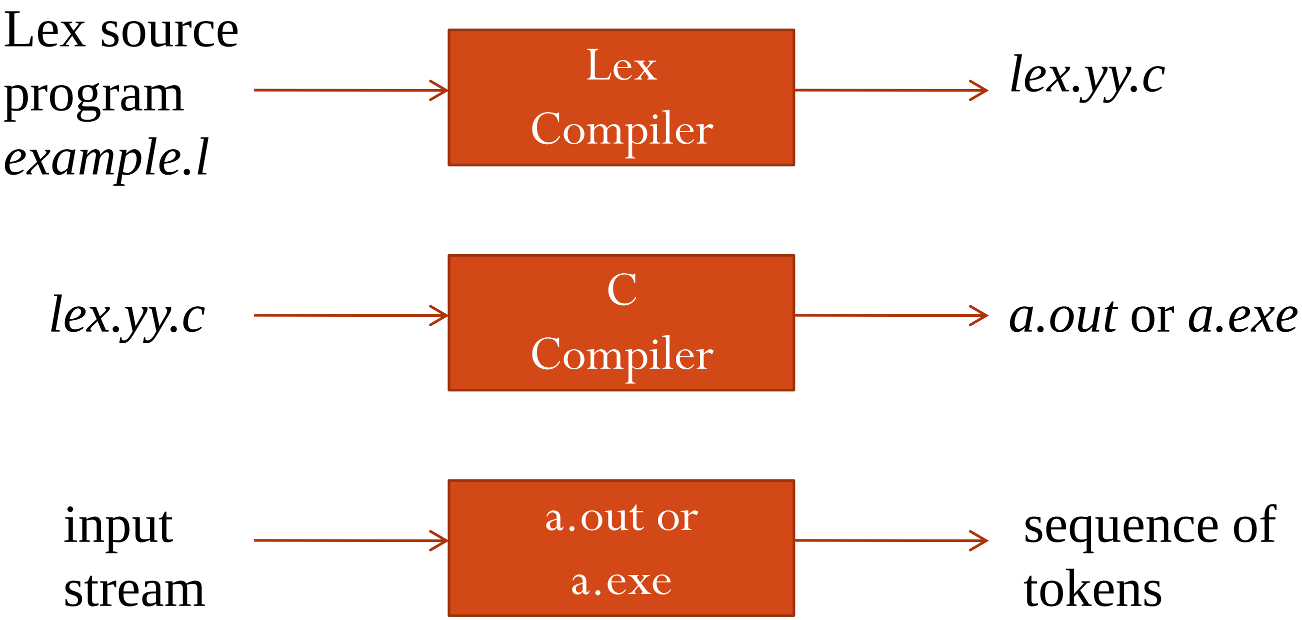
Based on J.R. Levine, T. Mason, D. Brown, *lex & yacc*,
O'Reilly & Associates, Inc., 1990.

Table of Contents

- Constructing Lexical Analyzers
- Lex Specifications
- Regular Expressions in Lex
- Generated Lexical Analzyer
- Matches and Actions
- Examples
- Separated Driver
- Special Directives

Constructing Lexical Analyzers

- Lex includes
  - Lex compiler
  - Lex language



Lex source program *example.l* → Lex Compiler → *lex.yy.c*

lex.yy.c → C Compiler → *a.out* or *a.exe*

input stream → a.out or a.exe → sequence of tokens

Lex Specifications

declarations
%%
translation rules
%%
auxiliary procedures

- declarations
  - declarations of variables
  - manifest constants
  - regular definitions
- translations rules

p_1 { action-1 }
p_2 { action-2 }
…
p_N { action-N }

p_i : regular expression
action-i : program fragment

- Auxiliary procedures

whatever procedures needed by actions

Regular Expressions in Lex

- Operators

  " \ [ ] ^ - ? . * + | ( ) $ / { } % < >

EXPRESSION	MATCHES	EXAMPLE
c	any non-operator character c	a
\c	character c literally	*
“s”	string s literally	“**”
.	any character but newline	a.*b
^	beginning of line	^The
$	end of line	file\.$
[s]	any character in s (^ - \ are special)	[abc]
[^s]	any character not in s	[^abc]
r*	zero or more r’s	a*
r+	one or more r’s	a+
r?	zero or one r	a?
r{m,n}	m to n occurrences or r	a{1,5}
r_1r_2	r1 then r2	ab
r_1|r_2	r1 or r2	a|b
(r)	r	(a|b)
r_1/r_2	r1 when followed by r2	abc/123

Regular Expressions in Lex

- Examples

```
[0-9]
[0-0]*
-?[0-9]+
[0-9]*\.[0-9]+
([0-9]+)|([0-9]*\.[0-9]+)
-?(([0-9]+)|([0-9]*\.[0-9]+))
[eE][-+]?[0-9]+
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)

C“++” or C\+\+           (C++)
object$ or object/\n     (line의 끝에 나타나는 object)
^The                     (line의 시작에 나타나는 The)
foo|bar* vs foo|(bar)*
```

Generated Lexical Analyzer

- **lex.yy.c** contains the scanning function '**yylex()**'

  ```
  int yylex()
  {   …  }
  ```

- **yylex()**
  - scans tokens from the global input file **yyin** (default value is **stdin**)
  - continues until it either reaches an EOF or executes a return statement
  - If **yylex()** stops scanning due to executing a return statement, the scanner may the be called again and it will resume scanning
  - **yylex()** returns
    - 0 at the end of file
    - the value an action routine defines (returns)

Matches and Actions

- Type of matches
  - Only one match
    - perform the corresponding action
  - More than one matches (of different lengths)
    - apply *the longest string principle*
  - More than one matches (of the same length)
    - apply the rule listed first
  - No match
    - the *default rule* is executed i.e. the next character is copied to its output
- Variables used in yylex()
  - **yytext[]** contains the string recognized
  - **yyleng** has the length of the token string
  - **yyin** is the file pointer to the input file (default is **stdin**)

Matches and Actions

- Actions
  - each pattern has a corresponding action
  - any arbitrary C statement
    - empty – ignoring the token
    - a sequence of statements
    - a block
  - action can or cannot include a return statement
    - if it does not include any return, **yylex()** continues to processing tokens
  - vertical bar(|) – same as the action for the next rule

e.g.

```
“ ”   |
\t    |
\n        printf(“ ”);
```

Examples

- Letter or Digit

  example.l

```
%{
#define LETTER 1
#define DIGIT 2
%}
blank [ \t\n]+
letter [a-zA-Z]
digit [0-9]
%%
{blank} ;
{letter} {return LETTER;}
{digit} {return DIGIT;}
%%
```

```
int main(void)
{
    int tok;
    while((tok=yylex())!=0)
        if(tok==LETTER)
            printf("letter! \n");
        else printf("digit!\n");
}
```

Examples

```
C:\flex> flex example.l
C:\flex> gcc lex.yy.c   -L” C:\MinGW\msys\1.0\lib”  -lfl
C:\flex> a.exe
0
digit!
a
letter!
+                    // no match
+
^Z
```

MinGW package 설치
// C:\MinGW\msys\1.0\bin\{flex,bison}
// C:\MinGW\msys\1.0\lib\{libfl.a,liby.a}

- empty action – discard the input token

  `{blank} ;   or {blank}{} or {blank}`

Examples

- Letter or Digit Revisited
  - using yytext[], yyleng , yylval variable

example2.l

```
%{
extern int yylval;
#define LETTER 1
#define DIGIT 2
%}
blank [ \t\n]+
letter [a-zA-Z]
digit [0-9]
%%
{blank}  {}
{letter}  {yylval=yytext[0]; return LETTER;}
{digit}  {yylval=yytext[0]-'0'; return DIGIT;}
%%
```

```
int yylval;
int main(void)
{
    int tok;
    while((tok=yylex())!=0)
        if(tok==LETTER)
             printf("letter %c!  \n",yylval);
        else
             printf("digit %d!\n", yylval);
}
```

Examples

```
C:\flex> flex example2.l
C:\flex> cl lex.yy.c /Feexample2 /I include /link /LIBPATH:lib libfl.a
C:\flex> example2
0
digit 0!
a
letter a!
^Z
```

MSVC Compiler

Examples

- Word Count
  - precedence of matching
  - without return in action routine

```
%{
unsigned charCount=0, wordCount=0, lineCount=0;
%}
word [^ \t\n]+
eol \n
%%
{word}  {wordCount++; charCount += yyleng; }  /*charCount +=strlen(yytext); */
{eol}    {charCount++; lineCount++; }
.          charCount++;
%%
main()
{
     yylex();
     printf("%d %d %d\n", charCount, wordCount, lineCount);
}
```

Examples

- Using Separated Driver and File Input
  - using **yyin** file pointer
  - external declaration of **yylex()** and **yyin** in the separated driver routine

in lex.1

lex specification

in driver routine (e.g. main.c)

```
extern int yylex();
extern FILE *yyin;  /* if input from file */
```

same symbolic constants definition as lex.l if any (e.g. LETTER in example.l)

Examples

in file.l

```
%{
extern unsigned charCount, wordCount, lineCount;
%}
word [^ \t\n]+
eol \n
%%
{word} { wordCount++; charCount += yyleng; }
{eol}   { charCount++; lineCount++; }
.                          charCount++;
%%
```

Examples

in main.c

```
#include <stdio.h>
#include <stdlib.h>

unsigned charCount = 0, wordCount = 0, lineCount = 0;
extern FILE *yyin;
extern int yylex();

int main(int argc, char* argv[])
{
        if (argc > 1) {
                    FILE *file;
                    file = fopen(argv[1], "r");
                    if (!file) {
                      fprintf(stderr,"could not open %s\n",argv[1]);
                      exit(-1);
          }
          yyin = file;
  }
  yylex();
  printf("%d %d %d\n",charCount, wordCount, lineCount);
  return 0;
}
```

Special Routines

- **ECHO** : copies yytext to the scanner's output.

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

- **REJECT** : directs the scanner to proceed on to the "second best" rule

```
int word_count = 0;
%%
abc          special(); REJECT;
[^ \t\n]+    ++word_count;
```

Special Routines

- **input()** : read a next character
- **output()** : writes a character on an output device
- **unput(c)** :puts the character c back onto the input stream

e.g. ignore all characters between " and "

```
\"        while (input() != '"');
```

Special Routines

- **yywrap()**
  - when yylex() reaches the end of its input file, it calls yywrap()
  - returns a value of 0 or 1
  - if the value is 1, the program is done (no more input)
  - if the value is 0, the lexer assumes that yywrap() has opened another file for it to read and assigned the open file to **yyin**
  - by default, yywrap() returns 1

Special Routines

- **yymore()** : tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of yytext

```
%%
mega- ECHO; yymore();
kludge ECHO;
input  : mega- and then kludge
output : mega-mega-kludge
```

- **yyless(n)** :returns all but the first *n* characters of the current token back to the input stream where they will be rescanned when the scanner looks for the next match

```
%%
foobar     ECHO; yyless(3);
[a-z]+     ECHO;
input  : foobar
output : foobarbar
```

References

- J.R. Levine, T. Mason, D. Brown, *lex & yacc*, O'Reilly & Associates, Inc., 1990.
- Aho, Lam, Sethi, Ullman: *Compilers: Principles, Techniques, & Tools,* 2nd Ed., Addison Wesley, 2007