中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compiler Design
编译器构造实验

Lab 1： Lex Tool

张献伟

xianweiz.github.io

DCS292, 2/24/2022

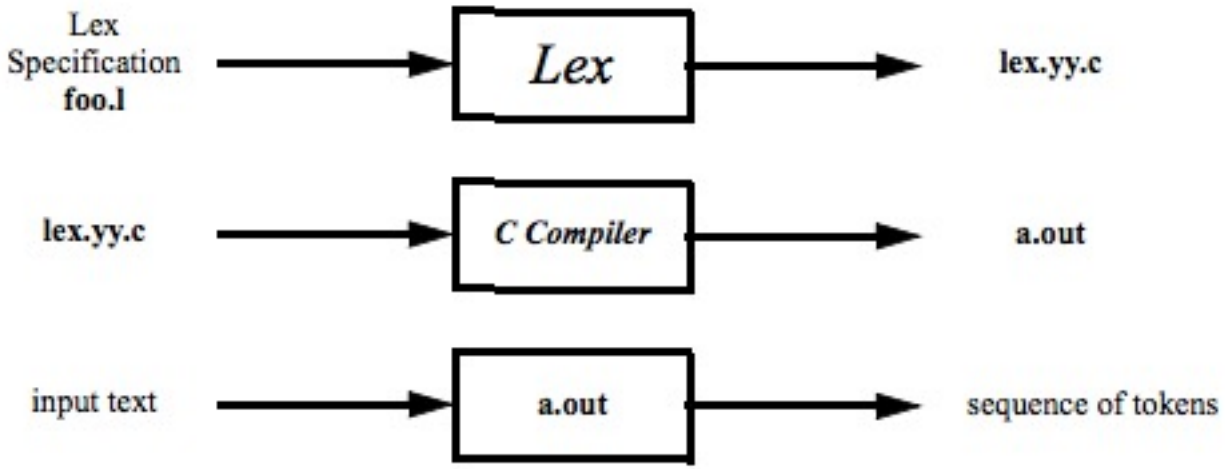
Lex Installation and Docs

- ubuntu
  - Check whether it has been installed: $lex -V
  - $sudo apt-get install flex

- Documents for reference
  - Lex – A Lexical Analyzer Generator, https://www.csee.umbc.edu/~chang/cs431/Lex_Manual.pdf
  - Using LEX, https://silcnitc.github.io/lex.html
  - A Lex Tutorial, https://www.cse.iitb.ac.in/~br/courses/cs699-autumn2013/refs/lextut-victor-eijkhout.pdf
  - How do Lex and YACC work internally, http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-6.html

Lex: Tool for Lexical Analysis

- Lex
  - Allows to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens
  - The tool itself is a Lex compiler
    - Transforms the input patterns into a transition diagram and generates code in a file lex.yy.c
    - Lex.yy.c simulates the transition diagram



How to Write *.l

- Different from your previous coding experience
  - Write REs instead of C code
  - Write actions in C associated with each RE
  - lex.yy.c is C code after REs in *.l are translated to C

```
… declarations …

%%

… translation rules …

%%

… auxiliary functions …
```

How to Write *.l (cont.)

- Structure of a Lex program
  - Declarations: include, extern, variables, patterns, etc.
    - Pattern format: <name> <definition>
  - Translation rules: pattern {Action}
    - Pattern: a RE, which may use the regular definitions in declarations
    - Action: fragments of code, typically written in C
  - Auxiliary functions: additional user-defined functions

```
%{
 int charcount = 0, linecount = 0;
%}

%%

. charcount++;
^(.*)\n { linecount++; charcount++; printf("%4d: %s\n", linecount, yytext); }

%%

int main()
{
  yylex();
  printf("There are %d chars in %d lines\n", charcount, linecount);

  return 0;
}
```

Example

```
1  %{
2   int yyline = 0, yyid = 0;
3  %}
4
5  letter       [a-zA-Z]
6  digit        [0-9]
7  id           {letter}({letter}|{digit})*
8  id_error     {digit}({letter}|{digit})*
9
10 newline \n
11
12 %%
13
14 {id}         {
15                 yyid ++;
16                 printf("\nid: %s (len=%zu)", yytext, yyleng);
17               }
18 {id_error}   {
19                 printf("\nwrong id: %s (len=%zu)", yytext, yyleng);
20               }
21 {newline}    { yyline ++; }
22
23 %%
24
25 int main() {
26   yylex();
27
28   printf("\n\n%d lines in total ...\n", yyline);
29
30   return 0;
31 }
```

How to Run?

- Translate into C
  - $lex example.l
- Compile the C code
  - $gcc -o <my> lex.yy.c -ll
- Run with input
  - $./my < sample.txt

```
value
test
0test
rst
100
```

input

```
id: value (len=5)
id: test (len=4)
wrong id: 0test (len=5)
id: rst (len=3)
wrong id: 100 (len=3)

5 lines in total ...
```

output

lex.yy.c

```
452  int yyline = 0, yyid = 0;
609 #define YY_DECL int yylex (void)

629 YY_DECL
630 {
631     register yy_state_type yy_current_state;
632     register char *yy_cp, *yy_bp;
633     register int yy_act;

666     while ( 1 )     /* loops until end-of-file is reached */
667         {
712         switch ( yy_act )
713     { /* beginning of action switch */
721 case 1:
724 {
725             yyid ++;
726             printf("\nid: %s (len=%zu)", yytext, yyleng);
727         }
729 case 2:
732 {
733             printf("\nwrong id: %s (len=%zu)", yytext, yyleng);
734         }
736 case 3:
740 { yyline ++; }
```

Lex Variables and Functions

- yytext
  - Of type char* and it contains the lexeme currently found
    - Lexeme is a sequence of chars in the input stream that matches some pattern in the Rules section
- yyleng
  - Of type int and it stores the length of the lexeme pointed by yytext
- yylex()
  - Scans through the input looking for a matching pattern