



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# CS323 Lab 3

Yepang Liu

[liuyp1@sustech.edu.cn](mailto:liuyp1@sustech.edu.cn)

# Outline

- Flex Tutorial
- SPL Grammar Rules

# The Lexical-Analyzer Generator Lex

- Lex, or a more recent tool Flex, allows one to specify a lexical analyzer by specifying regexps to describe patterns for tokens
- Often used with Yacc/Bison to create the frontend of compiler

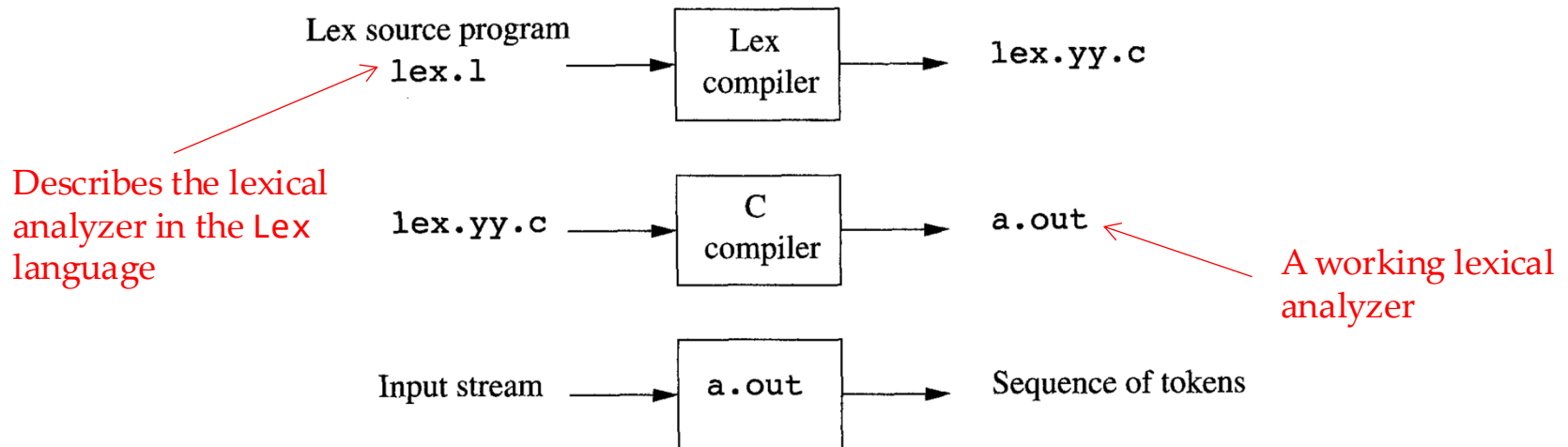


Figure 3.22: Creating a lexical analyzer with Lex

# Structure of Lex Programs

- A Lex program has three sections separated by %%
  - Declaration (声明)
    - Variables, constants (e.g., token names)
    - Regular definitions
  - Translation rules (转换规则) in the form “Pattern {Action}”
    - Each pattern (模式) is a regexp (may use the regular definitions of the declaration section)
    - Actions (动作) are fragments of code, typically in C, which are executed when the pattern is matched
  - Auxiliary functions section (辅助函数)
    - Additional functions that can be used in the actions

# Lex Program Example

```
%{  
    /* definitions of manifest constants  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
%}  
  
/* regular definitions */  
delim      [ \t\n]  
ws          {delim}+  
letter     [A-Za-z]  
digit      [0-9]  
id          {letter}({letter}|{digit})*  
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?  
  
%%
```

Anything in between %{ and }% is copied directly to lex.yy.c.

In the example, there is only a comment, not real C code to define manifest constants

Regular definitions that can be used in translation rules

Section separator

# Lex Program Example Cont.

```
{ws}      { /* no action and no return */ }
if         { return(IF); }
then       { return(THEN); }
else       { return(ELSE); }
{id}       { yylval = (int) installID(); return(ID); }
{number}   { yylval = (int) installNum(); return(NUMBER); }
"<"       { yylval = LT; return(RELOP); }
"<="      { yylval = LE; return(RELOP); }
"="        { yylval = EQ; return(RELOP); }
"<>"      { yylval = NE; return(RELOP); }
">"       { yylval = GT; return(RELOP); }
">="      { yylval = GE; return(RELOP); }
```

Continue to recognize other tokens

Return token name to the parser

Place the lexeme found in the symbol table

%%

A global variable that stores a pointer to the symbol table entry for the lexeme. Can be used by the parser or a later component of the compiler.

\* The characters inside have no special meaning (even if it is a special one such as \*).

# Lex Program Example Cont.

- Everything in the auxiliary function section is copied directly to the file `lex.yy.c`
- Auxiliary functions may be used in actions in the translation rules

```
int installID() { /* function to install the lexeme, whose
                  first character is pointed to by ytext,
                  and whose length is yleng, into the
                  symbol table and return a pointer
                  thereto */
}
```

Variables defined and set automatically  
by the lexical analyzer Lex generates

```
int installNum() { /* similar to installID, but puts numer-
                    ical constants into a separate table */
}
```

# Conflict Resolution

- When the generated lexical analyzer runs, it analyzes the input looking for **prefixes that match any of its patterns.**\*
- **Rule 1:** If it finds multiple such prefixes, it takes the **longest** one
  - The analyzer will treat <= as a single lexeme, rather than < as one lexeme and = as the next
- **Rule 2:** If it finds a prefix matching different patterns, **the pattern listed first** in the Lex program is chosen.
  - If the keyword patterns are listed before identifier pattern, the lexical analyzer will not recognize keywords as identifiers

\* See Flex manual for details (Chapter 8: How the input is matched) at <http://dinosaur.compilertools.net/flex/>



# Flex

- Flex的前身是Lex。Lex是1975年由Mike Lesk和当时还在贝尔实验室做暑期实习的Eric Schmidt（前谷歌CEO），共同完成的一款基于Unix环境的词法分析程序生成工具。虽然Lex很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。
- 1987年伯克利实验室（隶属美国能源部的国家实验室）的Vern Paxson使用C语言重写Lex，并将这个新程序命名为Flex（Fast Lexical Analyzer Generator）。无论从效率上还是稳定性上，Flex都远远好于它的前辈Lex。

\*我们在Linux下使用的是Flex在BSD License下的版本（和Bison不同，Flex不属于GNU计划）。

# An Example Flex Program

- A word-count program (see the code under lab3/wc)
- Build the program with the following commands (or “make wc”)
  - `flex lex.l` (you will see a `lex.yy.c` file generated)
  - `gcc lex.yy.c -lfl -o wc.out`

```
yepang@yepang-x1:~/Desktop/CS323-2024F/lab3/wc$ ./wc.out inferno3.txt
#lines  #words  #chars  file path
162      1088    6525    inferno3.txt
```

# A Closer Look

```
1 %{
2     // just let you know you have macros!
3     // C macro tutorial in Chinese: http://c.biancheng.net/view/446.html
4     #define EXIT_OK 0
5     #define EXIT_FAIL 1
6
7     // global variables
8     int chars = 0;
9     int words = 0;
10    int lines = 0;
11 %}
12 letter [a-zA-Z]
13
14 %%
15 {letter}+ { words++; chars+=strlen(yytext); }
16 \n { chars++; lines++; }
17 . { chars++; }
18
19 %%
20 int main(int argc, char **argv){
21     char *file_path;
22     if(argc < 2){
23         fprintf(stderr, "Usage: %s <file_path>\n", argv[0]);
24         return EXIT_FAIL;
25     } else if(argc == 2){
26         file_path = argv[1];
27         if(!(yyin = fopen(file_path, "r"))){
28             perror(argv[1]);
29             return EXIT_FAIL;
30         }
31         yylex();
32         printf("%-8s%-8s%-8s\n", "#lines", "#words", "#chars", "file path");
33         printf("%-8d%-8d%-8d\n", lines, words, chars, file_path);
34         return EXIT_OK;
35     } else{
36         fputs("Too many arguments! Expected: 2.\n", stderr);
37         return EXIT_FAIL;
38     }
39 }
```

The structure is the same as in a Lex program:

1. Declaration
2. Translation rules
3. Auxiliary functions

# More on Flex patterns

## Flex supports a rich set of conveniences:

Character classes	<code>[0-9]</code>	This means alternation of the characters in the range listed (in this case: <code>0</code>   <code>1</code>   <code>2</code>   <code>3</code>   <code>4</code>   <code>5</code>   <code>6</code>   <code>7</code>   <code>8</code>   <code>9</code> ). More than one range may be specified, e.g. <code>[0-9A-Za-z]</code> as well as specifying individual characters, as with <code>[aeiou0-9]</code> .
Character exclusion	<code>^</code>	The first character in a character class may be <code>^</code> to indicate the complement of the set of characters specified. For example, <code>[^0-9]</code> matches any non-digit character.
Arbitrary character	<code>.</code>	The period matches any single character <b>except newline</b> .
Single repetition	<code>x?</code>	0 or 1 occurrence of <b>x</b> .

# More on Flex patterns

Non-zero repetition	<b><code>x+</code></b>	<b><code>x</code></b> repeated one or more times; equivalent to <b><code>xx*</code></b> .
Specified repetition	<b><code>x{n,m}</code></b>	<b><code>x</code></b> repeated between <b><code>n</code></b> and <b><code>m</code></b> times.
Beginning of line	<b><code>^x</code></b>	Match <b><code>x</code></b> at beginning of line only.
End of line	<b><code>x\$</code></b>	Match <b><code>x</code></b> at end of line only.
Context-sensitivity	<b><code>ab/cd</code></b>	Match <b><code>ab</code></b> but only when followed by <b><code>cd</code></b> . The lookahead characters are left in the input stream to be read for the next token.
Literal strings	<b><code>"x"</code></b>	This means <b><code>x</code></b> even if <b><code>x</code></b> would normally have special meaning. Thus, <b><code>"x*"</code></b> may be used to match <b><code>x</code></b> followed by an asterisk. You can turn off the special meaning of just one character by preceding it with a backslash, .e.g. <b><code>\.</code></b> matches exactly the period character and nothing more.
Definitions	<b><code>{name}</code></b>	Replace with the earlier defined pattern called <b><code>name</code></b> . This kind of substitution allows you to re-use pattern pieces and define more readable patterns.

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf>

# Flex Exercise: C Identifier

- Count the occurrences of valid C identifiers
  - A valid C identifier starts with an English letter or an underscore followed by any number of English letters, digits, or underscores
- We make some assumptions to simplify the task
  - Only these reserved words may appear: char, int, return, while, if, else
  - There are no preprocessor commands (e.g., `#include <stdio.h>`)
  - There are no function calls

# Flex Exercise: C Identifier

1. Please modify the `lex.1` file under the `lab3/identifier` directory
2. Build the lexer
  - `make idcount`
3. Run the counting program
  - `./idcount.out test.c`

If you get the following output, your implementation is likely correct.

```
line 1: main
line 3: a
line 4: BBA
line 4: a_
line 5: _
line 7: a0
line 7: _
line 7: b0
line 8: _
line 8: b0
line 9: b
line 9: b1
line 9: b0
line 9: b2
line 10: c
There are 15 occurrences of valid identifiers
```

Please finish this simple exercise during class 😊

# Outline

- Flex Tutorial
- SPL Grammar Rules



# SPL Grammar Rules

```
/* high-level definition */
Program -> ExtDefList
ExtDefList -> ExtDef ExtDefList
           | $
ExtDef -> Specifier ExtDecList SEMI
       | Specifier SEMI
       | Specifier FunDec CompSt
ExtDecList -> VarDec
           | VarDec COMMA ExtDecList
```

```
/* specifier */
Specifier -> TYPE
           | StructSpecifier
StructSpecifier -> STRUCT ID LC DefList RC
               | STRUCT ID
```

```
/* declarator */
VarDec -> ID
        | VarDec LB INT RB
FunDec -> ID LP VarList RP
        | ID LP RP
VarList -> ParamDec COMMA VarList
        | ParamDec
ParamDec -> Specifier VarDec
```

```
/* statement */
CompSt -> LC DefList StmtList RC
StmtList -> Stmt StmtList
          | $
Stmt -> Exp SEMI
      | CompSt
      | RETURN Exp SEMI
      | IF LP Exp RP Stmt
      | IF LP Exp RP Stmt ELSE Stmt
      | WHILE LP Exp RP Stmt
```

```
/* local definition */
DefList -> Def DefList
        | $
Def -> Specifier DecList SEMI
DecList -> Dec
        | Dec COMMA DecList
Dec -> VarDec
    | VarDec ASSIGN Exp
```

```
/* Expression */
Exp -> Exp ASSIGN Exp
    | Exp AND Exp
    | Exp OR Exp
    | Exp LT Exp
    | Exp LE Exp
    | Exp GT Exp
    | Exp GE Exp
    | Exp NE Exp
    | Exp EQ Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp MUL Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp
    | NOT Exp
    | ID LP Args RP
    | ID LP RP
    | Exp LB Exp RB
    | Exp DOT ID
    | ID
    | INT
    | FLOAT
    | CHAR
Args -> Exp COMMA Args
    | Exp
```

<https://github.com/sqlab-sustech/CS323-2024F/blob/main/spl-spec/syntax.txt>

# Example

The parse tree:

```
int test_1_r01(int a, int b)
{
    c = 'c';
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

A syntactically valid program\*

\* Here, the vairable c is used without definition.  
This error will be caught during semantic analysis.

```
1 Program (1)
2 ExtDefList (1)
3 ExtDef (1)
4 Specifier (1)
5 TYPE: int
6 FunDec (1)
7 ID: test_1_r01
8 LP
9 VarList (1)
10 ParamDec (1)
11 Specifier (1)
12 TYPE: int
13 VarDec (1)
14 ID: a
15 COMMA
16 VarList (1)
17 ParamDec (1)
18 Specifier (1)
19 TYPE: int
20 VarDec (1)
21 ID: b
22 RP
23 CompSt (2)
24 LC
25 StmtList (3)
26 Stmt (3)
27 Exp (3)
28 Exp (3)
29 ID: c
30 ASSIGN
31 Exp (3)
32 CHAR: 'c'
33 SEMI
34 StmtList (4)
35 Stmt (4)
36 IF
37 LP
38 Exp (4)
39 Exp (4)
40 ID: a
41 GT
42 Exp (4)
43 ID: b
44 RP
45 Stmt (5)
46 CompSt (5)
47 LC
48 StmtList (6)
49 Stmt (6)
50 RETURN
51 Exp (6)
52 ID: a
53 SEMI
54 RC
55 ELSE
56 Stmt (9)
57 CompSt (9)
58 LC
59 StmtList (10)
60 Stmt (10)
61 RETURN
62 Exp (10)
63 ID: b
64 SEMI
65 RC
66 RC
```

# Example

```
1  int test_1_r03()  
2  {  
3      int i = 0, j = 1;  
4      float i = $;  
5      if(i < 9.0){  
6          return 1  
7      }  
8      return @;  
9  }
```

```
Error type A at Line 4: unknown lexeme $  
Error type B at Line 6: Missing semicolon ';'   
Error type A at Line 8: unknown lexeme @
```