



# Introduction to Lex

- **General Description**
- **Input file**
- **Output file**
- **How matching is done**
- **Regular expressions**
- **Local names**
- **Using Lex**



# General Description

- **Lex** is a program that automatically generates code for scanners.
- Input: a description of the tokens in the form of regular expressions, together with the actions to be taken when each expression is matched.
- Output: a text file with C source code defining a procedure `yylex()` that is a table implementing the DFA for the regular expressions.



# Input File

- **Lex input file is divided in three**

```
/* declarations*/  
...  
  
%%  
/* rules */  
...  
  
%%  
/* auxiliary functions*/  
...
```



# Input File – Declaration

- The declaration part includes the assignment of names to regular expressions in the form:

`<name> <regular_exp>`

- It can also include C code external to the definition of `yylex()` within `%{` and `%}` in the first column.
- Also, it is possible to specify some options with the syntax `%option`



# Input File - Rules

- The rules part specifies what to do when a regular expression is matched

`<regular_exp> <action>`

- Actions are normal C sentences (can be a complex C sentence between `{}`).



# Input File – Aux Functions

- The auxiliary functions part is only C code.
- It includes function definitions for every function needed in the rule part
- It can also contain the `main()` function if the scanner is going to be used as a standalone program.
- The `main()` function must call the function `yylex()`



# Code Generated by Lex

- The output of Lex is a file called `lex.yy.c`
- It is a C function that returns an integer, i.e., a code for a ***Token***
- If it contains the `main()` function definition, it must be compiled to run.
- Otherwise, the code can be an external function declaration for the function `int yylex()`



# How matching is done

- By running the generated scanner, it analyzes its input looking for strings that match any of its patterns, and then executes the action.
- If it finds more than one match, it selects the expression matching the longest text.
- If it finds two or more matches of the same length, the one listed first is selected.
- If no match is found, then the default rule is executed: i.e., the next character in the input is copied to the output.



# Regular expressions

- $\langle \text{char} \rangle ::= a$  the character  $a$
- $\langle \text{char} \rangle ::= "s"$  the string  $s$ , even if  $s$  contains metacharacters
- $\langle \text{char} \rangle ::= \backslash a$  the character  $a$  when  $a$  is a metacharacter
- $\langle \text{char} \rangle ::= .$  any character except newline
- $\langle \text{regExp} \rangle ::= [\langle \text{char} \rangle^+]$  any of the character  $\langle \text{char} \rangle$
- $\langle \text{regExp} \rangle ::= [\langle \text{char}1 \rangle - \langle \text{char}2 \rangle]$  any character from  $\langle \text{char}1 \rangle$  to  $\langle \text{char}2 \rangle$
- $\langle \text{regExp} \rangle ::= [^\langle \text{char} \rangle^+]$  any character except those  $\langle \text{char} \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp}1 \rangle^*$  zero or more repetitions of  $\langle \text{regExp}1 \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp}1 \rangle^+$  one or more repetitions of  $\langle \text{regExp}1 \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp}1 \rangle?$  zero or one repetitions of  $\langle \text{regExp}1 \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp}1 \rangle | \langle \text{regExp}2 \rangle$   $\langle \text{regExp}1 \rangle$  or  $\langle \text{regExp}2 \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp}1 \rangle \langle \text{regExp}2 \rangle$   $\langle \text{regExp}1 \rangle$  followed by  $\langle \text{regExp}2 \rangle$
- $\langle \text{regExp} \rangle ::= (\langle \text{regExp}1 \rangle)$  same as  $\langle \text{regExp}1 \rangle$
- $\langle \text{regExp} \rangle ::= \{ \langle \text{name} \rangle \}$  the named regular expression in the definitions part



# Internal names

- The rules, inside the action definition, can refer to the following variables:
  - `yytext`, the string being matched (lexeme)
  - `yyin`, the input file
  - `yyout`, the output file
  - `ECHO`, the default rule action
  - `yyval`, the global variable for communicating the Attribute for a Token to the Parser



# Using Lex

- There are several lex versions. We're going to use flex.
- In order to maximize compatibility, use `-l` option when compiling, and `%option noyywrap` in the definition part of the Lex input file.



# Example 1

```
%{  
#include <stdio.h>  
%}  
  
%%  
[0-9]+ { printf("%s\n", yytext); }  
.\n ;  
  
%%  
main()  
{  
    yylex();  
}
```



## Example 2

```
%{
    int c=0, w=0, l=0;
}%
word [^ \t\n]+
eol \n

%%
{word} {w++; c+=yyleng;};
{eol} {c++; l++;}
.      {c++;}

%%
main()
{
    yylex();
    printf("%d %d %d\n", l, w, c);
}
```



## Example 3

```
%{  
    int tokenCount=0;  
}%  
  
%%  
[a-zA-Z]+ { printf("%d WORD \"%s\"\n",  
                ++tokenCount, yytext); }  
[0-9]+    { printf("%d NUMBER \"%s\"\n",  
                ++tokenCount, yytext); }  
[^a-zA-Z0-9]+ { printf("%d OTHER \"%s\"\n",  
                ++tokenCount, yytext); }  
  
%%  
main() {    yylex(); }
```



## Example 4

```
%{  
#include <stdio.h>  
int lineno =1;  
%}  
  
line .*\\n  
  
%%  
{line} {printf("%5d %s", lineno++, yytext); }  
  
%%  
int main() {  
    yylex(); return 0; }
```



# Example 5

```
%{  
#include <stdio.h>  
%}  
  
comment_line \\V.*\n  
  
%%  
{comment_line} { printf("%s\n", yytext); }  
.*\n ;  
%%  
  
int main() {  
    yylex(); return 0;  
}
```



# Example 7

```
%{  
#include <stdio.h>  
  
%}  
digit [0-9]  
number {digit}+  
%%  
{number}    { int n = atoi(yytext);  
              printf("%x", n); }  
.           {;}  
%%  
  
int main() {  
    yylex(); return 0;  
}
```



## Example 8 (1/4)

```
%}  
#include "globals.h"  
#include "util.h"  
#include "scan.h"  
int lineno=0;  
FILE *listing;    // used to output source code listing  
FILE *code;       // used to output assembly code  
FILE *source;     // used to input tiny program source  
code  
int TraceScan = 1;  
int EchoSource = 1;  
  
/* lexeme of identifier or reserved word */  
char tokenString[MAXTOKENLEN+1];  
%}  
  
digit      [0-9]  
number     {digit}+  
letter     [a-zA-Z]  
identifier {letter}+  
newline    \n  
whitespace [ \t]+
```





## Example 8 (3/4)

```
{number}   {return NUM;}
{identifier} {return ID;}
{newline}   {lineno++;}
{whitespace} { /* skip whitespace */}
"{"        { char c;
           do
           { c = input();
             if (c == '\n') lineno++;
           } while (c != '}');
           }
```



## Example 8 (4/4)

```
%%  
TokenType getToken(void)  
{ static int firstTime = TRUE;  
  TokenType currentToken;  
  if (firstTime)  
  { firstTime = FALSE;  
    lineno++;  
    yyin = source=stdin;  
    yyout = listing=stdout; }  
  currentToken = (TokenType)yylex();  
  strncpy(tokenString,yytext,MAXTOKENLEN);  
  if (TraceScan) {  
    fprintf(listing,"\t%d: ",lineno);  
    printToken(currentToken,tokenString);  
  }  
  return currentToken; }  
  
int main(){  
  TraceScan = TRUE;  
  while( getToken() != ENDFILE);  
  return 0; }
```