

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 drive implementation of the DFA for the regular expressions.



Input File

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

```
/* definitions */
```

```
...
```

```
%%
```

```
/* rules */
```

```
...
```

```
%%
```

```
/* auxiliary routines */
```

```
...
```



Input File

- The definition 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

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

`<regular_exp> <action>`

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



Input File

- The auxiliary routines part is only C code.
- It includes function definition for every function needed in rules part
- It can also contain the `main()` function definition if the scanner is going to be run as a program.
- The `main()` function must call the function `yylex()`



Input File

- The output of Lex is a file called `lex.yy.c`
- If it contains the `main()` function definition, it must be compiled to be run.
- Otherwise, the code can be an external function definition for the function

```
int yylex();
```



How matching is done

- When the generated scanner is run, it analyzes its input looking for strings that match any of its patterns, and then executes the action.
- If it find more than one match, it selects the expression matching the most 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: the next character in the input is copied to the output.



Regular expressions

- $\langle \text{char} \rangle ::= a$ the character a
- $\langle \text{char} \rangle ::= "a"$ the character a, even if a is a metacharacter
- $\langle \text{char} \rangle ::= \backslash a$ 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 $\langle \text{regExp} 1 \rangle$
- $\langle \text{regExp} \rangle ::= \langle \text{regExp} 1 \rangle | \langle \text{regExp} 2 \rangle$ either $\langle \text{regExp} 1 \rangle$ or $\langle \text{regExp} 2 \rangle$
- $\langle \text{regExp} \rangle ::= (\langle \text{regExp} 1 \rangle)$ $\langle \text{regExp} 1 \rangle$ used for operator precedence
- $\langle \text{regExp} \rangle ::= \{ \langle \text{name} \rangle \}$ the named regular expression in the definitions part



Internal names

- In rules, inside the action definition, it is possible to refer to the following:
 - `yytext`, the string being match.
 - `yyin`, the input file
 - `yyout`, the output file
 - `ECHO`, the default rule action
 - `yyval`, an additional variable for communication between functions



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 \W.*\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 (2/4)

```
%%  
"if"      {return IF;}  
"then"    {return THEN;}  
"else"    {return ELSE;}  
"end"     {return END;}  
"repeat"  {return REPEAT;}  
"until"   {return UNTIL;}  
"read"    {return READ;}  
"write"   {return WRITE;}  
":="      {return ASSIGN;}  
"="        {return EQ;}  
"<"       {return LT;}  
"+"       {return PLUS;}  
"- "      {return MINUS;}  
"*"       {return TIMES;}  
"/"       {return OVER;}  
"("       {return LPAREN;}  
")"       {return RPAREN;}  
"."       {return SEMI;}  
","       {return SEMI;}  
"."       {return ERROR;}
```



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; }
```

