Lexical Analysis
Part II

Chapter 3: Flex

Slides adapted from :
© Robert van Engelen, Florida State University
The Lex and Flex Scanner Generators

- **Lex** and its newer cousin **Flex** are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications
Creating a Lexical Analyzer with Lex and Flex

lex source program lex.fl

Lex or Flex compiler

lex.yy.c

C compiler

lex.yy.c

a.out

input stream

a.out

sequence of tokens

C compiler
Lex Specification

• A lex specification consists of three sections separated by “%%”:

  C declarations enclosed by %{} and %}
  regular definitions
  %%
  translation rules (see next slide)
  %%
  user-defined auxiliary C procedures (optional)
Lex Specification

• The translation rules (second section) are of the form:

\[ p_1 \{ action_1 \} \]
\[ p_2 \{ action_2 \} \]
\[ \ldots \]
\[ p_n \{ action_n \} \]

where \( p_i \) is a regular expression and might use regular definitions from first section.
Example: `copy.fl`

- No translation rule
- No declaration
  - no regular definition
- No auxiliary Procedure (optional)

```
flex copy.fl
gcc lex.yy.c -o primo -lfl
./primo < prova.txt > out.txt
```
Example: `deletethis.fl`

```
% deletethis
%
```

- Translation rule but no action
- No declaration
  - No regular definition
- No auxiliary procedure

```
flex deletethis.fl
gcc lex.yy.c -o scan -lfl
./scan
This deletethis is not deletethis useful.
This is not useful.
^D
```
Example: replacer.fl

```
flex -o replacer.yy.c replacer.fl
gcc -o replacer replacer.yy.c -lfl
./replacer
```
This replacethis is not very replacethis useful.
This replaced is not very replaced useful.
Please dontreplacethisatall.
Please dontreplacedatall.
^D
Example: intextract.fl

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

Copied to output .cc file
Contains the matching lexeme
Invokes the lexical analyzer

Lex intextract.fl
gcc lex.yy.c -lfl
./a.out < intextract.fl
0
9
Regular Expressions in Lex

\* match the character \*
. match any character except newline
\./ match the character .
"string" match contents of string of characters
^ match beginning of a line
$ match the end of a line
Regular Expressions in Lex

[xyz] match one character x, y, or z
(use \ to escape specials –, \, ^)

[^xyz] match any character except x, y, and z

[a-z] match one of a to z

[a-z]{{-}[aeiou]} match a lower case consonant
Regular Expressions in Lex

- $r^*$: closure (match zero or more occurrences)
- $r+$: positive closure (match one or more occurrences)
- $r?$: optional (match zero or one occurrence)
- $r\{3\}$: match three occurrences
- $r\{3,7\}$: match three to seven occurrences
- $r\{3,\}$: match three or more occurrences
Regular Expressions in Lex

\( ^r \) match only if at the start of the line

\( r\$ \) match only if at the end of the line

\( r_1 r_2 \) match \( r_1 \) then \( r_2 \) (concatenation)

\( r_1 | r_2 \) match \( r_1 \) or \( r_2 \) (union)

\( ( r ) \) grouping

\( \{ d \} \) match the regular expression defined by \( d \)

\( r_1 / r_2 \) match \( r_1 \) when followed by \( r_2 \)
Example: \texttt{wc.fl}

Regular definition

```c
{%
#include <stdio.h>
int chars = 0; int words = 0; int lines = 0;
%
word[^ \t\n\r\f\v]+%
{word} { words++; chars += strlen(yytext); }
\n{n { chars++; lines++; }
 . { chars++; }
%
int main()
{
 yylex();
 printf("lines: %8d\nwords: %8d\ncharacters: %8d\n", lines, words, chars); }
```
Example: `scanner1.fl`

```c
#include <stdio.h>

#define YYSTYPE int

int main() {
    yylex();
}
```
Functions and global variables

\textbf{yylex()} \quad \text{scanning routine}
\textbf{yytext} \quad \text{string with matching lexeme}
\textbf{yyleng} \quad \text{length of } \textbf{yytext}
\textbf{yyin} \quad \text{input file (default: stdin)}
\textbf{yyout} \quad \text{output file (default: stdout)}
\textbf{unput(c)} \quad \text{places } c \text{ back in } \textbf{yyin}
\textbf{input()} \quad \text{reads next char}
\textbf{ECHO} \quad \text{the default echo function}
Start conditions

- Flex allows conditional activation of rules
- **Start conditions** are symbolic names that can be activated and deactivated
- Start conditions declared in first part
  - %s inclusive conditions
  - %x exclusive conditions
- Use `BEGIN` to activate start condition
Example: magic.fl

```c
%s AA BB CC
%
^a {ECHO; BEGIN AA;}
^b {ECHO; BEGIN BB;}
^c {ECHO; BEGIN CC;}
\n {ECHO; BEGIN 0;}
<AA>magic {printf("first");}
<BB>magic {printf("second");}
<CC>magic {printf("third");}
.
```

Inclusive start conditions

Start condition activation

Start condition reset

Conditionally activated rules
Start conditions

\( \langle s \rangle r \) match \( r \) only if in start condition \( s \)

\( \langle s_1, s_2 \rangle r \) match \( r \) only if in any of the start condition \( s_1, s_2 \)

\( \langle * \rangle r \) match \( r \) only if in any start condition
Example: scanner2.fl

/* scanner for a toy Pascal-like language */

{%
/* need this for the call to atoi()
   and atof() below */
#include <math.h>
%

DIGIT [0-9]
ID [a-z][a-z0-9]*
Example (cont’d)

%%

{DIGIT}+    {   printf("An integer: %s (%d)\n", 
                yytext, atoi( yytext ) );
}

{DIGIT}+"."{DIGIT}+  {   printf( "A float: %s (%g)\n", 
                yytext, atof( yytext ) );
}

if|then|begin|end|procedure|function    {   printf( "A keyword: %s\n", yytext );
}
Example (cont’d)

```c
{ID} { printf( "An identifier: %s\n", yytext ); }

"+"|"-"|"*"|"/" { printf( "An operator: %s\n", yytext ); }

"{"[^}\n]*"} /* eat up one-line comments */

[ \t\n]+ /* eat up whitespace */

. { printf("Unrecognized character: %s\n", yytext ); }
```
Example (cont’d)

```c
int main( int argc, char **argv )
{
++argv, --argc; /* skip over program name */
if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
else
    yyin = stdin;
yylex();
}
```
Example `scanner3.fl`

```plaintext
%
/* definitions of manifest constants */
define LT (256)
...
%

delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+-]？{digit}+)?
Example (cont’d)

```c
%%
{ws}   { }   
if     {return IF;}     
then   {return THEN;}   
else   {return ELSE,;}  
{id}   {yylval = install_id(); return ID;}  
{number} {yylval = install_num(); 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;}   
%%
int install_id()  
{ ... }  
```

Return token to parser
Token attribute

Install `yytext` as identifier in symbol table (if not already there)
Summary of Commands

• To run Flex:

  flex program.fl
gcc lex.yy.c -lfl
./a.out

  flex -o program.c program.fl
gcc -o program program.c -lfl
./program < input > output