Lex

• The Idea

• Program Structure

• What Lex does

• Using Lex

• Lex Operators

• States in Lex
The Lex Idea

- Lex is an *implementation*…
- that uses an *extension* of RE notation
- as well as *states*
- and C/C++
Extending REs

- LEs (Lex expressions) extend REs.

- The extra notation is convenient, but...

- LEs can represent (only) the same class of languages as REs...

- so we say they have the same representational power.
Implementation

• Lex recognizes LEs

• Lex is used to build lexical scanners for compilers.

• You also need to know C or C++.
Using Lex: the General Idea

- Lex expressions (LEs) are *patterns*, like REs.
- Lex searches for strings that *match* your LEs.
- Search in an input text (a program, document...)
- Use C or C++ to say what happens when a substring of input matches a pattern.
Parts of Lex Statements

- Lex expression (LE)
- whitespace
- C statement (optional)
Parts of Lex Programs

Definitions
%%

Rules (statements)
%%

Subroutines
A Lex Program

In Unix shell-initialization (.cshrc) files, “#” means that the rest of the line is a comment. Remove them.

```%
#.* ;
```

- The first line shows there are no definitions.
- The second line does the work.
Lex Program, cont.

```%
#$.*
```

- The second line has three parts:
  - a Lex expression (# .* )
  - then whitespace
  - and the C ‘no-op’

- In the Lex expression…
  - * is the RE closure operator.
  - . stands for any character.
Lex Execution Loop

• From current character, ...

• Try to *match* all LEs against input.

• Pick the *longest* match.

• Matched material is *removed*.

• *Execute* the C (or C++) statement.

• Move to next character.
To Learn More about Lex

- *man lex*  (the online manual, on Unix)
- **Compilers: Principles, Techniques and Tools**,  
  Aho, Sethi and Ullman
- **lex & yacc**,  
  Levine, Mason and Brown
How to Use Lex: 7 easy steps

1. Create a file containing a Lex program. Let’s call it LF. Put *no* blanks ahead of an LE on its line. *Do* put a blank before the C statement.

2. Convert your Lex file to C:

   lex LF

3. The file of your resulting C program will be lex.yy.c in the current directory. Compile it with a C compiler and the lex library (2 Ls).

   cc lex.yy.c -ll
How to Use Lex: 7 easy steps

4. Your executable is now in a.out. Simply type

    a.out

    to execute your program. It’s interactive, so type in a string and <Enter>, and the system will respond. Repeat, using various strings, and watch the results. To stop, type

    <control-d>
How to Use Lex: 7 easy steps

5. Create a data file. Let's call this file DF.

6. To use your program with your data, type

   a.out < DF

   “<” introduces an input file.

   “>” redirects output to a file (here, OF):

   a.out < DF > OF

7. For C++, use flex and a C++ compiler.
Lex Operators I

* for 0 or more occurrences (like REs)

- any character except newline

- * the rest of a line

| separates alternatives (not +)

+ Plus means one or more occurrences, so a+ acts like the RE or LE aa*

? 0 or 1 of its operand (so it’s optional);
Example: $L( b? ) = \{ b, \epsilon \}$
Precedence in Lex

- Highest: *unary* operators, as elsewhere.
- ?, *, and + are the unary operators.
- Concatenation has precedence over “|”.
Precedence Example #1

- $c?d+e$ : give the strings of length 2? 3? ...

- de, cde, dde, cdde, ddde, cddde,...
Precedence Example #2

• Give the extension and simplest LE for “the binary integers with no leading zeroes”?

• Extension: \{0, 1, -1, 10, -10, 11, \ldots\}

• LE: 0|-?1(0|1)*
Lex Operators II

[ ] character choice: [aeiou] = (a|e|i|o|u). Inside brackets, special symbols lose their special properties. [*+] = (*|+).

- inside brackets: characters with consecutive ASCII codes. [a–z] = (a | b | c | … | z).
  [a–zA–Z] matches any letter.

Brackets (and parentheses) also indicate grouping.
Lex Operators III

\ protects special characters.  \+ matches a plus sign and \. matches a period.

\t stands for tab and \n for end-of-line.  Use [ \t ] + for whitespace within a line.

To match multi-line input with a single pattern, use \n in the pattern.

$/ " Dollar, slash and double quote need a backslash for normal use.
Lex Operators IV

^ inside brackets: complement.

Example: 1 or more non-whitespace characters.

\[ ^\n\t \]+

Example: quoted material

\" [^" ]*\"

Caret elsewhere anchors a pattern to the start of an input line.
**Lex Operators IV, cont.**

{ } Braces around one or two numbers indicate repetitions,

Example: 1 to 8 capital letters

\[ [A-Z] \{1, 8\} \]

{ } around a defined term invoke its definition.

If \( D \) has been defined as \([0-9]\), the pattern \( \\$\{D\}+\ .\ \{D\} \{2\} \) will match an amount of dollars and cents.
**Lex vs RE**

Although Lex has several more operators than the three of REs, most of Lex is readily converted to RE form. In these examples, we use $\Sigma = \{a, b, c, d, e\}$.

<table>
<thead>
<tr>
<th>Lex</th>
<th>RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>a*</td>
<td>a*</td>
</tr>
<tr>
<td>a+</td>
<td>aa*</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a?</td>
<td>a+□</td>
</tr>
<tr>
<td>[ace]</td>
<td>a+c+e</td>
</tr>
<tr>
<td>[a–d]</td>
<td>a+b+c+d</td>
</tr>
<tr>
<td>.</td>
<td>a+b+c+d+e</td>
</tr>
<tr>
<td>.{1,2}</td>
<td>(a+b+c+d+e)(a+b+c+d+e+□)</td>
</tr>
</tbody>
</table>
States

- *Declare* a state
- *Transition* to a new one
- *Use* one - to constrain the applicability of a pattern
The Subroutine Section

• Subroutines go after the second `%%`

• For “post-processing” after Lex finishes

```c

%%
main()
{
    yylex();
    [ any C statements]
}
```