COMS W4115
Programming Languages and Translators
Lecture 4: Regular Expressions and Lexical Analysis
February 4, 2013

Lecture Outline

1. Regular expressions
2. Lex regular expressions
3. Specifying a lexical analyzer with Lex
4. Example Lex programs
5. Creating a lexical processor with Lex
6. Lex history

1. Regular Expressions

- Regular expressions in various forms are used in many programming languages and software tools to specify patterns and match strings.
- Regular expressions are well suited for matching lexemes in programming languages.
- In formal language theory regular expressions use a finite alphabet of symbols and the operators union, concatenation, and Kleene closure. They define the regular languages.
- Unix programs like egrep, awk, and Lex extend this simple notation with additional operators and shorthands.
- The POSIX (Portable Operating System Inteface for Unix) standard defines two flavors of regular expressions for Unix systems: Basic Regular Expressions and Extended Regular Expressions.
- Perl has amazingly rich regular expressions which further extend the egrep, awk, and Lex regular expressions. Perl compatible regular expressions have been adopted by Java, JavaScript, PHP, Python, and Ruby.
- The back-referencing operator in Perl regular expressions allows nonregular languages to be recognized and makes the pattern-matching problem NP-complete.

2. Lex Regular Expressions

- The declarative language Lex has been widely used for creating many useful lexical analysis tools including lexers.
- The following symbols in Lex regular expressions have special meanings:

  `\ " . ^ $ [ ] * + ? { } | ( ) /`

  To turn off their special meaning, precede the symbol by `\`.
  - Thus, `\*` matches `*`.
  - `\\` matches `\`.
- Examples of Lex regular expressions and the strings they match.

1. `"a.*b"` matches the string `a.*b`.
2. `.` matches any character except a newline.
3. `^` matches the empty string at the beginning of a line.
4. `$` matches the empty string at the end of a line.
5. `[abc]` matches an `a`, or a `b`, or a `c`.
6. `[a-z]` matches any lowercase letter between `a` and `z`.
7. `[A-Za-z0-9]` matches any alphanumeric character.
8. `[^abc]` matches any character except an `a`, or a `b`, or a `c`.
9. `[^0-9]` matches any nonnumeric character.
10. `a*` matches a string of zero or more `a`'s.
11. `a+` matches a string of one or more `a`'s.
12. `a?` matches a string of zero or one `a`'s.
13. `a{2,5}` matches any string consisting of two to five `a`'s.
14. `(a)` matches an `a`.
15. `a/b` matches an `a` when followed by a `b`.
16. `\n` matches a newline.
17. `\t` matches a tab.

- Lex chooses the longest match if there is more than one match. E.g., `ab*` matches the prefix `abb` in `abbc`.

3. Specifying a Lexical Analyzer with Lex

- Lex is a special-purpose programming language for creating programs to process streams of input characters.
- Lex has been widely used for contructing lexical analyzers.

- A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
- The translation rules are each of the form

```
pattern        {action}
```

- Each pattern is a regular expression which may use regular definitions defined in the declarations section.
- Each action is a fragment of C-code.
- The auxiliary functions section starting with the second %% is optional. Everything in this section is copied directly to the file `lex.yy.c` and can be used in the actions of the translation rules.

4. Example Lex programs

Example 1: Lex program to print all words in an input stream

- The following Lex program will print all alphabetic words in an input stream:

```
%%
[A-Za-z]+        { printf("%s\n", yytext); }
.|\n               { }
```

- The pattern part of the first translation rule says that if the current prefix of the unprocessed input stream consists of a sequence of one or more letters, then the longest such prefix is matched and assigned to the Lex string variable `yytext`. The action part of the first translation rule prints the prefix that was matched. If this rule fires, then the matching prefix is removed from the beginning of the unprocessed input stream.
- The dot in pattern part of the second translation rule matches any character except a newline at the beginning of the unprocessed input stream. The `\n` matches a newline at the beginning of the unprocessed input stream. If this rule fires, then the character of the beginning of the unprocessed input stream is removed. Since the action is empty, no output is generated.
- Lex repeated applies these two rules until the input stream is exhausted.

Example 2: Lex program to print number of words, numbers, and lines in a file

```
        int num_words = 0, num_numbers = 0, num_lines = 0;
word     [A-Za-z]+
number   [0-9]+
%%
{word}   {++num_words;}
{number} {++num_numbers;}
\n       {++num_lines; }
.        { }
%%
int main()
{
 yylex();
 printf("# of words = %d, # of numbers = %d, # of lines = %d\n",
        num_words, num_numbers, num_lines );
}
```

Example 3: Lex program for some typical programming language tokens

- See ALSU, Fig. 3.23, p. 143.

```
%{ /* definitions of manifest constants */
  LT, LE,
```

```
  IF, 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}+)?
%%
{ws}         { }
if         {return(IF);}
else         {return(ELSE);}
{id}         {yylval = (int) installID(); return(ID);}
{number} {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP); }
"<="         {yylval = LE; return(RELOP); }
%%
int installID()
{
 /* function to install the lexeme, whose first character
    is pointed to by yytext, and whose length is yyleng,
    into the symbol table; returns pointer to symbol table
    entry */
}

int installNum() {
 /* analogous to installID */
}
```

5. Creating a Lexical Processor with Lex

- Put lex program into a file, say `file.l`.
- Compile the lex program with the command:

```
lex file.l
```

This command produces an output file `lex.yy.c`.

- Compile this output file with the C compiler and the lex library `-ll`:

```
gcc lex.yy.c -ll
```

The resulting `a.out` is the lexical processor.

6. Lex History

- The initial version of Lex was written by Michael Lesk at Bell Labs to run on Unix.
- The second version of Lex with more efficient regular expression pattern matching was written by Eric Schmidt at Bell Labs.
- Vern Paxson wrote the POSIX-compliant variant of Lex, called Flex, at Berkeley.
- All versions of Lex use variants of the regular-expression pattern-matching technology described in Chapter 3 of ALSU.
- Today, many versions of Lex use C, C++, C#, Java, and other languages to specify actions.

7. Practice Problems

1. Write a Lex program that copies a file, replacing each nonempty sequence of whitespace consisting of blanks, tabs, and newlines by a single blank.
2. Write a Lex program that converts a file of English text to "Pig Latin." Assume the file is a sequence of words separated by whitespace. If a word begins with a consonant, move the consonant to the end of the word and add "ay". (E.g., "pig" gets mapped into "igpay".) If a word begins with a vowel, just add "ay" to the end. (E.g., "art" gets mapped to "artay".)

8. Reading Assignment

- ALSU, Sects. 3.3, 3.5
- See The Lex & Yacc Page for Lex and Flex tutorials and manuals.

aho@cs.columbia.edu