Introduction to Lex

Outlines

- Review of scanner
- Introduction to lex
- Regular Expression

Lex

Lex - Lexical analyzer generator

- Lex is a tool for generating scanners.
- Lex source is a table of regular expressions and corresponding program fragments.
- Generates *`lex.yy.c`* which defines a routine *`yylex()`*

Format of the Input File

- The lex input file consists of three sections, separated by a line with just %% in it:

```
definitions
%%
rules
%%
user code
```

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Rules Section

- The rules section of the lex input contains a series of rules of the form:

```
pattern action
```

- Example:

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

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

Action

- If the action contains a `{`, the action spans till the balancing `}` is found, as in C.
- An action consisting only of a vertical bar ('|') means "same as the action for the next rule.“
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

Precedence Problem

- For example: a “<“ can be matched by “<“ and “<=“.
- The one matching most text has higher precedence.
- If two or more have the same length, the rule listed first in the flex input has higher precedence.

A Simple Example

```
%{int num_lines = 0, num_chars = 0;
%}

%%
\n     ++num_lines; ++num_chars;
.      ++num_chars;

%%
main()  {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
               num_lines, num_chars );
}
```

User Code Section

- If the Lex program is to be used on its own, this section will contain main program.
- If we leave this section empty then we will get the default main:

```
int main()
{
yylex();
return 0;
}
```

Where yylex() is the parser that is built from the rule.

Regular Expression

Regular Expression (1/3)

x match the character 'x'

. any character (byte) except newline

[xyz] a "character class"; in this case, the pattern matches either an 'x', or a 'y', or a 'z'

[abj-oZ] a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'

[^A-Z] a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

[^A-Z\n] any character EXCEPT an uppercase letter or a newline

Regular Expression (2/3)

r*	zero or more r's, where r is any regular expression
r+	one or more r's
r?	zero or one r's (that is, "an optional r")
r{2,5}	anywhere from two to five r's
r{2,}	two or more r's
r{4}	exactly 4 r's
{name}	the expansion of the "name" definition (see above)

Regular Expression (3/3)

\0 a NUL character (ASCII code 0)

\123 the character with octal value 123

\x2a the character with hexadecimal value 2a

(r) match an r; parentheses are used to override precedence (see below)

rs the regular expression r followed by the regular expression s; called "concatenation"

r|s either an r or an s

^r an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

r$ an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".