

## Introduction to Lex

22c:131

---

---

---

---

---

---

---

---

## flex - fast lexical analyzer generator

- Flex is a tool for generating scanners.
- Flex 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 flex 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 flex 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).

---

---

---

---

---

---

---

---

## User Code Section

- The user code section is simply copied to `lex.yy.c` verbatim.
- The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with the `%{}`'s removed).

---

---

---

---

---

---

---

---

## 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 );
}
```

---

---

---

---

---

---

---

---

## Regular Expression Basics

- . : matches any single character except `\n`
- \* : matches 0 or more instances of the preceding regular expression
- + : matches 1 or more instances of the preceding regular expression
- ? : matches 0 or 1 of the preceding regular expression
- | : matches the preceding or following regular expression
- [ ] : defines a character class
- () : groups enclosed regular expression into a new regular expression
- "...": matches everything within the "" literally

---

---

---

---

---

---

---

---

### Example Regular Expressions

- x match the character 'x'
- [xyz] a "character class"; in this case, the pattern matches either an 'x', 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

---

---

---

---

---

---

---

---

### Lex Regular Expression

- x|y x or y
- {i} definition of i
- x/y x, only if followed by y (y not removed from input)
- x{m,n} m to n occurrences of x
- ^ x x, but only at beginning of line
- x\$ x, but only at end of line
- "s" exactly what is in the quotes (except for "\" and following character)

A regular expression finishes with a space, tab or newline

---

---

---

---

---

---

---

---

### Example Regular Expressions

- 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)
- "[xyz]"foo" the literal string: [xyz]"foo"
- \X if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'X' (used to escape operators such as \*)

---

---

---

---

---

---

---

---

### Example Regular Expressions

- \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".

---

---

---

---

---

---

---

---

### Meta-characters

- meta-characters (do not match themselves, because they are used in the preceding reg exps):
  - ( ) [ ] { } < > + / , ^ \* | . \ \* \$ ? - %
- to match a meta-character, prefix with "\"
- to match a backslash, tab or newline, use \\, \t, or \n

---

---

---

---

---

---

---

---

### Regular Expression Examples

- an integer: 12345  
[1-9][0-9]\*
- a word: cat  
[a-zA-Z]\*
- a (possibly) signed integer: 12345 or -12345  
[-+]?[1-9][0-9]\*
- a floating point number: 1.2345  
[0-9]\*\.[0-9]\*

---

---

---

---

---

---

---

---

## Two Rules

1. lex will always match the longest (number of characters) token possible.
2. If two or more possible tokens are of the same length, then the token with the regular expression that is defined first in the lex specification is favored.

---

---

---

---

---

---

---

---

## Regular Expression Examples

•C++ comment: // call foo() here!!

`"/".*`

•white space

`[\t]+`

•English sentence: Look at this!

`([\t]+|[a-zA-Z]+)(["'"]?!)`

---

---

---

---

---

---

---

---

## Special Functions

- `yytext`
  - where text matched most recently is stored
- `yytext`
  - number of characters in text most recently matched
- `yyval`
  - associated value of current token
- `yytext`
  - append next string matched to current contents of `yytext`
- `yyless(n)`
  - remove from `yytext` all but the first n characters
- `yyinput(c)`
  - return character c to input stream
- `yywrap()`
  - may be replaced by user
  - The `yywrap` method is called by the lexical analyser whenever it inputs an EOF as the first character when trying to match a regular expression

---

---

---

---

---

---

---

---

## A Simple Example

```

%{
  int num_lines = 0, num_words = 0;
%}
word [a-zA-Z]+
eol \n
%%
{eol}      { ++num_lines; }
{word}     { ++num_words; }
.
%%
main() {
  yylex();
  printf( "# of lines = %d, # of words = %d\n",
          num_lines, num_words ); }

```

---

---

---

---

---

---

---

---

## 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 ); }
}

```

---

---

---

---

---

---

---

---

## Resources

- Google directory of lexer and parser generators.
- Flex homepage:  
<http://flex.sourceforge.net/>

---

---

---

---

---

---

---

---