Lex Project
EECS 376
Fall 2012
A Lexical Analyzer from Regular Expressions

Lex is a standard Unix tool for generating lexical analyzers from a specification written with regular expressions. A lexical analyzer is the part of an application program (such as a compiler, interpreter, database program, etc.) that recognizes tokens. A token class consists of strings that represent a meaningful unit in a language. A lexeme is an element of a token class. For example, in the C program fragment

```
main()
{
int n, square;
printf("Number?\n");
GetInt(n);
square = n * n;
printf("The square is ", square);
}
```

the lexemes are (1) `main`, (2) ((left parenthesis), (3)) (right parenthesis), (4) {, (5) `int`, (6) `n`, (7) `square`, (8) ; (semicolon), (9) `printf`, (10) `"Number?\n"`, (11) `GetInt`, (12) =, (13) *, (14) `"The square is "`, (15) }. However, some of these lexemes belong to the same token class. For example, `n` and `square` belong to the *identifier* token class, and `"Number?\n"` and `"The square is "` belong to the *string* token class. When a compiler reads a program like this, it must recognize the lexemes and identify the token classes they belong to.

In this assignment you will use lex to generate a program that converts tokens into numbers. Here is a list of the tokens for an imaginary programming language.

1. `AND`
2. `DO`
3. `ELSE`
4. `IF`
5. `NOT`
6. `OR`
7. `PROGRAM`

8. `READ`
9. `THEN`
10. `UNTIL`
11. `VARIABLES`
12. `WHILE`
13. `WRITE`
14. Type: The strings `INTEGER, FLOAT, BOOLEAN`
15. Identifier: a sequence of letters, digits, and underscores. It must begin with a letter, not end with an underscore, there should never be two consecutive underscores, and it should contain at least one digit. Thus, `AaA_98, var1, v_a_r_1` are identifiers, but not `a_, var9__8, var`.
16. String: a sequence of characters enclosed by quotation marks (`"`), but with no quotation marks in between, or a sequence of characters enclosed by single quotes (`'`), but with no single quotes in between
17. Integer constant: a plus (+) or minus sign (−) followed by a nonempty string of digits. This string of digits should not begin with a 0. (There is only one exception to this: the string 0 is an integer constant, but +0 and −0 are not.)
18. Real constant: A plus or minus sign followed by a nonempty string of digits (where 0 may not be the first digit unless the string consists of just one 0) followed by a decimal point, followed by an optional fractional part (a string of digits, possibly empty), followed by an optional exponential part consisting of the string `x10^` followed by a plus or minus sign, followed by a nonempty string of digits (not beginning with `0`). However, we specify that the following, either alone or followed by an exponential part, are not real constants:

    $$+0. \qquad +0.0 \qquad +0.00 \qquad \cdots$$

    $$-0. \qquad -0.0 \qquad -0.00 \qquad \cdots$$

    Finally, `0.` and `0.0` (with no plus or minus or exponential part) are real constants. Thus, `+0.1`, `−0.1x10^+1`, and `+1.x10^−11` are real constants, but not `0.1`, `−0.1x10^1`.
19. Semicolon: ;
20. Colon: :
21. Left parenthesis: (
22. Right parenthesis:)

23. Left brace: {

24. Right brace: }

25. Assignment: =

26. Operations: +, −, ∗, /

27. Relations: <, >, <=, >=, ==, <>

28. Comma: ,

There is another type of token that does not have a token number — the comment token. This is a sequence of characters enclosed between ($ and $). A comment may contain no occurrence of $) in between. ($) is not a comment.

How will you generate your lexical analyzer?

First check the Web link to *A Compact Guide to Lex & Yacc* by Thomas Niemann at http://epaperpress.com/lexandyacc/. This is a good source on the basics of Lex.

Next look at the CTools website. There is (or soon will be) a link to a file "Lex Source". Copy it into a file `file.l` (or whatever name you like). This is a partial lex source file. You will need to edit it and supply the missing regular expressions describing the tokens given above. Some of these are easy. For example, the first token class above has a single lexeme `AND` so you have a line

```
"AND"  fprintf(yyout,"%s 1 ",yytext);
```

in your file. This will cause your lexical analyzer to print 1 to an output file whenever the token `AND` is found in the input file. Notice that I have left a blank space where you can write this in your file `file.l`. Some of the token classes will be harder to describe. Suppose I had made identifiers (token class 15) a little less complicated – a letter followed by any number of letters or digits. I have

```
digit    [0-9]
letter   [a-zA-Z]
```

in the first part of the file (to indicate that the ASCII characters from `0` to `9` are digits and the ASCII characters from `a` to `z` and from `A` to `Z` are letters. In the second part of the file I would write

```
{letter}({letter}|{digit})*   fprintf(yyout,"%s 15 ",yytext);
```

This causes a 15 to be printed out whenever an identifier is found. (Of course you will have to give a more complicated definition of identifiers.) This is really a regular expression. We write | rather than ∪, enclose the character class names like `letter` and `digit` in brackets, and leave out the concatenation symbol ·.

Do this for every token class above. The last line of your file will be a regular expression for comments (with no `fprintf` following it). You may want to review the man pages for lex to learn some shortcut notations. When you complete your file `file.l` type (after the Unix prompt)

```
lex file.l
```

If you list your directory you will see that a file `lex.yy.c` has been created for you. This is C program for a lexical analyzer. To create an executable file `a.out` type

```
cc lex.yy.c -ll
```

This file can now be run on program in the imaginary programming language. I will put a sample program on CTools later for you to test your analyzer.

There is a program very similar to Lex called Flex. If you feel more comfortable using this program you can use it for this project.

Several more clarifications:

- Keywords are case insensitive.
- Add the error rule at the end of the rule list as follows.

  ```
  .   fprintf(yyout,"%s 99 ",yytext);
  ```

  This way, you can clearly distinguish errors from valid outputs.
- Do not rely on ”-i” option of flex for case insensitivity. If you don’t know what is ”-i” option of flex, don’t worry about it.
- Strings may contain new line. This may sound absurd, but if we don’t allow new line, then we have to forbid other control characters, and there’s no point in worrying about that. Just put no restriction on strings inside quotation marks or single quotes other than the restrictions described above.
- We will give you instructions on how to submit your program later this week.