Homework #1: CMPT-379

Distributed on Sep 13; due on Sep 29

Anoop Sarkar – anoop@cs.sfu.ca

Reading for this homework includes Chp 3 of the Dragon book. If needed, refer to:

http://tldp.org/HOWTO/Lex-YACC-HOWTO.html

Only submit answers for questions marked with †. You must use a `makefile` such that `make` compiles all your programs, and `make test` runs each program, and supplies the necessary input files. You have been provided a file `makefile.answer` which you should use as your makefile for the entire Homework #1. The stdout when you run `make test` must match `makefile.answer.out`, and stderr must match `makefile.answer.err`.

(1) The simplest way to use lex is to match an input string with a regular expression. For instance, the following is complete lex program that matches a single regexp with the entire input string and prints yes if it matches, and no otherwise.

```
%%
^(ab?)*$        { printf("yes\n"); }
^.*$            { printf("no\n"); }
%%
```

Assume we save the above text to the file `matchre.lex`. The command `flex matchre.lex` produces a C file called `lex.yy.c`, which is compiled using the command `gcc -o matchre lex.yy.c -lfl`. The main function is provided by the lex library `libfl.a` which is statically linked to the object code from `lex.yy.c` to create the final binary file `matchre`.
Write down a makefile to compile and test the above lex source. Compare this lex regexp matching with regexp matching using Perl. Here is one way to test this:

```
perl -e 'print "a" x 100000; print "\n"' | ./matchre        # using lex
perl -e 'print "yes\n" if (("a" x 100000) =~ /^(ab?)*$/);'  # using perl
```

Do you observe any difference in behaviour? Can you explain precisely why this difference exists?

(2) Lex can match a *right context* while matching a regexp by using the r_1/r_2 where the regexp r_1 will match only if the right context matches the regexp r_2. Lex can also match the *left context* with the use of *states*. The following is a lex program that matches keyword called `inputfile` as the left context so that a string following this keyword is treated differently from one that does not.

```
%s INPUT

%%

[ \t\n]+                ;
inputfile       BEGIN INPUT;
<INPUT>\".*\"   { BEGIN 0; printf("Input file:"); ECHO; printf("\n"); }
\".*\"          { ECHO; printf("\n"); }
.                ;

%%
```

Modify the above lex program to include a new left context for the keyword `outputfile` so that for the input:

```
inputfile "fileA"
outputfile "fileB"
```

Your program produces the output:

```
Input file:"fileA"
Output file:"fileB"
```

(3) Lex can be forced to perform backtracking regexp matching using the REJECT command:

```
%{
int numpat1, numpat2;
%}

%%
a+      { numpat1++; REJECT; }
a*b?    { numpat2++; REJECT; }
%%

int main () {
  yylex();
  printf("pattern a+: %d -- pattern a*b?: %d\n", numpat1, numpat2);
  return(0);
}
```

On input `aaa` the above lex program will produce the output:

```
pattern a+: 6 -- pattern a*b?: 6
```

This is because there are $\frac{n(n+1)}{2}$ substrings for a string of length n. Predict the number of pattern matches for the following input, and check by running the lex program.

```
aaa
aa
ab
```

(4) † Provide a lex program that reports the frequency of each pair of words in a text file. Each word is a sequence of non-whitespace (space and tab) characters separated by one or more whitespace characters. For example, on the input `aa bb bb aa aa bb aa aa` it should produce the following output:

```
aa aa 2
aa bb 2
bb aa 2
bb bb 1
```

Hint: you should consider the REJECT command, and also consider using the map or hashmap capabilities of the C++ STL library in the lex actions in order to store counts of word pairs.

(5) † Provide a lex program to remove all single-line and multi-line comments from C or C++ programs. Single-line comments begin with `//` and continue to the end of the line, and multi-line comments begin with `/*` and end with `*/` and have no intervening `*/`. Note that `/*/` is not a valid comment.

You must provide the output of your program on the file `testcomments.c`

Hint: Using states (see Q. 2) to match an appropriate left context can be useful, although it is possible to solve this question without using states.

(6) A lex program for a simple lexical analyzer is provided:

```
%{
#include <stdio.h>
#define NUMBER     256
#define IDENTIFIER 257
%}

%%

[0-9]+          { return NUMBER; }
[a-zA-Z0-9]+    { return IDENTIFIER; }
.               { return -1; }

%%

int main () {
  int token;
  while ((token = yylex())) {
    switch (token) {
      case NUMBER: printf("NUMBER: %s, LENGTH:%d\n", yytext, yyleng); break;
      case IDENTIFIER: printf("IDENTIFIER: %s, LENGTH:%d\n", yytext, yyleng); break;
      default: printf("Error: %s not recognized\n", yytext);
    }
  }
}
```

Modify the pattern definition of the token IDENTIFIER so that it has to start with a letter (a-z or A-Z) and followed by a (possibly empty) sequence of letters or numbers. For example, for input `12AB` the output should be:

```
NUMBER: 12, LENGTH:2
IDENTIFIER: AB, LENGTH:2
```

And for input `AB12` the output should be:

```
IDENTIFIER: AB12, LENGTH:4
```

(7) † The general architecture of a lexical analyzer for a programming language is to specify tokens in terms of patterns.
For instance, we can define a set of tokens (`T_A, T_B, T_C`) for each pattern p_i:

`T_A` p_1
`T_B` p_2
`T_C` p_3

These patterns can be written as regular expressions. For example,

`T_A` a
`T_B` abb
`T_C` a^*b^+

The lexical analysis engine should always pick the longest match possible, and in case of two patterns matching a prefix of the input of equal length, we break the tie by picking the pattern that was listed first in the specification (e.g. the token `T_B` is preferred over `T_C` for the input string *abb*, and for the same input string, token `T_A` followed by `T_C` would be incorrect).

Provide a lexical analyzer using lex for the tokens shown above. Test your lexical analyzer on the following inputs.
If the input text file contains *aaba*, the program should produce the following output token types and their values (lexemes):

```
T_C     aab
T_A     a
```

If the input text file contains *aabaaabbbbabba*, the program should produce the following output token types and their values (lexemes):

```
T_C     aab
T_C     aaabbbb
T_B     abb
T_A     a
```

If the input text file contains *aabaaabbbbsbba* which includes an illegal input character `s`, the program should print *illegal token* to `stderr` and signal an ERROR in the output stream.

```
T_C aab
T_C aaabbbb
illegal token
ERROR s
T_C bb
T_A a
```

You can optionally continue (as shown above) after detecting an error, and you can add more elaborate error reporting if you wish (such as line and character number where the error occurred).

(8) † Using the **`Decaf`** language definition as your guide, provide a lex program that is a lexical analyzer for the **`Decaf`** language.

a. To help you test your lexical analyzer, some sample **`Decaf`** programs are provided along with the output expected from your lexical analyzer.

b. Note that the token names and lexeme values should be identical to the sample output provided to you. You can also refer to the full list of token names in the file `decaf-token-names.txt`.

c. You should include a special whitespace and comment token. The whitespace token should have a lexeme value that includes all the whitespace characters. The whitespace and comment lexemes should convert the newline character into the literal string `\n` so that the line number and character number of each token can be recovered from the lexical analyzer output.

d. Provide appropriate error reporting by referring to the **`Decaf`** language specification. Include the line number and location in the line where the error was detected. A list of **`Decaf`** programs with lexical errors are provided to you (`error?.decaf` contains the **`Decaf`** program, `error?.out` contains the token output until the error is detected, and `error?.err` contains the error report).