Homework #10

Complete By: Wednesday, March 18th @ 11:59pm
Assignment: a recursive descent parser for REs
Policy: Individual work only, late work *not* accepted
Submission: electronic submission via BB

A Programming Exercise

In class we talked briefly about *recursive descent* parsing; see the lecture notes for March 4th (PPT, PDF). You can read about recursive descent parsing: **https://en.wikipedia.org/wiki/Recursive_descent_parser** . Your assignment is to define a grammar for a language, and then implement a recursive descent parser for this language; you are required to follow the recursive descent approach.

When it comes time to write your recursive descent parser, follow the approach discussed in the lecture notes, based on *nextchar( )*, *match( )*, and the throwing of exceptions. You may program in any language you want, though make sure your language supports exceptions since it makes the parser much easier to write. The exercise is meant to be a fun diversion from our usual paper-and-pencil homework.

The Assignment

The assignment is to write a recursive descent parser for regular expressions. The alphabet of the regular expressions is { 'a', 'b', 'c', 'd', '|', '*', '(', ')' }. Union is denoted by '|', and kleene star by '*'. Concatenation is denoted by placing two sub-expressions next to each other. Assume 1 line of input — i.e. a single string — and no spaces, much like the parser shown in class. Here are examples of valid regular expressions:

```
a                 (a|bc)*           a*|b*
abcd              abc*              a|b|c|d(a|b)c*
a|b               abc***            (((a)*)|((b)*))b
ab|bc|c|d         a*bc*d            (a)(b)(c)|(d)*
```

For this assignment, the EMPTY STRING is *not* a valid regular expression. You should correctly capture the precedence of the operators in your parser, which from lowest to highest is union, concatenation, and kleene. For example, the regular expression

```
ab*c|d
```

is equivalent `(a(b*)c)|(d)` when precedence is taken into account.

Getting Started

Your first order of business is to write the grammar for the language of regular expressions. For some direction, see the example we did in class on March 2nd (PPT, PDF). Or see example 2.4 on page 105 of the text; you may also want to read the author's discussion of "designing CFGs" on pages 106-107. If you follow these examples, note that your grammar will contain *left recursion*, i.e. rules of the form

```
<expr> -> <expr> ...
```

In this case you will need to eliminate this left recursion since it leads to an infinite loop in a recursive descent parser. The good news is that left recursion is easily removed, see

http://en.wikipedia.org/wiki/Left_recursion

In order to understand your parser, your grammar must be written in the form of a comment at the top of your main program file. If you need to remove left recursion, write your grammar twice in the comments: first with the left recursion in place (which is easier to read), and then again with the left recursion removed (which you need in order to write the parser).

Once you have the grammar, implement the parser using recursive descent. You must use recursive descent, which implies your main program should look something like this (this is C++):

```
int main()
{
  cout << "** please enter a regular expression: ";
  getline(cin, input);

  input = input + "$";  // add EOS marker to end:
  index = 0;            // start with first input char:

  try
  {
    RE();        // call start symbol to derive input:
    match('$');  // if we reach EOS, input is a RE!

    cout << endl;
    cout << "** Yes, input is a valid RE!" << endl;
    cout << endl;
  }
  catch (...)
  {
    cout << endl;
    cout << "** Sorry, input is NOT a valid RE..." << endl;
    cout << endl;
  }

  return 0;
}
```

Questions should be posted to piazza; please do not post your complete grammar or code, unless you post

privately (i.e. post "visible to staff only").

Optional Challenge Exercise

If you want to play with this a bit more, here's a challenging, completely optional extension to the assignment. But first, please make a copy of your program so that if you don't finish, you still have something to submit that works :-)

The challenge exercise is the following: if the input is a valid RE, echo the RE back out to the console, with the correct order of operation denoted by ( ). For example, if the input is

```
ab|c
```

then the output from your program should be

```
((ab)|c)
```

More interestingly, if the input is

```
ab*c|cd
```

then the output should be along the lines of

```
(((a(b*))c)|(cd))
```

Hint: define a global stack. Think of the recursive calls as traversing an imaginary parse tree. As you traverse, push the input string and the appropriate ( ) onto this global stack. When you're done, this stack will contain the complete output, with ( ) for showing precedence of operation.

Electronic Submission

Before you submit, make sure your name appears in a comment at the top of your main source code file, like this:

```
//
// Regular expression parser
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS301, Spring 2015: HW10
//
```

Now, submit the source code file(s) for your program. If you have multiple files,.zip together and submit the .zip file. To submit, login to http://blackboard.uic.edu/, select CS 301, click "Assignments", and submit under ***HW10*** by attaching your submission file. You may submit as many times as you want before the due date, but we only see (and grade) the last one submitted.