# HW #8-PE

**Complete By:** **Friday 3/22, 11:59pm**
**Submission:** **electronic via Blackboard**

## PE == Programming Exercise

In class we talked briefly about *recursive descent* parsing; see the lecture notes for 11 March. You can read about recursive descent parsing on Wikipedia: **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. 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 C++ or Java. The exercise is meant to be a fun diversion from our usual paper-and-pencil homework; you are required to work individually.

## 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 really this when precedence is taken into account:

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

Your first order of business is to write the grammar for the language of regular expressions. For a hint on how to get started, see example 2.4 on page 105 of the Sipser text. Your grammar must be written in the form of a comment at the top of your main program file. If your grammar contains *left recursion*, i.e. rules of the form

```
<R> -> <R> ...
```

then you will need to eliminate this recursion since it will lead 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**

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 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:

```
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 in language:

    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.

## 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))
```

A stack can be used for this; compilers build a tree, and then output the contents of the tree bottom-up. You will have to modify the recursive descent methods to either (a) use the stack explicitly, or (b) create, pass, and return tree nodes.

## Electronic Submission

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

```
//
// RE parser
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS301, Spring 2013: HW8-PE
//
```

Now, submit the source code file(s) for your program — the C++ or Java files. If you have multiple files, please .zip together and submit the .zip file. To submit, login to http://blackboard.uic.edu/, select CS 301, click "Assignments", and submit under ***HW8-PE*** 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.