Robin Miller
CSC173
Prolog Week 3-4
Writeup

To begin this assignment I examined the grammar provided as figure 2.7 in Scott's book:

```
expr → term
       →  expr add_op term
term → factor |
       → term mult_op factor
factor → id
        → number
        → - factor
        → (expr)
add_op → +
          → -
mult_op → *
           → /
```

The problem with implementing this grammar directly is that it contains left recursion which could result in an infinite loop. The left recursion has been removed in my grammar, and I also took out the statement that says a factor can go to an id since I am not using identifiers in my language. My new grammar is:

```
expr → term add_op expr
       → term
term → factor mult_op term
       → factor
factor → number
        → - factor
        → (expr)
add_op → +
          → -
mult_op → *
           → /
```

To make this an LL(1) grammar it would be necessary to remove the common prefixes as well. However, Prolog creates fail states on its own and this current grammar can be implemented with little difficulty. It will mean, however, that the parse trees will grow deeper than it would if the common prefixes were removed. I decided to keep the deeper trees since they seemed a little simpler to work with. With removing common prefixes, this grammar would require thirteen lines of code. Since I chose to allow for the common prefixes, the code for my parser is only 11 lines. I am assuming that this is allowable since one of the “best” example assignments from last year provided by the instructor also allowed for common prefixes, and because this doesn't cause any problems in functionality within Prolog.

To demonstrate the way in which my program accepts input questions, and to illustrate that my parser does work I have provided some sample input and output for select test cases:

```
?- [myparse].
% myparse compiled 0.00 sec, 3,464 bytes
true.
```

```
?- phrase(expr(Tree),[4,*,3]).
Tree = enode(tnode(fnode(4), *, tnode(fnode(3))))
```

```
?- phrase(expr(Tree),[1,*,'(',2,-,3,*,4,')']).
Tree = enode(tnode(fnode(1), *, tnode(fnode(enode(tnode(fnode(2)), -, enode(tnode(fnode(3), *,
tnode(fnode(4)))))))))
```

```
?- phrase(expr(Tree),['(','(',4,+,7,')',-,3,')','/',4]).
Tree = enode(tnode(fnode(enode(tnode(fnode(enode(tnode(fnode(4)), +, enode(tnode(fnode(...)))))), -,
enode(tnode(fnode(3))))), /, tnode(fnode(4))))
```

```
?- phrase(expr(Tree),[5,4,+,'(',25,'/',5,')']).
false.
```

```
?- phrase(expr(Tree),[-1,+,2]).
Tree = enode(tnode(fnode(-1)), +, enode(tnode(fnode(2))))
```

```
?- phrase(expr(Tree),[4,/,3]).
Tree = enode(tnode(fnode(4), /, tnode(fnode(3))))
```

```
?- phrase(expr(Tree),['(','(','(','(','(',2,+,1,')',*,2,+,1,')',*,2,+,1,')',*,2,+,1,')',*,2,+,1,')',*,2,+,1]).
Tree = enode(tnode(fnode(enode(tnode(fnode(enode(tnode(fnode(enode(..., ..., ...)), *, tnode(fnode(...))), +,
enode(tnode(fnode(...))))), *, tnode(fnode(2))), +, enode(tnode(fnode(1))))), *, tnode(fnode(2))), +,
enode(tnode(fnode(1))))
```

```
?- phrase(expr(Tree),[1,+,2,*,3,-]).
false.
```

I utilized separate node structures up to this point to illustrate how the parse tree was being created. When I began to evaluate the expressions, I found it easier to simply make a single 'node' type rather than have 'enode', 'tnode', etc. With separate node types the rules for recursively evaluating get much more complex and greater in number. This change makes the parse trees a little harder to follow, which is why I ran the examples before making this alteration. With the change to a generic node type a parse tree would look like this:

```
?- phrase(expr(Tree),[1,*,'(',2,-,3,*,4,')']).
Tree = node(node(node(1), *, node(node(node(node(node(2)), -, node(node(node(3), *, node(node(4)))))))) .
```

At this point we can notice that numbers are held within two node layers. This can be removed and a number can just be stored as itself without the use of an extra node. The parse tree then becomes:

```
?- phrase(expr(Tree),[1,*,'(',2,-,3,*,4,')']).
Tree = node(node(1, *, node(node(node(node(2), -, node(node(3, *, node(4)))))))
```

You may notice that while some numbers now appear as themselves, others are still nested within an extra node structure. This is because of the fact that an expression that goes to a single term and a term that goes to a single expression are still held within a node. When these two occur, it is safe to assume that only one piece of information will need to be stored. Therefore, the inclusion of a node for these can be removed. Now all single operations are stored as themselves without unnecessary layers of nodes. In the final version of my parser, a 'node' type is only created where the tree can split and have several branches. Single bits of information are stores as themselves. The new parse tree from my ongoing example appears as:

```
?- phrase(expr(Tree),[1,*,'(',2,-,3,*,4,')']).
Tree = node(1, *, node(2, -, node(3, *, 4)))
```

At this point, my parser was similar to the provided examples. However, I have shown all the steps and given explanations to make it clear that I understand the procedures and have worked through many versions of my parser to arrive here on my own.

Having made all of these changes, evaluating is a trivial problem. The example points out that there are only three cases, yet gives them in the incorrect order. Implementing the three procedures as given in the example will result in an infinite loop. The correct order for these procedures is as follows:

1. If the node is a number, return it.
2. If the node has three children, two numbers separated by an operator, evaluate it and return the answer
3. Otherwise, evaluate the left and right sides separately and then evaluate them together.

These three cases translated into 6 predicates in my program.

To illustrate that my program parses and evaluates correctly at this point, here is some sample input/output:

```
?- phrase(expr(Tree),[4,*,3]), evaluate(Tree, Answer).
Tree = node(4, *, 3),
Answer = 12
```

```
?- phrase(expr(Tree),['(','(',4,+,7,')',-,3,')',/,4]), evaluate(Tree,Answer).
Tree = node(node(node(4, +, 7), -, 3), /, 4),
Answer = 2
```

```
?- phrase(expr(Tree),[3,+,2,*,2,+,3]), evaluate(Tree,Answer).
Tree = node(3, +, node(node(2, *, 2), +, 3)),
Answer = 10
```

I then decided to further illustrate my understanding, and make an even clearer distinction between my own parser and the example code, I chose to utilize different types of nodes once again, which meant my evaluation predicates had to be rewritten. I made a distinction between expression nodes (enode), term nodes (tnode), and factor nodes (fnodes). The evaluation part of the program then had to be expanded into 13 predicates. To illustrate that my program still works with these changes, here is some sample input/output:

```
?- phrase(expr(Tree),[4,*,3,+,2]), evaluate(Tree,Answer).
Tree = enode(tnode(4, *, 3), +, 2),
Answer = 14 .
```

```
?- phrase(expr(Tree),['(','(',4,+,7,')',-,3,')',/,4]), evaluate(Tree,Answer).
Tree = tnode(enode(enode(4, +, 7), -, 3), /, 4),
Answer = 2
```

```
?- phrase(expr(Tree),[3,+,2,*,2,-,1]), evaluate(Tree,Answer).
Tree = enode(3, +, enode(tnode(2, *, 2), -, 1)),
Answer = 6
```

Next, I added the power operator ^ to the list of tokens that can be parsed. In doing so I made a new type of node called 'pnode'. To illustrate that the parsing for this works, here is some sample parsing:

```
?- phrase(expr(Tree),[2,*,3,^,2]).
Tree = tnode(2, *, pnode(3, ^, 2)) .
```

```
?- phrase(expr(Tree),['(',10,-,6,')',/,2,^,1]).
Tree = tnode(enode(10, -, 6), /, pnode(2, ^, 1))
```

After having done this, I had to add another two predicates to my evaluation predicates to handle the addition of the power operator. To show that the parsing and evaluation still work properly with these additions, here is some sample input/output:

```
?- phrase(expr(Tree),[2,*,3,^,2]), evaluate(Tree,Answer).
Tree = tnode(2, *, pnode(3, ^, 2)),
Answer = 18
```

```
?- phrase(expr(Tree),['(',10,-,6,')',/,2,^,1]), evaluate(Tree,Answer).
Tree = tnode(enode(10, -, 6), /, pnode(2, ^, 1)),
Answer = 2 .
```

```
?- phrase(expr(Tree),[11,*,2,+,2,^,'(',4,-,1,')']), evaluate(Tree,Answer).
Tree = enode(tnode(11, *, 2), +, pnode(2, ^, enode(4, -, 1))),
Answer = 30
```