Prefixer

A commandline program writtin in Objective-C used to to output the prefix notation of an infix expression.

Usage

```
$ prefixer [-r] inputFile
```

inputfile contains the the input expression for the program. The format of the input expression is highly restricted. All values are either single alphabetic characters or positive integers. All operators, including (and), are always separated by at least one space from other values or operators. The optional -r paramater indicates whether the expression should be reduced. See the problem discription for more information. The file should contain only one line on which the expression is notated.

Implementation

Strategy

Representing the expression

To represent a mathematical expression I have chosen for an expression tree (over a stack). A tree allows a pre-order output of the nodes which achieves prefix notation. The same applies for infix and postfix notations, these are, however, not implemented. To build the expression tree the input is split into tokens. A node in a tree is either an operand or an operator. In the case of an operator it has a left- and rightnode aswell as a value indicating operator precedence. The tree shall thus have the operator with lowest precedence (left-to-right) as it's root, *or* a single operand.
Tokens get added as nodes one by one without look-aheads. Because it doesn't use a lookahead it adds the nodes in place, to be added to and replaced by the next node (when appropriate). This allows for near linear time complexity, in the worst case it has a time complexity of O(nlogn). In this worst case scenario the parser has come upon an operand with *lower* precedence than the one before it and has the be higher in the tree. This requires traversal up in the tree untill the precedence (of the new node) is lower than the previous operator. This is done in logarithmic time.

Reducing an expression tree

To reduce an expression tree, all nodes under a given node need to be reduced before beeing able to reduce the current node. While this is not true for every expression, it applies to most and doesn't hurt expressions it doesn't apply to. If both sides are, or have reduced to, value operands we can perform a calculation to reduce the current node. When there is a variable involved most expressions can't be fully reduced any more. Expressions that can still be reduced have predictably behavior, like multiplying by 1 or 0. Check if these predictabilities apply, if so reduce according to behavior unique to the operator. The last step I have implemented invovles equality amongst the left and right operands. If this is the case, regardless whether it has variables, there is predictably behaviour, like "(x * x) / (x * x)", which can be applied to reduce the current node. Because every node is only visited once to reduce, it has a linear time complexity. However the last step I use (eqaulity) use the prefix notation to compare, which has time complexity of O(nlogn). The worst case time complexity is thus: O(nnlogn).

Choices made

Before I started my implementation I drew a few approaches on my whiteboard. This gave me insight in severals strategy's. While I saw use for a stack strategy, it didn't think of it as an easy approach to achieve prefix notation. I thought the stack approach was better suited for mere reducing an infix expression. I did not consider a PEG-parser more than to name it on the whiteboard, I judged it was overkill and time-consuming.
Before implementing the final step, I analyzed the several requirements of *'complex'* expressions and drew steps to reduce them. I did not find a reliable way that I aught worth my time that reduces the following expression: "x + 1 - x" or "(y * x) / (x * y)". This is because I do not consider commutativity.
I had little succes finding a strategy that doesn't first check normally than swaps left and right and tries again, recursively. I found many caveats about this strategy and hence didn't feel positive about it. I have marked the location where this should be implemented in the source code.
I choose to use no external code to assist with specific parts of the challenge, I feel this is necessary for the challenge to remain challenging.