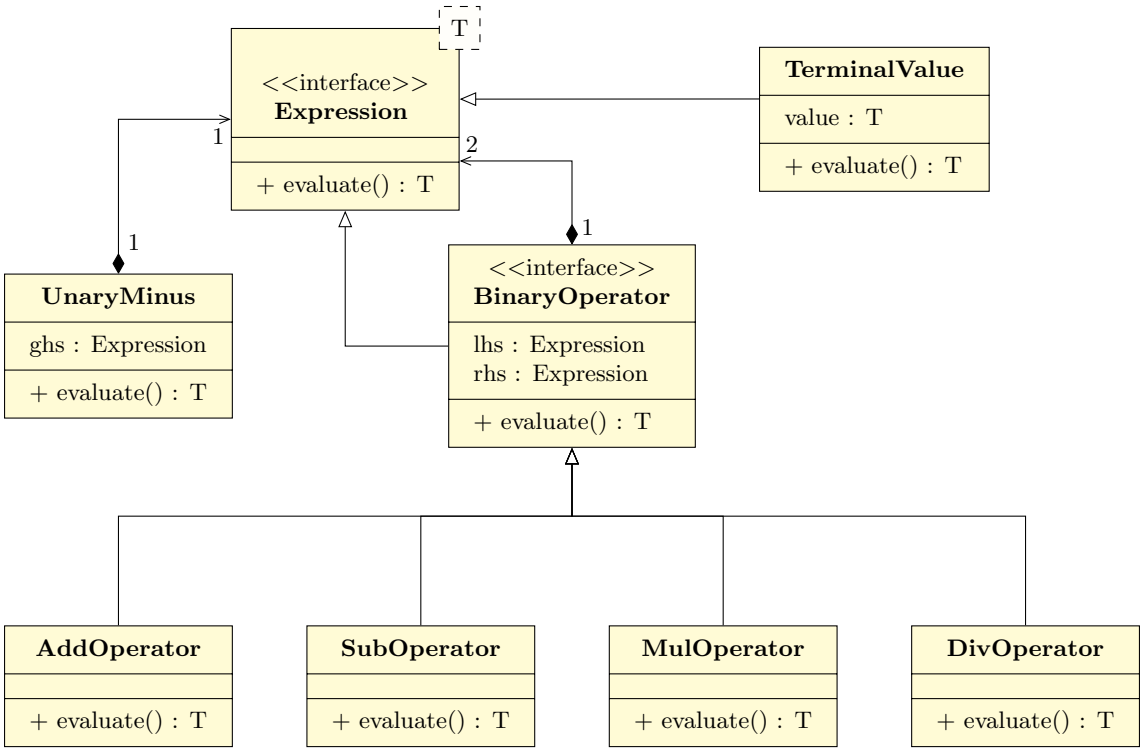
SE 464 - Lab 2

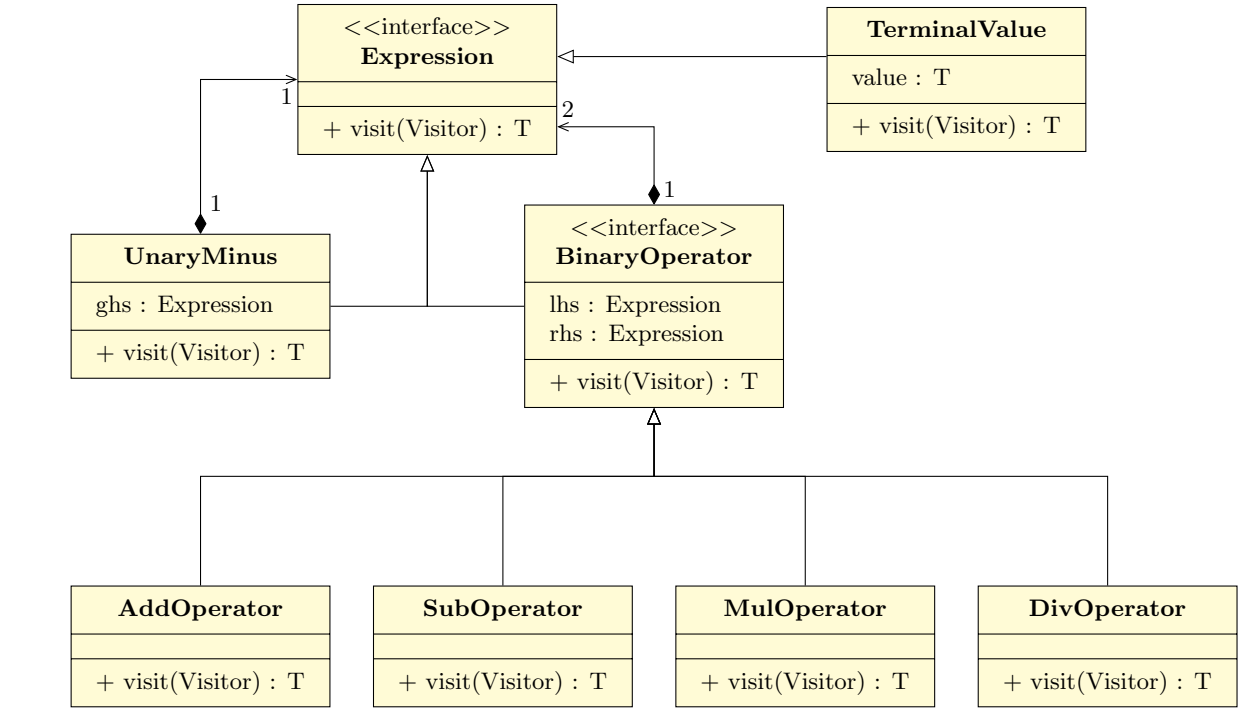
Alexander Maguire
#20396195

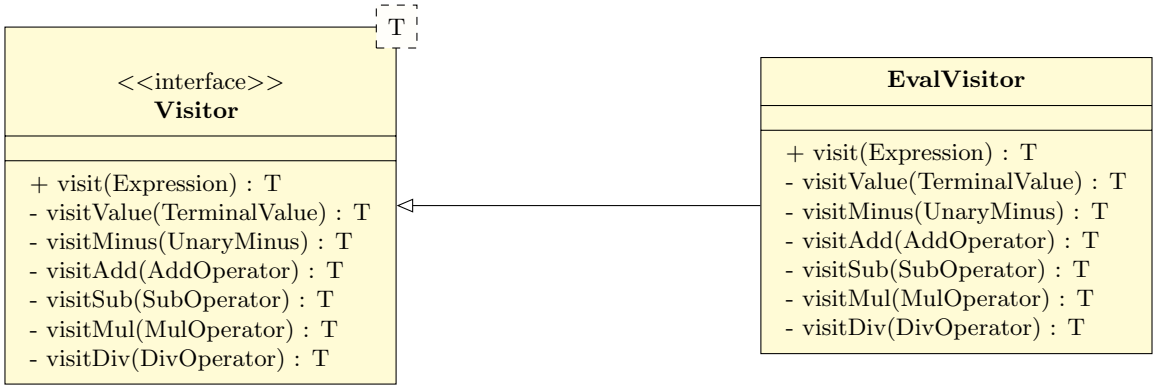
June 6, 2014

1 Interpreter Design



2 Visitor Design





3 Outside-the-box Design

Ignoring the irony of using the provided "outside-the-box" solution, one option is to use the Reverse Polish Notation, which has the distinct advantage over traditional infix notation that it is very easy to evaluate without having been parsed first.

Provided is a Haskell function to evaluate a string in RPN form – by the author's preference, the elegance of functional programming really shines in this example.

```
evalExpr :: (Fractional a, Read a) => String -> a
evalExpr stringExpr = head . foldl go [] . words $ stringExpr
    where
        go (x:y:xs) "*" = (x * y):xs
        go (x:y:xs) "/" = (y / x):xs
        go (x:y:xs) "+" = (x + y):xs
        go (x:y:xs) "-" = (y - x):xs
        go xs num       = read num:xs
```

This example has the unique property (when compared with the aforementioned solutions) of being entirely localized – it and its helper function are totally self-enclosed. This makes it very easy to reason about, and to extend in the future.

4 "Cheating" Design

We can exploit the fact that most modern programming languages do math in the same way we write it. When running our code through an interpreter, we can simply pass the original expression to the programming language's interpreter and use their implementation of the math system.

For example, in Python:

```
def evalExpr(stringExpr):
    return eval("return " + stringExpr)
```

Assuming we have already verified that *stringExpr* is valid math, this is an easy (if unsafe) solution.

5 Analysis

a) Of the analyzed designs above, all meet the basic functional requirement of evaluating arithmetic, though the RPN calculator has the notable disadvantage of having a unique syntax.

b) In terms of implementation difficulty, the "cheating" solution is a definite winner, coming in with a (provided) 2 SLOC.

The outside-box design comes in close second, with 8 SLOC, though requires the code base be written in Haskell – a language with which regrettably few are familiar.

For estimating the difficulty of implementation of the OOP solutions, we will assume the following complexity score applies:

$$L_c = 10(m_c + 1) + 7p_c \tag{1}$$

$$L = \sum_i L_c \tag{2}$$

Where c is a class, m_c is the number of methods in the class, and p_c is the number of properties (with getters and setters). We assume 10 lines of overhead per class, for definition and constructors, etc.

Since the Visitor solution is a strict superset in terms of isomorphic operations to the Interpreter solution, it is necessarily more difficult.

The final ordering of solutions in terms of ease of implementation is thus:

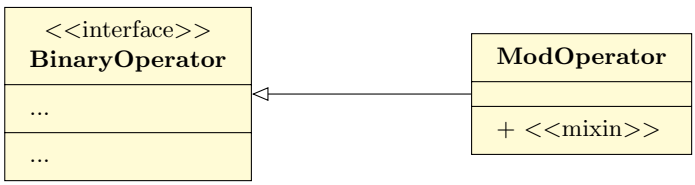
$$\text{eval()} > \text{RPN} > \text{Interpreter} > \text{Visitor}$$

c) Extending the expression grammar to allow factorials is possible in all of the presented solutions. It is trivial in the eval() solution, since there is no work to be done – the Python interpreter will take care of it for us.

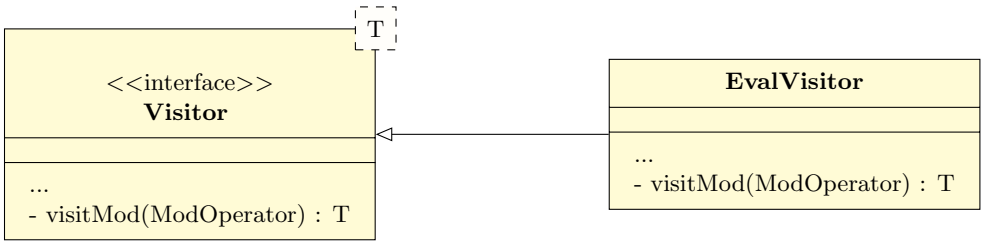
Extending the RPN solution is also very easy, and can be accomplished by adding one case to the helper function:

```
go (x:y:xs) "%" = (y `mod` x):xs
```

For evaluating the added complexity to the OOP solutions, we will need to extend the class hierarchy:



Where <<mixin>> should be replaced with the appropriate *evaluate*() or *visit*(*Visitor*) method. Additionally, the Visitor solution will need to be modified:



Again, by inspection we can see that the Visitor pattern will have the highest complexity score as defined by (1) and (2). Thus, again the ordering of solutions in terms of ease of grammar extension is as follows:

$$\text{eval()} > \text{RPN} > \text{Interpreter} > \text{Visitor}$$

d) In order to add a pretty printer, the Visitor pattern is a clear winner in terms of the least complexity added.

By complexity criterion (1), this is roughly 95 SLOC – all of the code being trivial and mostly boilerplate.

For the Interpreter pattern, 8 classes need to be touched, adding one method to each, although it is evident that these changes will not be mere boilerplate, since each node will need special cases for maintaining precedence rules (this is sidestepped in the Visitor solution since there is a localized place for the algorithm). As a rough estimate, we will argue that the three levels of precedence will add an arbitrary factor of 3 to the complexity to the necessary changes, scoring the Interpreter at 240 SLOC.

It's not immediately obvious how to extend either of the *eval*() or RPN solutions to this new behavior. The author would suggest rewriting *eval*() as follows:

```
def prettyExpr(stringExpr):
    return shuntingYard(stringExpr)
```

with an appropriate implementation of the shunting-yard algorithm for linear parsing of the expression string. From experience, the author would suggest this solution to have a complexity > 300 SLOC, but will not prove this formally.

The RPN solution will need to be entirely written, likely creating an entire class hierarchy since we can no longer exploit $foldl$ to perform our desired algorithm any further. Though implementation of classes in Haskell is simpler than in more standard OOP languages, we will use complexity criteria (1) and (2) again for a fair comparison between the languages, making this solution $\approx$ 348 SLOC (double the naïve 188 SLOC since it is necessary to *re-implement* the evaluation solution in our new class structure.)

The author thus considers the ease of implementation of the pretty-printer by solution to be:

$$\text{Visitor} > \text{Interpreter} > \text{eval()} > \text{RPN}$$

e) In addition to the Interpreter and Visitor patterns, it would also make sense to implement our future-fitness by means of a *Strategy* pattern, allowing us to lift the Visitor solution by means of object functors (or other higher-level functions).

This would allow us to easily create one-off walks of the mathematical AST, though would limit the reusability of any such components.