Building & Evaluating Symbolic Expression Trees

Brian Reavis
breavis1@uwyo.edu

COSC2030

December 3, 2010

Contents

(no help received)

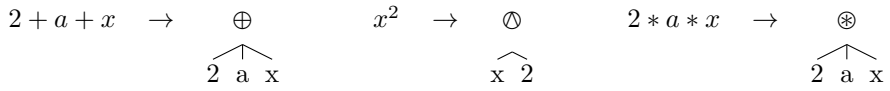
1 Introduction

Computers do exceptionally well at math when given all the numbers. However, when equations are a mixture of number and *variables*, there's no simple way to evaluate them—for it's no longer a matter of rudimentary arithmetic; it's a matter of carefully applying mathematical rules to "massage" an expression/equation into its final form. Also, with expressions containing variables, one can perform a whole variety of operations: differentiation, integration, factoring, etc to arrive at an entirely-new expression. This type of evaluation that keeps variables intact is known as symbolic evaluation.

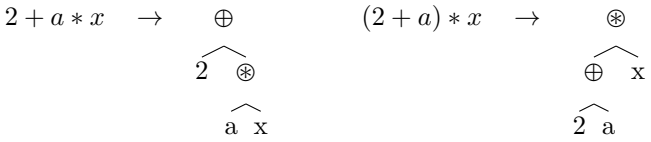
I developed software to parse expressions and perform symbolic evaluation and differentiation of them (as one would do by hand in a Calculus class). I think it'd be really interesting to eventually use these symbolic expression trees and methods with evolutionary computing to derive equations that describe difficult-to-describe phenomena. The "fitness" of a tree could be evaluated using two crucial factors: (1) the complexity of the tree, with low-complexity being favored so that it doesn't grow into something gigantic, and (2) its actual ability to describe the situation. Over time, one could mutate elements of the tree. Each type of object in the tree would have differing probabilities of mutation. For instance, constant coefficients should probably be more likely to change than operators (*i.e.* changing a "+" sign to multiplication "∗").

1.1 Parsing

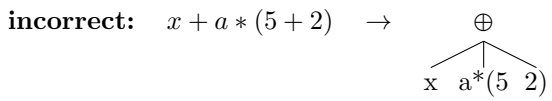
In order to evaluate an expression, one must first parse the equation string (*e.g.* `2+a+x`) into a tree structure that the computer can work with. For each type of operator encountered in the string, a tree is constructed with the operator itself at the root and the terms as the children of the tree, as follows:



Now suppose we have an expression that is a mixture of operators. Being careful to not violate order-of-operation rules, the tree becomes a multi-level tree:

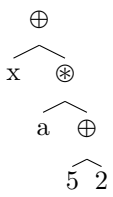


The actual implementation of this gets a bit tricky when you must account for the possibility of many nested parentheses—because you don't want the operators within the parentheses affecting the splitting and construction of the current level. For example, if one has an equation like `x+a*(5+2)`, the naïve parser would likely produce:



However, when the parentheses are properly accounted for (by tracking opening and closing parentheses using a stack), the proper tree looks like:

correct: $x + a * (5 + 2) \quad \rightarrow$



At the moment, the program I developed relies on having no implied operators—so one must be pretty explicit. An expression like `abx` will be parsed as if "abx" is a single variable. The proper way to write this expression would be `a*b*x`. Here are some examples of expressions properly represented using no implied operators:[1]

$((x+2)^x)^{(ax)}$	$\rightarrow$	`((x+2)^x)^(a*x)`
$x^2 + 3x^2$	$\rightarrow$	`x^2+3*x^2`
$6xx + 2x + 5$	$\rightarrow$	`6*x*x+2*x+5`
x^2	$\rightarrow$	`x^2`
$(2x + 5x)^2 + (7x)^2$	$\rightarrow$	`(2*x+5*x)^2+(7*x)^2`
$(2x + 5x)^2(7x)^3$	$\rightarrow$	`(2*x+5*x)^2*(7*x)^3`
$2x^2 + (6x + 5)^2$	$\rightarrow$	`2*x^2+(6*x+5)^2`
e^2	$\rightarrow$	`e^2`
$e^{(2x)}$	$\rightarrow$	`e^(2*x)`
$e^{\left(2x^3+x\right)}$	$\rightarrow$	`e^(2*x^3+x)`
$xe^{\left(5x^2\right)}$	$\rightarrow$	`x*e^(5*x^2)`
$2x^2yx^2$	$\rightarrow$	`2*x^2*y*x^2`
x^5xx^2	$\rightarrow$	`x^5*x*x^2`
$(x+2)^x ax$	$\rightarrow$	`((x+2)^x)*(a*x)`
$(x+2)xax$	$\rightarrow$	`((x+2)*x)*(a*x)`

One current limitation of the parser is not being able to recognize certain functions and notations. Using any of the following operations will result in undesired behavior:

$\sin(x)$, $\cos(x)$, ...	Trigonometric functions.
$\log(x)$, $\ln(x)$, ...	Logarithmic functions.
$\langle x^2, x, 3\rangle$	Vector functions.

1.2 Evaluation

Once an expression is parsed into its tree form, one of the most fundamental and useful operations that can be performed is merely evaluating it: determining the resulting expression when a variable in the tree is substituted by a number, or other expression. For example, if $f(x) = x^2$, then $f(3) = 9$.

To do this, the program recursively traverses the tree and looks for variables. For each one that is encountered, if variable name is identical to the one that's having a value provided for it, it is replaced by that value—with the value either being a provided number, variable, or expression tree. Once these substitutions are complete, the tree undergoes optimization (described in §1.2.1 below) and has basic arithmetic performed.

[1] These expressions are used as the test cases for the program.

The evaluation routine can be found in `expr_tree_evaluate.h`.

1.2.1 Optimization Rules

A huge part of math is the basic process of working an expression into a simpler, more-digestable form. This process can't be skipped over—otherwise, operations like differentiation would produce very unsightly and incomprehensible results (which defeats the very purpose of this project). I programmed the following methods (each of which is fully functional):

Term Consolidation When identical terms are added together, they can be expressed using a single term multiplied by a coefficient (the number of times the term is repeated).
(See `expr_tree_optimize_consolidate_terms.h`)

$x + 2x + x + x \to 5x$

Power Consolidation When a term or expression is raised to multiple powers, the power of the term/expression can be reduced to the product of the powers. (See `expr_tree_optimize_consolidate_powers.h`)

$((2 + x)^x)^x \to (2 + x)^{x^2}$

Factor Consolidation When a identical terms (possibly being raised to a power) are multiplied together, the terms can be reduced to a single term, raised to the sum of the powers of each.
(See `expr_tree_optimize_consolidate_factors.h`)

$x^2 x^2 \to x^4$

Scalar Consolidation When multiple constants appear under a multiplication operation, they can be joined to form a single constant. (See `expr_tree_optimize_consolidate_scalars.h`)

$2 * 5 * x \to 10 * x$

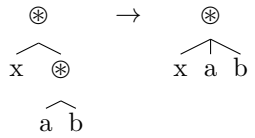
Constants Raised to Powers Whenever a constant is grouped with other variables in a multiplication operation, and all of this is raised to a constant power, the power can be applied to the constant base and moved out from the power operation. (See `expr_tree_optimize_const_prod_pow.h`)

$(2x)^2 \to 4x^2$

Distributing Constants If a sum of variables are being multiplied by a constant, the constant can be applied to each variable of the sum. (See `expr_tree_optimize_distribute_constants.h`)

$4\,(2 + x) \to 8 + 4x$

Redundant Operators Occasionally, operations on the tree will produce unnecessary or redundant tree levels. In order to not have the tree growing needlessly, nodes having child trees using the same operator will have their children moved up a level in the tree. Also, if a node only has one child, the child will take the place of the tree it's a part of—because all the primitive operators (+,−,∗, etc) need two or more operands to produce a different result. (See `expr_tree_optimize_redundant_operators.h`)



The rules are applied to each node of the tree until either (1) optimization produces no further changes, or (2) a maximum iteration limit is reached.

1.3 Differentiation

Computing the rate of change of a function at point can be very useful in a variety of cases, so I decided to build in that feature. To compute derivatives, the program follows the basic differentiation rules (given the tree's operator):

- $\oplus$ Differentiate each child individually and add the computed derivatives together.
- $\ominus$ Differentiate each child individually and subtract the computed derivatives from the first.
- $\circledast$ For each child, calculate the derivative and multiply it by the other children. Add each of these results together.
- $\oslash$ With powers, a couple cases need to be considered:
  - If the base is e, multiply the exponential by the derivative of its power.
  - If the power is a constant (e.g. 2, 5, 8, etc), multiply the base by the power and subtract 1 from the power.
  - If the power is not constant (e.g. x), this is quite a bit more difficult and hasn't been implemented yet. Support for functions (log in particular) must be added first.

At a glance, differentiation is supported for the following class of functions:

$2 * x^2 + 4x$	Basic polynomials
$(2x + 3)^5$	Functions raised to constant powers.
e^{2x^2+4x}	Exponentials
$(x^3 + 2)xe^{2x}$	Products

Some cases that currently aren't supported include:

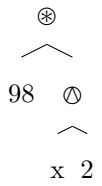
x^{2x}	Variables / expressions with non-constant powers (if the base is not e).
$\sin(x)$, $\cos(x)$, ...	Trigonometric functions.
$1/x$	Quotients.
$\log(x)$	Logarithms.

2 Odds and Ends

2.1 Tree Typesetting

This writeup was built using LaTeX, a common document preparation system that allows one to write and assemble documents using plain text along with text commands. Using the `qtree` package, trees can easily be typeset. Here is how to specify a tree with `qtree` and the result:

```
\Tree [.$\varoast$ 98 [.$\varowedge$ x 2 ] ]
```



The source code that generates `qtree` tree markup can be found in `expr_latex.h`.

2.2 Notes

- I implemented expression trees using a polymorphic approach that I'm not particularly happy about, in hindsight. Each node in an expression can either be an `ExpressionTree` or `ExpressionVariable`—both of which inherit from `ExpressionObject` (which defines common methods, like `differentiate()`, `evaluate()`, etc.) This has its advantages, in that all objects contain common fundamental methods which can be called regardless of whether it's an `ExpressionTree` or `ExpressionVariable`. Also, with each operating fairly differently, it makes sense to separate the logic.

  The primary disadvantage lies in the heavy use of dynamic casting (`dynamic_cast<type*>(*obj)`) to determine type of an object in cases where the type represented by the `ExpressionObject` *does* matter (such as in the optimization routines). These dynamic casts are slow (because many runtime checks are performed) and visually clutter the code.

  A possible solution is to simply have expression trees and variables represented by a single class (i.e. an `ExpressionObject` class that is not abstract as it is now). One could have an `objectType` member that specifies if the object represents a tree or variable. While there would be class members that aren't applicable in certain cases (like `children` if the object is a variable), the code would be much more more clear without a quagmire of dynamic casting.

- Integration needs to be added.

- Eventually it would be neat to add the ability to express equations, and be able to solve for variables by cleverly rearranging the tree.

3 Test Cases

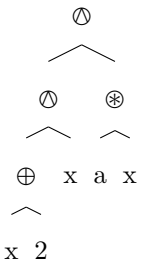
$f(x)$ The original function.

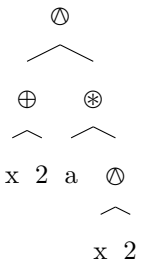
$g(x)$ The original function, having been optimized.

$\frac{d}{dx}g(x)$ The optimized differentiated function.

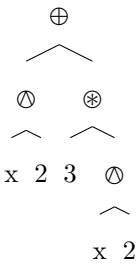
$\varnothing$ Indicates the expression is invalid or could not be computed.

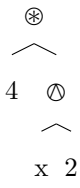
(1) $f(x) = ((x+2)^x)^{(ax)}$ $g(x) = (x+2)^{(ax^2)}$ $\frac{d}{dx}g(x) = \varnothing$

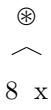




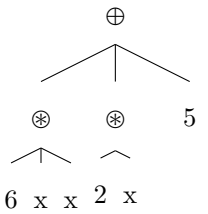
(2) $f(x) = x^2 + 3x^2$ $g(x) = 4x^2$ $\frac{d}{dx}g(x) = 8x$

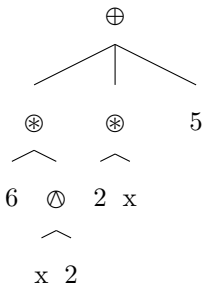


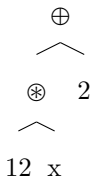




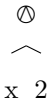
(3) $f(x) = 6xx + 2x + 5$ $g(x) = 6x^2 + 2x + 5$ $\frac{d}{dx}g(x) = 12x + 2$

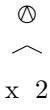


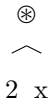




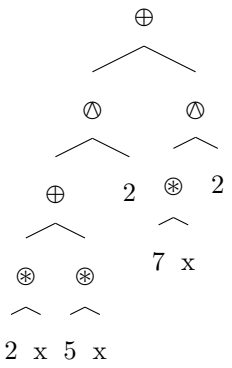
(4) $f(x) = x^2$ $g(x) = x^2$ $\frac{d}{dx}g(x) = 2x$

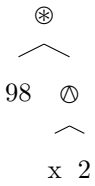


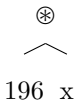




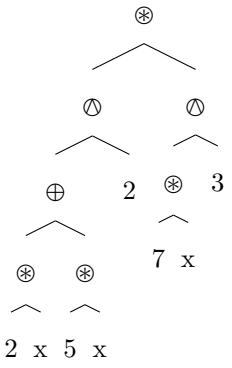
(5) $f(x) = (2x + 5x)^2 + (7x)^2$ $g(x) = 98x^2$ $\frac{d}{dx}g(x) = 196x$

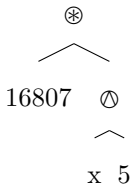


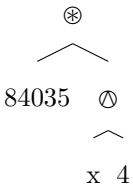




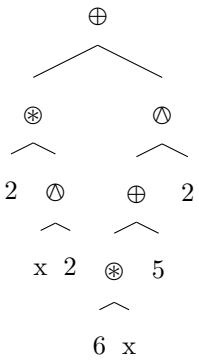
(6) $f(x) = (2x + 5x)^2(7x)^3$ $g(x) = 16807x^5$ $\frac{d}{dx}g(x) = 84035x^4$

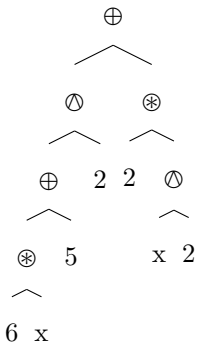


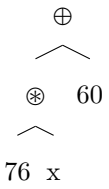




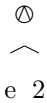
(7) $f(x) = 2x^2 + (6x + 5)^2$ $g(x) = (6x + 5)^2 + 2x^2$ $\frac{d}{dx}g(x) = 76x + 60$

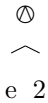






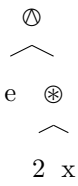
(8) $f(x) = e^2$ $g(x) = e^2$ $\frac{d}{dx}g(x) = 0$



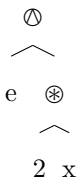


0

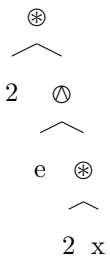
(9) $f(x) = e^{(2x)}$



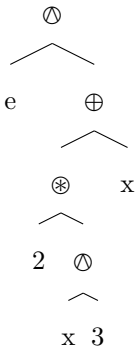
$g(x) = e^{(2x)}$



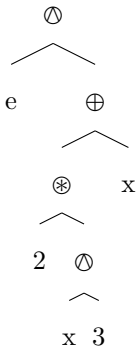
$\frac{d}{dx}g(x) = 2e^{(2x)}$



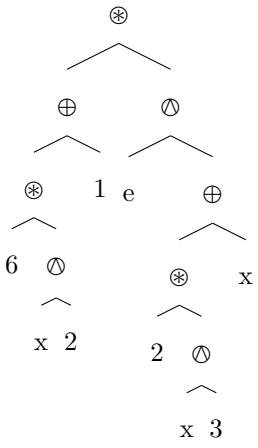
(10) $f(x) = e^{\left(2x^3+x\right)}$



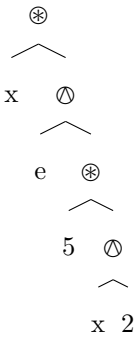
$g(x) = e^{\left(2x^3+x\right)}$



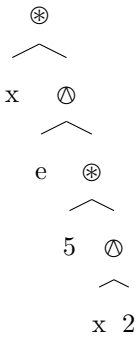
$\frac{d}{dx}g(x) = (6x^2+1)e^{\left(2x^3+x\right)}$



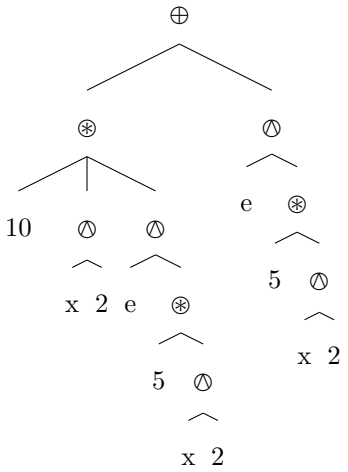
(11) $f(x) = xe^{\left(5x^2\right)}$



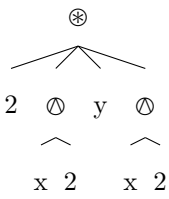
$g(x) = xe^{\left(5x^2\right)}$

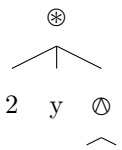


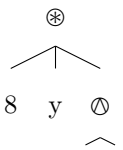
$\frac{d}{dx}g(x) = 10x^2e^{\left(5x^2\right)} + e^{\left(5x^2\right)}$



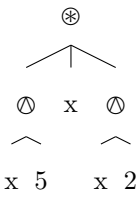
(12) $f(x) = 2x^2yx^2$ $g(x) = 2yx^4$ $\frac{d}{dx}g(x) = 8yx^3$

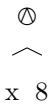


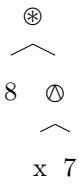




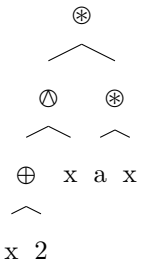
(13) $f(x) = x^5xx^2$ $g(x) = x^8$ $\frac{d}{dx}g(x) = 8x^7$

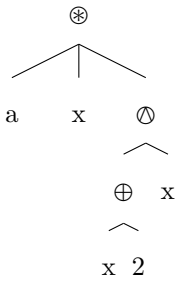




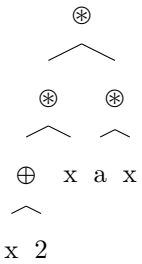


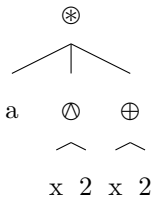
(14) $f(x) = (x+2)^x ax$ $g(x) = ax(x+2)^x$ $\frac{d}{dx}g(x) = \varnothing$

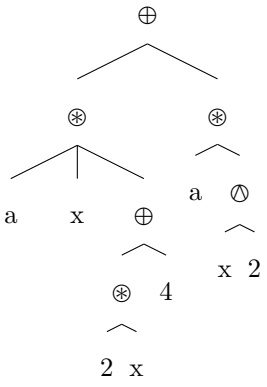




(15) $f(x) = (x+2)xax$ $g(x) = ax^2(x+2)$ $\frac{d}{dx}g(x) = ax(2x+4) + ax^2$





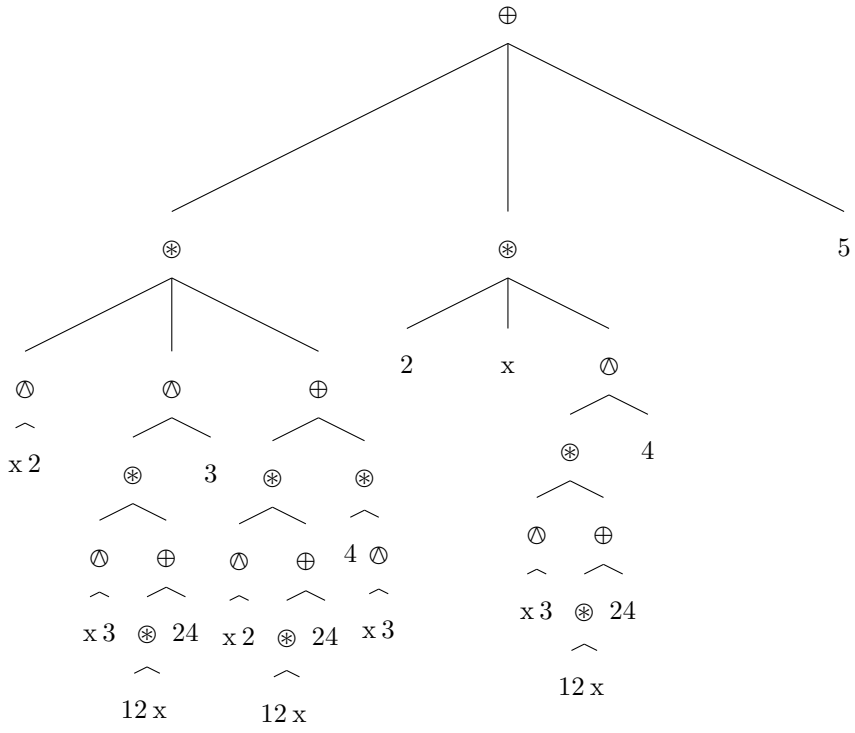


3.1 Large Test

Just for kicks, I tested out differentiation on a large expression that would be fairly tedious to do by hand (one that involves many applications of the chain rule and product rule):

$$f(x) = x^2\big((x+2)x^3\big)^4 + 5x$$

$$f'(x) = x^2\big(x^3(12x+24)\big)^3(x^2(12x+24)+4x^3) + 2x\big(x^3(12x+24)\big)^4 + 5$$



As one can see, there are obviously some further optimizations that can be done on the tree. The needed optimization that strikes me first is the necessity of applying constant powers to children if the children are being multiplied together (i.e. $\big(x^5(x+2)^4\big)^2 \to x^{10}(x+2)^8$). Furthermore, expanding polynomials under constant powers would be beneficial.

4 Evaluation Tests

$f(x) = (x+2)^{(ax^2)}$
$f(0) = 1$
$f(1) = 3^a$
$f(3) = 5^{(9a)}$
$f(x^2) = \big(x^2+2\big)^{(ax^4)}$

$f'(x) = \varnothing$

$f(x) = 4x^2$
$f(0) = 0$
$f(1) = 4$
$f(3) = 36$
$f(x^2) = 4x^4$

$f'(x) = 8x$
$f'(0) = 0$
$f'(1) = 8$
$f'(3) = 24$
$f'(x^2) = 8x^2$

$f(x) = 16807x^5$
$f(0) = 0$
$f(1) = 16807$
$f(3) = 4.0841 \times 10^6$
$f(x^2) = 16807x^{10}$

$f(x) = 6x^2 + 2x + 5$
$f(0) = 5$
$f(1) = 13$
$f(3) = 65$
$f(x^2) = 2x^2 + 6x^4 + 5$

$f'(x) = 84035x^4$
$f'(0) = 0$
$f'(1) = 84035$
$f'(3) = 6.80684 \times 10^6$
$f'(x^2) = 84035x^8$

$f'(x) = 12x + 2$
$f'(0) = 2$
$f'(1) = 14$
$f'(3) = 38$
$f'(x^2) = 12x^2 + 2$

$f(x) = (6x + 5)^2 + 2x^2$
$f(0) = 25$
$f(1) = 123$
$f(3) = 547$
$f(x^2) = \left(6x^2 + 5\right)^2 + 2x^4$

$f(x) = x^2$
$f(0) = 0$
$f(1) = 1$
$f(3) = 9$
$f(x^2) = x^4$

$f'(x) = 76x + 60$
$f'(0) = 60$
$f'(1) = 136$
$f'(3) = 288$
$f'(x^2) = 76x^2 + 60$

$f'(x) = 2x$
$f'(0) = 0$
$f'(1) = 2$
$f'(3) = 6$
$f'(x^2) = 2x^2$

$f(x) = e^2$
$f(0) = e^2$
$f(1) = e^2$
$f(3) = e^2$
$f(x^2) = e^2$

$f(x) = 98x^2$
$f(0) = 0$
$f(1) = 98$
$f(3) = 882$
$f(x^2) = 98x^4$

$f'(x) = 0$
$f'(0) = 0$
$f'(1) = 0$
$f'(3) = 0$
$f'(x^2) = 0$

$f'(x) = 196x$
$f'(0) = 0$
$f'(1) = 196$
$f'(3) = 588$
$f'(x^2) = 196x^2$

$f(x) = e^{(2x)}$
$f(0) = 1$
$f(1) = e^2$
$f(3) = e^6$
$f(x^2) = e^{\left(2x^2\right)}$

$$f'(x) = 2e^{(2x)}$$
$$f'(0) = 2$$
$$f'(1) = 2e^2$$
$$f'(3) = 2e^6$$
$$f'(x^2) = 2e^{(2x^2)}$$

$$f(x) = e^{(2x^3+x)}$$
$$f(0) = 1$$
$$f(1) = e^3$$
$$f(3) = e^{57}$$
$$f(x^2) = e^{(2x^6+x^2)}$$

$$f'(x) = (6x^2+1)e^{(2x^3+x)}$$
$$f'(0) = 1$$
$$f'(1) = 7e^3$$
$$f'(3) = 55e^{57}$$
$$f'(x^2) = (6x^4+1)e^{(2x^6+x^2)}$$

$$f(x) = xe^{(5x^2)}$$
$$f(0) = 0$$
$$f(1) = e^5$$
$$f(3) = 3e^{45}$$
$$f(x^2) = x^2e^{(5x^4)}$$

$$f'(x) = 10x^2e^{(5x^2)} + e^{(5x^2)}$$
$$f'(0) = 1$$
$$f'(1) = 11e^5$$
$$f'(3) = 91e^{45}$$
$$f'(x^2) = 10x^4e^{(5x^4)} + e^{(5x^4)}$$

$$f(x) = 2yx^4$$
$$f(0) = 0$$
$$f(1) = 2y$$
$$f(3) = 162y$$
$$f(x^2) = 2yx^8$$

$$f'(x) = 8yx^3$$
$$f'(0) = 0$$
$$f'(1) = 8y$$
$$f'(3) = 216y$$
$$f'(x^2) = 8yx^6$$

$$f(x) = x^8$$
$$f(0) = 0$$
$$f(1) = 1$$
$$f(3) = 6561$$
$$f(x^2) = x^{16}$$

$$f'(x) = 8x^7$$
$$f'(0) = 0$$
$$f'(1) = 8$$
$$f'(3) = 17496$$
$$f'(x^2) = 8x^{14}$$

$$f(x) = ax(x+2)^x$$
$$f(0) = 0$$
$$f(1) = 3a$$
$$f(3) = 375a$$
$$f(x^2) = ax^2\left(x^2+2\right)^{\left(x^2\right)}$$

$$f'(x) = \varnothing$$

$$f(x) = ax^2(x+2)$$
$$f(0) = 0$$
$$f(1) = 3a$$
$$f(3) = 45a$$
$$f(x^2) = ax^4(x^2+2)$$

$$f'(x) = ax(2x+4) + ax^2$$
$$f'(0) = 0$$
$$f'(1) = 7a$$
$$f'(3) = 39a$$
$$f'(x^2) = ax^2(2x^2+4) + ax^4$$

5 Source Code

5.1 main.cpp

```
// Brian Reavis
```

```
// COSC2030
// 11/17/2010
// Symbolic Function Differentiator

#include <stdio.h>
#include <iostream>
#include <fstream>
#include "expr.h"
using namespace std;

int main(){
    ifstream input;
    ofstream output;
    ofstream output_eval;
    ofstream output_eqns;
    ofstream output_large;
    input.open("tests.txt", ifstream::in);
    if (!input.is_open()){
        cerr << "Could not open tests.txt" << endl;
        exit(1);
    }
    output.open("tests_results.tex", ifstream::out);
    if (!output.is_open()){
        cerr << "Could not open tests_results.tex for writing." << endl;
        exit(1);
    }
    output_eval.open("tests_results_eval.tex", ifstream::out);
    if (!output_eval.is_open()){
        cerr << "Could not open tests_results_eval.tex for writing." << endl;
        exit(1);
    }
    output_eqns.open("eqns.tex", ifstream::out);
    if (!output_eqns.is_open()){
        cerr << "Could not open eqns.tex for writing." << endl;
        exit(1);
    }
    output_large.open("test_large.tex", ifstream::out);
    if (!output_large.is_open()){
        cerr << "Could not open test_large.tex for writing." << endl;
        exit(1);
    }

    string line;
    int n = 0;
    while (getline(input, line)){
        ExpressionObject *expr = ExpressionObject::parse(line, false);
        if (!expr){
            cout << "Failed to parse: " << line << endl;
            continue;
        }
        ++n;

        output << "% --- " << line << " ---" << endl;
        output << "\\item[(" << n << ")]" << endl;

        // original expression
        output << "\\parbox[t]{\\treeboxwidth}{" << endl;
        output << "\t\\noindent$f(x)=" << expr->toLaTeX() << "$" << endl;
        output << "\t\\vspace{1em}\\newline" << expr->toLaTeXTree() << endl;
        output << "}" << endl;

        output_eqns << "$" << expr->toLaTeX() << "$ &  $\\rightarrow$ & \\verb|" << line << "
            |\\\\" << endl;

        // optimized expression
        expr = expr->optimize();
        output << "\\parbox[t]{\\treeboxwidth}{" << endl;
        output << "\t\\noindent$g(x)=" << expr->toLaTeX() << "$" << endl;
```

```
        output << "\t\\vspace{1em}\\newline" << expr->toLaTeXTree() << endl;
        output << "}" << endl;

        output_eval << "% ---- " << line << " ----" << endl;
        output_eval << "\\begin{align*}" << endl;
        output_eval << "f(x) &= " << expr->toLaTeX() << "\\\\" << endl;
        output_eval << "f(0) &= " << expr->evaluate(0.0, "x")->toLaTeX() << "\\\\ " << endl;
        output_eval << "f(1) &= " << expr->evaluate(1.0, "x")->toLaTeX() << "\\\\ " << endl;
        output_eval << "f(3) &= " << expr->evaluate(3.0, "x")->toLaTeX() << "\\\\ " << endl;
        output_eval << "f(x^2) &= " << expr->evaluate(ExpressionObject::parse("x^2"), "x")->
            toLaTeX() << "\\\\" << endl;
        output_eval << "\\end{align*}" << endl;

        // differentiated expression

        ExpressionObject *expr_diff = expr->differentiate("x");
        if (expr_diff){
            output << "\\parbox[t]{\\treeboxwidth}{" << endl;
            output << "\t\\noindent$\\frac{d}{dx}g(x)=" << expr_diff->toLaTeX() << "$" <<
                endl;
            output << "\t\\vspace{1em}\\newline" << expr_diff->toLaTeXTree() << endl;
            output << "}" << endl;

            output_eval << "\\begin{align*}" << endl;
            output_eval << "f'(x) &= " << expr_diff->toLaTeX() << "\\\\" << endl;
            output_eval << "f'(0) &= " << expr_diff->evaluate(0.0, "x")->toLaTeX() << "\\\\ 
                " << endl;
            output_eval << "f'(1) &= " << expr_diff->evaluate(1.0, "x")->toLaTeX() << "\\\\" 
                 << endl;
            output_eval << "f'(3) &= " << expr_diff->evaluate(3.0, "x")->toLaTeX() << "\\\\ 
                " << endl;
            output_eval << "f'(x^2) &= " << expr_diff->evaluate(ExpressionObject::parse("x^2
                "), "x")->toLaTeX() << "\\\\" << endl;
            output_eval << "\\end{align*}" << endl;
        }else{
            output << "\\parbox[t]{\\treeboxwidth}{" << endl;
            output << "\t\\noindent$\\frac{d}{dx}g(x)=\\varnothing$" << endl;
            output << "}" << endl;

            output_eval << "\\begin{align*}" << endl;
            output_eval << "f'(x) &= \\varnothing\\\\" << endl;
            output_eval << "\\end{align*}" << endl;
            cout << "Could not differentiate: " << line << endl;
        }

        cout << line << endl;

    }

    ExpressionObject *large = ExpressionObject::parse("5*x+x^2*((x+2)*x^3)^4");
    output_large << "$f(x)=" << large->toLaTeX() << "$" << endl << endl;
    ExpressionObject *large_diff = large->differentiate("x");
    output_large << "$f'(x)=" << large_diff->toLaTeX() << "$" << endl << endl;
    output_large << large_diff->toLaTeXTree() << endl;

    output_large.close();
    output_eqns.close();
    output_eval.close();
    output.close();
    input.close();

    return 0;
}
```

5.2 expr.h

```
/*
```

```
 *  expr.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

// Features:
// - terms automatically sorted by complexity
// -

#ifndef _EXPR_H
#define _EXPR_H

#include <string>
#include <stack>
#include <list>
#include <vector>
#include <sstream>
#include <math.h>
#include <set>
#include <algorithm>
using namespace std;

#define TAB "    "

/**
 * EXPRESSION OBJECT
 * the abstract class lining out the methods
 * that ExpressionVariable and ExpressionTree share.
 */
class ExpressionObject {
public:
    virtual ExpressionObject* clone() = 0;

    // parses a mathematical expression given as a string. if the
    // expression is cannot be parsed, the tree that is returned will
    // have tree.valid set to false.
    static ExpressionObject* parse(string expr, bool optimizeResult);
    static ExpressionObject* parse(string expr);

    virtual void print(ostream &stream) = 0;
    virtual void print(ostream &stream, int level) = 0;
    virtual string toLaTeX() = 0;
    virtual string toLaTeXTree() = 0;
    virtual string toLaTeXTree(int level) = 0;

    virtual ExpressionObject* evaluate(double value, string variable) = 0;
    virtual ExpressionObject* evaluate(ExpressionObject *value, string variable) = 0;
    virtual ExpressionObject* differentiate(string variable) = 0;
    virtual ExpressionObject* optimize() = 0;

    static int  compare(ExpressionObject *obj1, ExpressionObject *obj2);
    static bool sortcompare(ExpressionObject *obj1, ExpressionObject *obj2);
    static bool rsortcompare(ExpressionObject *obj1, ExpressionObject *obj2);
    static bool equal(ExpressionObject *obj1, ExpressionObject *obj2);

    // returns an expression that is marked as 'invalid'
    // (i.e. it couldn't be parsed successfully)
    static ExpressionObject* invalid(){
        return NULL;
    };
};

#include "expr_helpers.h"

#include "expr_operator.h"
```

```
#include "expr_variable.h"
#include "expr_tree.h"
#include "expr_parse.h"
#include "expr_latex.h"


bool operator == (const ExpressionTree &obj1, const ExpressionVariable &obj2) { return false
    ; }
bool operator != (const ExpressionTree &obj1, const ExpressionVariable &obj2) { return true;
     }
bool operator == (const ExpressionVariable &obj1, const ExpressionTree &obj2) { return false
    ; }
bool operator != (const ExpressionVariable &obj1, const ExpressionTree &obj2) { return true;
     }

bool ExpressionObject::equal(ExpressionObject *obj1, ExpressionObject *obj2){
    ExpressionTree *obj1_tree = dynamic_cast<ExpressionTree*> (obj1);
    ExpressionTree *obj2_tree = dynamic_cast<ExpressionTree*> (obj2);
    ExpressionVariable *obj1_var = obj1_tree == NULL ? dynamic_cast<ExpressionVariable*> (
        obj1) : NULL;
    ExpressionVariable *obj2_var = obj2_tree == NULL ? dynamic_cast<ExpressionVariable*> (
        obj2) : NULL;
    if (obj1_tree && obj2_tree) return (*obj1_tree) == (*obj2_tree);
    if (obj1_var && obj2_var) return (*obj1_var) == (*obj2_var);
    return false;
}

int ExpressionObject::compare(ExpressionObject *obj1, ExpressionObject *obj2){
    ExpressionTree *obj1_tree = dynamic_cast<ExpressionTree*> (obj1);
    ExpressionTree *obj2_tree = dynamic_cast<ExpressionTree*> (obj2);
    ExpressionVariable *obj1_var = obj1_tree == NULL ? dynamic_cast<ExpressionVariable*> (
        obj1) : NULL;
    ExpressionVariable *obj2_var = obj2_tree == NULL ? dynamic_cast<ExpressionVariable*> (
        obj2) : NULL;

    // differing types
    if (obj1_tree && obj2_var) return 1;
    if (obj2_tree && obj1_var) return -1;

    // identical types (tree)
    if (obj1_tree && obj2_tree){
        if ((*obj1_tree) == (*obj2_tree)) return 0;
        int tree1_size = obj1_tree->size();
        int tree2_size = obj2_tree->size();
        if (tree1_size != tree2_size) return tree1_size > tree2_size ? 1 : -1;
        return 0; // <-- why to never, ever use this compare function for equality-checking
            purposes
    }

    // identical types (var)
    if (obj1_var && obj2_var){
        if ((*obj1_var) == (*obj2_var)) return 0;
        if (obj1_var->isnumeric && !obj2_var->isnumeric) return -1;
        if (obj2_var->isnumeric && !obj1_var->isnumeric) return 1;
        if (obj1_var->isnumeric && obj2_var->isnumeric) return obj1_var->numericvalue >
            obj2_var->numericvalue ? 1 : -1;
        return obj1_var->var.compare(obj2_var->var);
    }

    return 0;
}

bool ExpressionObject::sortcompare(ExpressionObject *obj1, ExpressionObject *obj2){
    // true if first goes before second
    int cmp = ExpressionObject::compare(obj1, obj2);
    return cmp < 0;
}
```

```
bool ExpressionObject::rsortcompare(ExpressionObject *obj1, ExpressionObject *obj2){
    return !ExpressionObject::sortcompare(obj1, obj2);
}

#endif
```

5.3 expr_parse.h

```
/*
 *  expr_parse.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/3/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

ExpressionObject* ExpressionObject::parse(string expr, bool optimizeResult){
    int i;
    int start = 0;
    int n = expr.length();
    list<ExpressionObject*> parts;
    list<char> rawbuffer;
    list<int> operators;

    // ignore parentheses at the beginning and end
    // of the expression string whenever possible
    int j = 0;
    stack<char> parens;
    stack<bool> parens_removable;
    bool removable = true;
    int removable_count = 0;
    while (j < n){
        char c = expr[j];
        if (c == '(' || c == '['){
            parens.push(c);
            parens_removable.push(removable);
        }else if (c == ')' || c == ']'){
            char open = parens.top();
            if ((c == ')' && open == '(') || (c == ']' && open == '[')){
                bool open_removable = parens_removable.top();
                if (open_removable){
                    ++removable_count;
                }else{
                    removable_count = 0;
                }
                parens.pop();
                parens_removable.pop();
            }else{
                return NULL;
            }
        }else{
            removable_count = 0;
            removable = false;
        }
        ++j;
    }

    start = removable_count;
    n = expr.length() - removable_count;

    // determine the operator that will serve as the root of this tree.
    // add / subtract take precedence over multiply / divide in this step.
    i = start;
    char op_split = 0;
    while (i < n){
        char c = expr[i];
```

```
        switch (c){
            case '+':
            case '-':
            case '*':
            case '/':
            case '^':
                if (op_split == 0){
                    op_split = c;
                    if (c == '+' || c == '-') break;
                }else if(c == '+' || c == '-'){
                    op_split = c;
                    break;
                }else{
                    if (op_split == '^' && (c == '*' || c == '/')){
                        op_split = c;
                    }
                }
                break;
            case '(':
            case '[':
                i = advance_matching_paren(expr, i, n);
                if (i == -1) return ExpressionObject::invalid();
                break;
        }
        ++i;
    }

    if (op_split == 0){
        return new ExpressionVariable(expr.substr(start, n));
    }else{
        ExpressionTree *tree = new ExpressionTree();
        tree->root = new ExpressionOperator(op_split);
        // split by the root operator
        i = start;
        int raw_begin = i;
        while (i < n){
            if (expr[i] == '(' || expr[i] == '['){
                i = advance_matching_paren(expr, i, n);
            }else if(expr[i] == op_split){
                if (i - raw_begin > 0){
                    tree->children.push_back(ExpressionObject::parse(expr.substr(raw_begin,
                        i-raw_begin), optimizeResult));
                }
                raw_begin = i + 1;
                if (op_split == '^'){ i = n; break; }
            }
            ++i;
        }
        if (i - raw_begin > 0){
            tree->children.push_back(ExpressionObject::parse(expr.substr(raw_begin, i-
                raw_begin), optimizeResult));
        }
        return (optimizeResult) ? tree->optimize() : tree;
    }
}

ExpressionObject* ExpressionObject::parse(string expr){
    return parse(expr, true);
}
```

5.4 expr_helpers.h

```
#ifndef _EXPR_HELPERS_H
#define _EXPR_HELPERS_H
#include <string>
using std::string;
```

```
string str_repeat(string str, int count){
    string output = "";
    for (int i = 0; i < count; i++){
        output += str;
    }
    return output;
}

// returns the index of the matching parentheis to the parentheis
// located at expr[start]. if the parenthesis are not properly nested,
// -1 is returned.
int advance_matching_paren(string expr, int start, int stopat){
    stack<char> parens;
    parens.push(expr[start]);
    int j = start + 1;
    int n = !stopat ? expr.length() : stopat;
    while (j < n){
        switch (expr[j]){
            case '(':
            case '[':
                parens.push(expr[j]);
                break;
            case ')':
                if (parens.top() != '(') return -1;
                parens.pop();
                break;
            case ']':
                if (parens.top() != '[') return -1;
                parens.pop();
                break;
            default:
                ++j;
                continue;
        }
        if (parens.empty()){
            return j;
        }
        ++j;
    }
    if (!parens.empty()) return -1;
        return j;
    }

#endif
```

5.5 expr_latex.h

```
/*
 *  expr_latex.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/3/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

string ExpressionVariable::toLaTeX(){
    if (isnumeric){
        int epos = var.find('e');
        if (epos == string::npos){
            return var;
        }else{
            // fix the formatting of the power up a bit
            stringstream power_str(var.substr(epos + 1));
            double power = 1;
            power_str >> power;
```

```
            ostringstream power_strstream;
            power_strstream.precision(3);
            power_strstream << power;

            string sci_notation = var.substr(0, epos);
            sci_notation.append("\\times 10^{");
            sci_notation.append(power_strstream.str());
            sci_notation.append("}");
            return sci_notation;
        }
    }
    return var;
}

string ExpressionTree::toLaTeX(){
    string str;
    string op(1, root->operation);
    for (int i = 0; i < children.size(); ++i){
        if (i > 0 && root->operation != '*') str.append(op);
        ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (children[i]);
        if (child_tree != NULL){
            bool wrap = false;
            bool wrap_bracket = false;
            if (child_tree->root->operation == '+' || child_tree->root->operation == '-'){
                wrap = root->operation == '*' || root->operation == '/' || root->operation
                    == '^';
                wrap_bracket = root->operation == '^';
            }
            if (root->operation == '^'){
                wrap = true;
                wrap_bracket = true;
            }
            if (wrap) str.append(wrap_bracket ? "{\\left(" : "(");
            str.append(children[i]->toLaTeX());
            if (wrap) str.append(wrap_bracket ? "\\right)}" : ")");
        }else{
            bool implicit_bracket = root->operation == '^' || root->operation == '/';
            if (implicit_bracket) str.append("{");
            str.append(children[i]->toLaTeX());
            if (implicit_bracket) str.append("}");
        }
    }

    bool implicit_bracket = root->operation == '^';
    return (implicit_bracket ? "{" : "") + str + (implicit_bracket ? "}" : "");
}


string ExpressionVariable::toLaTeXTree(){
    return ExpressionVariable::toLaTeXTree(0);
}
string ExpressionVariable::toLaTeXTree(int level){
    if (level == 0){
        return "\\Tree [." + toLaTeX() + " ]";
    }else{
        return toLaTeX();
    }
}


string ExpressionTree::toLaTeXTree(){
    return ExpressionTree::toLaTeXTree(0);
}
string ExpressionTree::toLaTeXTree(int level){
    string str;
    string op = root->toLaTeX();
    if (level == 0){
        str.append("\\Tree ");
```

```
    }
    str.append("[." + op + "␣");
    for (int i = 0; i < children.size(); ++i){
        str.append(children[i]->toLaTeXTree(level + 1));
        str.append("␣");
    }
    str.append("]");
    return str;
}
```

5.6 expr_operator.h

```
/*
 *  expr_operator.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

#ifdef _EXPR_H

/**
 * EXPRESSION OPERATOR
 * located at the root of an expression tree, expression
 * operators represent the operator that joins the children
 */
class ExpressionOperator {
public:
    static const char MULT = '*';
    static const char DIV  = '/';
    static const char ADD  = '+';
    static const char SUB  = '-';
    char operation;
    ExpressionOperator(char op) : operation(op) {};
    double getk(){
        if (operation == '*' || operation == '/') return 1;
        return 0;
    }
    double evaluate(double a, double b){
        switch (operation){
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/': return a / b;
            case '^': return pow(a, b);
        }
        return 0;
    }

    //equality operators
    bool operator==(const ExpressionOperator &obj) const {
        return operation == obj.operation;
    }
    bool operator!=(const ExpressionOperator &obj) const {
        return !(*this == obj);
    }
    string toLaTeX(){
        switch (operation){
            case '+': return "$\\varoplus$";
            case '-': return "$\\varominus$";
            case '*': return "$\\varoast$";
            case '/': return "$\\varoslash$";
            case '^': return "$\\varowedge$";
        }
        return "";
```

```
    }
    void print(ostream &stream){
        stream << "(" << operation << ")" << endl;
    }
    void print(ostream &stream, int level){
        cout << str_repeat(TAB, level);
        print(stream);
    }
};

#endif
```

5.7 expr_variable.h

```
/*
 *  expr_variable.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

#ifdef _EXPR_H

class ExpressionVariable : public ExpressionObject {
public:
    string var;
    bool isnumeric;
    double numericvalue;

    // constructors
    ExpressionVariable(string variable) : var(variable) {
        stringstream strstream(variable);
        isnumeric = !((strstream >> numericvalue).fail());
    }
    ExpressionVariable(double value) : isnumeric(true), numericvalue(value) {
        ostringstream strstream;
        strstream << value;
        var = strstream.str();
    }

    ExpressionVariable* clone(){
        ExpressionVariable *obj = new ExpressionVariable(var);
        return obj;
    }

    //equality operators
    bool operator==(const ExpressionVariable &obj) const {
        if (isnumeric && !obj.isnumeric) return false;
        if (!isnumeric && obj.isnumeric) return false;
        if (isnumeric) return numericvalue == obj.numericvalue;
        return var == obj.var;
    }
    bool operator!=(const ExpressionVariable &obj) const {
        return !(*this == obj);
    }

    void print(ostream &stream){
        stream << var << endl;
    }
    void print(ostream &stream, int level){
        stream << str_repeat(TAB, level);
        print(stream);
    }
    string toLaTeX();
    string toLaTeXTree();
```

```
    string toLaTeXTree(int level);

    bool isNumeric(){
        return isnumeric;
    }
    ExpressionVariable* differentiate(string variable){
        if (isnumeric) return new ExpressionVariable(0);
        return new ExpressionVariable(var == variable ? 1 : 0);
    }
    ExpressionVariable* evaluate(double value, string variable) {
        if (isnumeric) this->clone();
        if (variable == var) return new ExpressionVariable(value);
        return this->clone();
    }
    ExpressionObject* evaluate(ExpressionObject *value, string variable) {
        if (isnumeric) return this->clone();
        if (variable == var) return value->clone();
        return this->clone();
    }
    ExpressionObject* optimize(){ return this; };
};

#endif
```

5.8 expr_tree.h

```
/*
 *  expr_tree.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

#ifdef _EXPR_H

/**
 * EXPRESSION TREE
 * contains a list of expression objects joined
 * by an operator.
 */
class ExpressionTree : public ExpressionObject {
public:
    vector<ExpressionObject*> children;

    // the operator that joins all the tree's children together
    ExpressionOperator *root;

    // default constructor
    ExpressionTree() : root(NULL) {};

    // clone
    ExpressionTree* clone() {
        ExpressionTree *obj = new ExpressionTree();
        obj->root = new ExpressionOperator(root->operation);
        for (int i = 0; i < children.size(); ++i){
            obj->children.push_back(children[i]->clone());
        }
        return obj;
    };

    // equality operator
    bool operator==(const ExpressionTree &obj) const {
        if (root == NULL && obj.root != NULL) return false;
        if (root != NULL && obj.root == NULL) return false;
        if (root->operation != obj.root->operation) return false;
```

```
    if (children.size() != obj.children.size()) return false;
    for (int i = 0; i < children.size(); ++i){
        if (!ExpressionObject::equal(children[i], obj.children[i])) return false;
    }
    return true;
}
bool operator!=(const ExpressionTree &obj) const {
    return !(*this == obj);
}

// calculates the size of the tree
// (used for comparing trees for sorting purposes)
int size(){
    int n = 1;
    for (int i = 0; i < children.size(); ++i){
        ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (children[i]);
        if (child_tree != NULL) n += child_tree->size();
        else ++n;
    }
    return n;
}

// reduces the complexity of the tree wherever possible.
// e.g. 2x + 5x -> 7x

ExpressionObject* optimize(){
    for (int i = 0; i < children.size(); ++i){
        children[i] = children[i]->optimize();
    }
    const int MAX_ITERATIONS = 4;
    int iteration = 0;
    ExpressionObject *optimized = this;
    ExpressionObject *before;
    do {

        ExpressionTree *optimized_tree;
        before = optimized;

        optimized_tree = dynamic_cast<ExpressionTree*> (optimized);

        // optimize_redundant_operators
        if (optimized_tree){
            optimized = optimized_tree->optimize_redundant_operators();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // distribute constants (2(x+1) = 2x+2)
        if (optimized_tree){
            optimized = optimized_tree->optimize_distribute_constants();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_consolidate_terms (8a + a = 9a)
        if (optimized_tree){
            optimized = optimized_tree->optimize_consolidate_terms();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_consolidate_scalars (8 * 8 * x = 64 * x)
        if (optimized_tree){
            optimized = optimized_tree->optimize_consolidate_scalars();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_const_prod_pow ( (2*x)^2 = 4(x)^2 )
        if (optimized_tree){
            optimized = optimized_tree->optimize_const_prod_pow();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_consolidate_powers ((x^2)^2 = x^4)
        if (optimized_tree){
            optimized = optimized_tree->optimize_consolidate_powers();
```

```
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_consolidate_factors (x^2 * x^2 = x^4)
        if (optimized_tree){
            optimized = optimized_tree->optimize_consolidate_factors();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }
        // optimize_redundant_operators
        if (optimized_tree){
            optimized = optimized_tree->optimize_redundant_operators();
            optimized_tree = dynamic_cast<ExpressionTree*> (optimized);
        }

        // reoptimize the children since they might have been changed
        if (optimized_tree){
            for (int i = 0; i < optimized_tree->children.size(); ++i){
                optimized_tree->children[i] = optimized_tree->children[i]->optimize();
            }
        }

        // neaten up the children
        if (optimized_tree) optimized_tree->sort_children();

        ++iteration;
    } while (iteration < MAX_ITERATIONS && !ExpressionObject::equal(optimized, before));

    return optimized;
}

ExpressionObject* optimize_redundant_operators();
ExpressionObject* optimize_consolidate_terms();
ExpressionObject* optimize_consolidate_scalars();
ExpressionObject* optimize_consolidate_factors();
ExpressionObject* optimize_consolidate_powers();
ExpressionObject* optimize_distribute_constants();
ExpressionObject* optimize_const_prod_pow();
void sort_children();

// differentiates the equation represented by the expression tree
// and returns a tree representing the result.
ExpressionObject* differentiate(string variable);

// evaluates the expression tree for a given number
ExpressionObject* evaluate(double value, string variable);
ExpressionObject* evaluate(ExpressionObject *value, string variable);

void print(ostream &stream){
    print(stream, 0);
}
void print(ostream &stream, int level){
    if (root) root->print(stream, level);
    for (int i = 0; i < children.size(); ++i){
        children[i]->print(stream, level + 1);
    }
}
string toLaTeX();
string toLaTeXTree();
string toLaTeXTree(int level);
};

#include "expr_tree_evaluate.h"
#include "expr_tree_differentiate.h"

#include "expr_tree_optimize_redundant_operators.h"
#include "expr_tree_optimize_consolidate_terms.h"
#include "expr_tree_optimize_consolidate_factors.h"
#include "expr_tree_optimize_consolidate_scalars.h"
#include "expr_tree_optimize_consolidate_powers.h"
```

```
#include "expr_tree_optimize_distribute_constants.h"
#include "expr_tree_optimize_const_prod_pow.h"

#include "expr_tree_sort_children.h"

#endif
```

5.9 expr_tree_evaluate.h

```
/*
 *  expr_tree_evaluate.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/3/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */


ExpressionObject* ExpressionTree::evaluate(ExpressionObject *value, string variable) {

    // evaluation if a power operation
    if (root->operation == '^'){
        ExpressionObject *base = children[0]->evaluate(value, variable);
        ExpressionVariable *base_var = dynamic_cast<ExpressionVariable*> (base);
        ExpressionObject *power = children[1]->evaluate(value, variable);
        ExpressionVariable *power_var = dynamic_cast<ExpressionVariable*> (power);
        if (power_var && power_var->isnumeric && power_var->numericvalue == 0){
            return new ExpressionVariable(1);
        }
        if (base_var && base_var->isnumeric && power_var && power_var->isnumeric){
            return new ExpressionVariable(pow(base_var->numericvalue, power_var->
                numericvalue));
        }
        ExpressionTree *ans = new ExpressionTree();
        ans->root = new ExpressionOperator('^');
        ans->children.push_back(base);
        ans->children.push_back(power);
        return ans->optimize();
    }

    double k = root->getk();
    vector<ExpressionObject*> nonscalars;
    int start = (root->operation == '-' || root->operation == '/') ? 1 : 0;

    // determine the object being divided / subtracted-from, if appropriate
    ExpressionObject *first;
    if (start == 1){
        first = children[0]->evaluate(value, variable);
        ExpressionVariable *first_var = dynamic_cast<ExpressionVariable*> (first);
        if (first_var && first_var->isnumeric){
            k = first_var->numericvalue;
            first = NULL;
        }
    }

    // evaluate the children
    for (int i = start; i < children.size(); ++i){
        ExpressionObject *child_result = children[i]->evaluate(value, variable);
        ExpressionVariable *child_result_var = dynamic_cast<ExpressionVariable*> (
            child_result);
        if (child_result_var != NULL && child_result_var->isnumeric){
            k = root->evaluate(k, child_result_var->numericvalue);
        }else{
            nonscalars.push_back(child_result);
        }
    }
```

```
    // assemble the result
    if (start == 1){
        if (k == root->getk() && nonscalars.size() == 0){
            return first == NULL ? new ExpressionVariable(k) : first;
        }
        ExpressionTree *ans = new ExpressionTree();
        ans->root = new ExpressionOperator(root->operation);
        ans->children.push_back(first == NULL ? new ExpressionVariable(k) : first);
        for (int i = 0; i < nonscalars.size(); ++i){
            ans->children.push_back(nonscalars[i]);
        }
        return ans->optimize();
    }else{
        if (k == root->getk()){
            if (nonscalars.size() == 0) return new ExpressionVariable(k);
            if (nonscalars.size() == 1) return nonscalars[0];
        }
        ExpressionTree *ans = new ExpressionTree();
        ans->root = new ExpressionOperator(root->operation);
        if (k != root->getk()){
            ans->children.push_back(new ExpressionVariable(k));
        }
        for (int i = 0; i < nonscalars.size(); ++i){
            ans->children.push_back(nonscalars[i]);
        }
        return ans->optimize();
    }
}


ExpressionObject* ExpressionTree::evaluate(double value, string variable) {
    return evaluate(new ExpressionVariable(value), variable);
}
```

5.10 expr_tree_differentiate.h

```
/*
 *  expr_tree_differentiate.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/3/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

ExpressionObject* ExpressionTree::differentiate(string variable){
    ExpressionTree *differentiated = new ExpressionTree();
    switch (root->operation){
        case '+':
        case '-':
            differentiated->root = new ExpressionOperator(root->operation);
            for (int i = 0; i < children.size(); ++i){
                ExpressionObject *diff_child = children[i]->differentiate(variable);
                if (!diff_child) return ExpressionObject::invalid();
                ExpressionVariable *diff_child_var = dynamic_cast<ExpressionVariable*> (
                    diff_child);
                if (diff_child_var != NULL && diff_child_var->isnumeric && diff_child_var->
                    numericvalue == 0) continue;
                differentiated->children.push_back(diff_child);
            }
            break;
        case '*':
            differentiated->root = new ExpressionOperator('+');
            for (int i = 0; i < children.size(); ++i){
                ExpressionObject *diff_child = children[i]->differentiate(variable);
                if (!diff_child) return ExpressionObject::invalid();
```

```
                ExpressionVariable *diff_child_var = dynamic_cast<ExpressionVariable*> (
                    diff_child);
                if (diff_child_var != NULL && diff_child_var->isnumeric && diff_child_var->
                    numericvalue == 0) continue;
                bool diff_child_isone = diff_child_var != NULL && diff_child_var->isnumeric
                    && diff_child_var->numericvalue == 1;

                if (diff_child_isone && children.size() == 2){
                    differentiated->children.push_back(children[(i+1)%2]);
                }else{
                    ExpressionTree *prod = new ExpressionTree();
                    prod->root = new ExpressionOperator('*');
                    prod->children.push_back(diff_child);
                    for (int j = 0; j < children.size(); ++j){
                        if (j == i) continue;
                        prod->children.push_back(children[j]);
                    }
                    differentiated->children.push_back(prod);
                }
            }
            break;
        case '/':
            return ExpressionObject::invalid();
            break;
        case '^':
            // at the moment, this will only differentiate constant powers
            // e.g. (x+2)^3 and NOT (x+2)^x
            // and exponentials
            if (children.size() != 2) return ExpressionObject::invalid();
            differentiated->root = new ExpressionOperator('*');
            ExpressionObject *base = children[0];
            ExpressionVariable *base_var = dynamic_cast<ExpressionVariable*> (base);
            ExpressionObject *power = children[1];
            ExpressionVariable *power_var = dynamic_cast<ExpressionVariable*> (power);
            if (base_var && !base_var->isnumeric && base_var->var == "e"){
                ExpressionObject *power_diff = power->differentiate(variable);
                if (!power_diff) return ExpressionObject::invalid();
                differentiated->children.push_back(power_diff);
                differentiated->children.push_back(this);
            }else if (power_var && power_var->isnumeric){
                double new_power = power_var->numericvalue - 1;
                ExpressionObject *base_diff = base->differentiate(variable);
                if (!base_diff) return ExpressionObject::invalid();
                if (power_var->numericvalue != 1){
                    differentiated->children.push_back(new ExpressionVariable(power_var->
                        numericvalue));
                }
                differentiated->children.push_back(base_diff);
                if (new_power != 0 && new_power != 1){
                    ExpressionTree *pow_tree = new ExpressionTree();
                    pow_tree->root = new ExpressionOperator('^');
                    pow_tree->children.push_back(base);
                    pow_tree->children.push_back(new ExpressionVariable(new_power));
                    differentiated->children.push_back(pow_tree);
                }else if(new_power == 1){
                    differentiated->children.push_back(base);
                }
            }else{
                return ExpressionObject::invalid();
            }
            break;
    }

    return differentiated->optimize();
}
```

5.11 expr_tree_optimize_consolidate_terms.h

```
/*
 *  expr_tree_optimize_factor.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

ExpressionObject* ExpressionTree::optimize_consolidate_terms(){
    if (root->operation == '+'){
        ExpressionTree *child_tree;
        vector<ExpressionObject*> final_terms;
        final_terms.reserve(children.size());

        set<int> indices;
        for (int i = 0; i < children.size(); ++i){
            indices.insert(i);
        }

        while (!indices.empty()){
            set<int>::iterator i_iter = indices.begin();
            int i = *i_iter;

            double k = 1;

            // accumulate the variables / coefficients of this term
            vector<ExpressionObject*> vars;
            child_tree = dynamic_cast<ExpressionTree*>(children[i]);
            if (child_tree != NULL){
                if (child_tree->root->operation == '*'){
                    int j = 0;
                    ExpressionVariable *expr_var = dynamic_cast<ExpressionVariable*>(
                        child_tree->children[0]);
                    if (expr_var != NULL && expr_var->isnumeric){
                        k = expr_var->numericvalue;
                        j = 1;
                    }
                    while (j < child_tree->children.size()){
                        vars.push_back(child_tree->children[j]);
                        ++j;
                    }
                }else{
                    vars.push_back(child_tree);
                }
            }else{
                vars.push_back(children[i]);
            }

            // check the other terms for identical variables
            // (variables existing in the vars vector)
            for (int j = 0; j < children.size(); ++j){
                if (j == i) continue;

                // accumulate the variables / coefficients of this sibling term
                double c = 1;
                vector<ExpressionObject*> vars_sibling;
                child_tree = dynamic_cast<ExpressionTree*>(children[j]);
                if (child_tree != NULL){
                    if (child_tree->root->operation == '*'){
                        int j = 0;
                        ExpressionVariable *expr_var = dynamic_cast<ExpressionVariable*>(
                            child_tree->children[0]);
                        if (expr_var != NULL && expr_var->isnumeric){
                            c = expr_var->numericvalue;
                            j = 1;
```

```
                    }
                    while (j < child_tree->children.size()){
                        vars_sibling.push_back(child_tree->children[j]);
                        ++j;
                    }
                }else{
                    vars_sibling.push_back(child_tree);
                }
            }else{
                vars_sibling.push_back(children[j]);
            }

            // check to see if the variables are identical to those
            // of the variable we're looking at currently
            if (vars_sibling.size() == vars.size()){
                for (int m = 0; m < vars.size(); ++m){
                    bool sibling_found = false;
                    for (int n = 0; n < vars_sibling.size(); ++n){
                        if (ExpressionObject::equal(vars_sibling[n], vars[m])){
                            sibling_found = true;
                            vars_sibling.erase(vars_sibling.begin() + n);
                            break;
                        }
                    }
                    if (!sibling_found) break;
                }
                if (vars_sibling.size() == 0){
                    indices.erase(indices.find(j));
                    k += c;
                }
            }
        }

        // add the term to the final terms list
        if (k == 0){
            // do nothing
        }else if (vars.size() == 1 && k == 1){
            final_terms.push_back(vars[0]);
        }else{
            ExpressionTree *new_tree = new ExpressionTree();
            new_tree->root = new ExpressionOperator('*');
            if (k != 1){
                new_tree->children.push_back(new ExpressionVariable(k));
            }
            for (int m = 0; m < vars.size(); ++m){
                new_tree->children.push_back(vars[m]);
            }
            final_terms.push_back(new_tree);
        }
        indices.erase(indices.find(i));
    }

    // reassemble the children
    children.clear();
    if (final_terms.size() == 1 && (child_tree = dynamic_cast<ExpressionTree*>(
        final_terms[0]))){
        root->operation = child_tree->root->operation;
        children = child_tree->children;
    }else{
        for (int i = 0; i < final_terms.size(); ++i){
            children.push_back(final_terms[i]);
        }
    }
}
return this;
}
```

5.12 expr_tree_optimize_consolidate_factors.h

```
/*
 *  expr_tree_optimize_consolidate_factors.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/4/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */


ExpressionObject* ExpressionTree::optimize_consolidate_factors(){
    if (root->operation == '*'){
        // are there any zero-coefficients?
        for (int i = 0; i < children.size(); ++i){
            ExpressionVariable *child_var = dynamic_cast<ExpressionVariable*> (children[i]);
            if (child_var != NULL && child_var->isnumeric && child_var->numericvalue == 0){
                return new ExpressionVariable(0);
            }
        }

        // join terms with identical factors
        vector<ExpressionObject*> final_terms;
        final_terms.reserve(children.size());

        set<int> indices;
        for (int i = 0; i < children.size(); ++i){
            indices.insert(i);
        }

        while (!indices.empty()){
            set<int>::iterator i_iter = indices.begin();
            int i = *i_iter;

            double pow_scalar;
            vector<ExpressionObject*> pow_vars;

            // determine the base of the current child (along with the power it's raised to)
            ExpressionObject *base;
            ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (children[i]);
            if (child_tree != NULL && child_tree->root->operation == '^'){
                base = child_tree->children[0];
                pow_scalar = 0;
                for (int p = 1; p < child_tree->children.size(); ++p){
                    ExpressionVariable *power_var = dynamic_cast<ExpressionVariable*> (
                        child_tree->children[p]);
                    if (power_var != NULL && power_var->isnumeric){
                        pow_scalar += power_var->numericvalue;
                    }else{
                        pow_vars.push_back(child_tree->children[p]);
                    }
                }
            }else{
                base = children[i];
                pow_scalar = 1;
            }

            // look for identical siblings
            for (int j = 0; j < children.size(); ++j){
                if (j == i) continue;
                set<int>::iterator j_iter = indices.find(j);
                if (j_iter == indices.end()) continue;

                // is the term identical?
                if (ExpressionObject::equal(children[j], base)){
                    ++pow_scalar;
                    indices.erase(indices.find(j));
```

```
                continue;
            }
            // is the term an identical term raised to a power?
            ExpressionTree *tree = dynamic_cast<ExpressionTree*> (children[j]);
            if (tree != NULL && tree->root->operation == '^'){
                ExpressionObject *base_sibling = tree->children[0];
                if (ExpressionObject::equal(base_sibling, base)){
                    indices.erase(j_iter);
                    for (int p = 1; p < tree->children.size(); ++p){
                        ExpressionVariable *power_var = dynamic_cast<ExpressionVariable
                            *> (tree->children[p]);
                        if (power_var != NULL && power_var->isnumeric){
                            pow_scalar += power_var->numericvalue;
                        }else{
                            pow_vars.push_back(tree->children[p]);
                        }
                    }
                }
            }
        }

        if (!pow_vars.empty()){
            ExpressionObject *pow_sum;
            if (pow_scalar != 0 || pow_vars.size() > 1){
                ExpressionTree *pow_sum_tree = new ExpressionTree();
                pow_sum_tree->root = new ExpressionOperator('+');
                if (pow_scalar != 0){
                    pow_sum_tree->children.push_back(new ExpressionVariable(pow_scalar))
                        ;
                }
                for (int j = 0; j < pow_vars.size(); ++j){
                    pow_sum_tree->children.push_back(pow_vars[j]);
                }
                pow_sum = pow_sum_tree;
            }else{
                if (pow_scalar != 0) pow_sum = new ExpressionVariable(pow_scalar);
                else pow_sum = pow_vars[0];
            }
            ExpressionTree *pow = new ExpressionTree();
            pow->root = new ExpressionOperator('^');
            pow->children.push_back(base);
            pow->children.push_back(pow_sum);
            final_terms.push_back(pow);
        }else if(pow_scalar != 1){
            ExpressionTree *pow = new ExpressionTree();
            pow->root = new ExpressionOperator('^');
            pow->children.push_back(base);
            pow->children.push_back(new ExpressionVariable(pow_scalar));
            final_terms.push_back(pow);
        }else{
            final_terms.push_back(base);
        }

        indices.erase(indices.find(i));
    }

    children = final_terms;
  }
  return this;
}
```

5.13 expr_tree_optimize_consolidate_scalars.h

```
/*
 *  expr_tree_optimize_consolidate_scalars.h
 *  CS-Final-Project
 *
```

```
 *  Created by Brian Reavis on 12/3/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */


ExpressionObject* ExpressionTree::optimize_consolidate_scalars(){
    if (root->operation == '^') return this;
    double k = root->getk();
    vector<ExpressionObject*> children_new;
    children_new.push_back(NULL);
    for (int i = 0; i < children.size(); ++i){
        ExpressionVariable *child_var = dynamic_cast<ExpressionVariable*> (children[i]);
        if (child_var != NULL && child_var->isnumeric){
            k = root->evaluate(k, child_var->numericvalue);
        }else{
            children_new.push_back(children[i]);
        }
    }
    if (k != root->getk()){
        children_new[0] = new ExpressionVariable(k);
    }else{
        children_new.erase(children_new.begin());
    }
    children = children_new;
    return this;
}
```

5.14 expr_tree_optimize_consolidate_powers.h

```
/*
 *  expr_tree_optimize_consolidate_powers.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/5/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */


ExpressionObject* ExpressionTree::optimize_consolidate_powers(){
    if (root->operation == '^' && children.size() == 2){
        double k = 1;
        vector<ExpressionObject*> powers;

        ExpressionObject *power, *base;
        ExpressionVariable *power_var;
        ExpressionTree *base_tree = this;
        do {
            base = base_tree->children[0];
            power = base_tree->children[1];
            base_tree = dynamic_cast<ExpressionTree*> (base);
            power_var = dynamic_cast<ExpressionVariable*> (power);
            if (power_var && power_var->isnumeric){
                k *= power_var->numericvalue;
            }else{
                ExpressionTree *power_tree = dynamic_cast<ExpressionTree*> (power);
                if (power_tree != NULL && power_tree->root->operation == '*'){
                    for (int i = 0; i < power_tree->children.size(); ++i){
                        powers.push_back(power_tree->children[i]);
                    }
                }else{
                    powers.push_back(power);
                }
            }
        } while(base_tree && base_tree->root->operation == '^' && base_tree->children.size()
            == 2);
```

```
        ExpressionTree *reduced = new ExpressionTree();
        reduced->root = new ExpressionOperator('^');
        reduced->children.push_back(base);
        if (k == 0) return new ExpressionVariable(0);
        if (powers.size() == 0 && k == 1){
            return base;
        }
        if (powers.size() == 1 && k == 1){
            reduced->children.push_back(powers[0]);
            return reduced;
        }

        ExpressionTree *new_power = new ExpressionTree();
        new_power->root = new ExpressionOperator('*');
        if (k != 1) new_power->children.push_back(new ExpressionVariable(k));
        for (int i = 0; i < powers.size(); ++i){
            new_power->children.push_back(powers[i]);
        }
        reduced->children.push_back(new_power);
        return reduced;
    }
    return this;
}
```

5.15 expr_tree_optimize_consolidate_redundant_operators.h

```
/*
 *  expr_tree_optimize_redundant_operators.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

ExpressionObject* ExpressionTree::optimize_redundant_operators(){
    if (children.size() == 1){
        return children[0];
    }

    vector<ExpressionObject*> new_children;
    new_children.reserve(children.size() + 5);

    for (int i = 0; i < children.size(); ++i){
        ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (children[i]);
        if (child_tree != NULL && (child_tree->children.size() == 1 || (child_tree->root->
            operation == root->operation && root->operation != '^'))){
            for (int j = 0; j < child_tree->children.size(); ++j){
                new_children.push_back(child_tree->children[j]);
            }
        }else{
            new_children.push_back(children[i]);
        }
    }

    children = new_children;
    return this;
}
```

5.16 expr_tree_optimize_const_prod_pow.h

```
/*
 *  expr_tree_optimize_const_prod_pow.h
 *  CS-Final-Project
```

```
 *
 *  Created by Brian Reavis on 12/4/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

ExpressionObject* ExpressionTree::optimize_const_prod_pow(){
    if (root->operation == '^' && children.size() == 2){

        ExpressionTree *base_tree = dynamic_cast<ExpressionTree*> (children[0]);
        if (base_tree == NULL || base_tree->root->operation != '*') return this;

        ExpressionVariable *pow_var = dynamic_cast<ExpressionVariable*> (children[1]);
        if (pow_var == NULL || !pow_var->isnumeric) return this;

        double k = 1;
        vector<ExpressionObject*> base_vars;
        for (int i = 0; i < base_tree->children.size(); ++i){
            ExpressionVariable *base_var = dynamic_cast<ExpressionVariable*> (base_tree->
                children[i]);
            if (base_var && base_var->isnumeric){
                k *= pow(base_var->numericvalue, pow_var->numericvalue);
                continue;
            }
            base_vars.push_back(base_tree->children[i]);
        }

        if (base_vars.size() ==  1){
            children[0] = base_vars[0];
        }else{
            base_tree->children = base_vars;
        }

        if (k == 0){
            return new ExpressionVariable(0);
        }else if (k != 1){
            ExpressionTree *new_tree = new ExpressionTree();
            new_tree->root = new ExpressionOperator('*');
            new_tree->children.push_back(new ExpressionVariable(k));
            new_tree->children.push_back(this);
            return new_tree;
        }
    }
    return this;
}
```

5.17 expr_tree_optimize_distribute_constants.h

```
/*
 *  expr_tree_optimize_distribute_constants.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/5/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

// 2(x+1)(x^2+1) -> (2x+2)(x^2+1)
ExpressionObject* ExpressionTree::optimize_distribute_constants(){
    if (root->operation == '*'){
        int k = 1;
        ExpressionTree *add_tree = NULL;

        // determine the scalar being distributed
        // along with the location of the (+) tree
        vector<ExpressionObject*> factors;
        for (int i = 0; i < children.size(); ++i){
```

```
            ExpressionVariable *child_var = dynamic_cast<ExpressionVariable*> (children[i]);
            if (child_var != NULL && child_var->isnumeric){
                k *= child_var->numericvalue;
                continue;
            }
            if (add_tree == NULL){
                ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (children[i]);
                if (child_tree && (child_tree->root->operation == '+' || child_tree->root->
                    operation == '-')){
                    add_tree = child_tree;
                }
            }
            factors.push_back(children[i]);
        }

        // distribute k
        if (k == 0) return new ExpressionVariable(0);
        if (k != 1 && add_tree != NULL){
            for (int i = 0; i < add_tree->children.size(); ++i){
                ExpressionObject *child = add_tree->children[i];
                ExpressionTree *child_tree = dynamic_cast<ExpressionTree*> (child);
                if (child_tree != NULL && child_tree->root->operation == '*'){
                    child_tree->children.push_back(new ExpressionVariable(k));
                }else{
                    ExpressionTree *prod_tree = new ExpressionTree();
                    prod_tree->root = new ExpressionOperator('*');
                    prod_tree->children.push_back(new ExpressionVariable(k));
                    prod_tree->children.push_back(child);
                    add_tree->children[i] = prod_tree;
                }
            }
            children = factors;
        }

    }
    return this;
}
```

5.18 expr_tree_sort_children.h

```
/*
 *  expr_tree_sort_children.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
 *
 */

void ExpressionTree::sort_children(){
    if (root->operation == '+'){
        sort(children.begin(), children.end(), ExpressionObject::rsortcompare);
    }else if(root->operation == '*'){
        sort(children.begin(), children.end(), ExpressionObject::sortcompare);
    }
}
```

5.19 expr_function.h

```
/*
 *  expr_function.h
 *  CS-Final-Project
 *
 *  Created by Brian Reavis on 12/2/10.
 *  Copyright 2010 __MyCompanyName__. All rights reserved.
```

```
 *
 */

/**
 * EXPRESSION FUNCTION
 * represents fundamental mathematical functions
 * (ln, log, sin, cos, tan, etc)
 */
class ExpressionFunction : public ExpressionObject {
public:
    // function classes
    static const char EXP = 1;
    static const char LOG = 2;
    static const char SIN = 3;
    static const char COS = 4;
    static const char TAN = 5;
};
```