

CSE 335 Project 1, Spring 2005

This project is due at 11:59pm on Wednesday, February 9, 2005.

Objectives:

1. Develop a cohesive library of classes;
2. Apply the composite pattern;
3. Implement complex polymorphic operations;
4. Implement a reference-counting protocol for managing memory usage.

Description: In this project, you are to develop a library of classes for representing, evaluating, simplifying, and differentiating mathematical expressions. Expressions in this language can be:

- a floating-point literal, e.g., 5.0;
- a variable, e.g., x ;
- a sum of two expressions, e.g., $x + 5$;
- a difference of two expressions, e.g., $x - 5$;
- a product of two expressions, e.g., $x * (x - y + 5)$;
- the negation of an expression, e.g., $-(y^2 + x)$;
- the sine of an expression, e.g., $\sin(x + 5)$;
- the cosine of an expression, e.g., $\cos(x + 5)$;
- the exponentiation of an expression, e.g., e^{x+2} ;
- the natural logarithm of an expression, e.g., $\ln(x + 3)$;
- an expression raised to some integer power, e.g., $(x + 2y)^2$;

Each of the operations in this language must be implemented using a separate class, and organized into a hierarchy according to the principles we have been discussing.

This class hierarchy must support at least the five polymorphic operations listed below. Any operations that result in new expressions should construct and return new `Expr` trees.

- **Printing:** Produces a human-readable ASCII representation of the expression to an output stream, which must be supplied as an argument, i.e.:

```
void print( ostream& ) const;
```

You will probably need to add parentheses around sub-expressions in order to ensure that the printed output correctly represents the expression being printed. For this project, it is not necessary to minimize the number of added parentheses.

- Partial differentiation: Produces an expression that is the partial derivative of the target expression with respect to some variable. The name of this variable must be supplied when the operation is invoked, i.e.:

```
Expr* differentiate( const string& ) const;
```

For example, if `e` points to an `Expr` tree representation of $2xy + x^3$, then executing:

```
e->differentiate("x");
```

should produce an `Expr` tree representation of $2y + 3x^2$; whereas executing:

```
e->differentiate("y");
```

should produce an `Expr` tree representation of $2x + 0$.

- Evaluation: Attempts to evaluate the target expression to produce a numeric result. If the expression contains no variables, then the operation returns `true` and copies the result into its argument. Otherwise, it returns `false` and does not modify its argument. The operation signature is:

```
bool evaluate(float&) const;
```

For example, the expression $\cos(3.0 + 35.2) + 0.0$ may be evaluated to yield (approximately) 0.786.

- Closure: Produces an expression that is structurally equivalent to the expression being closed, except that variable references are replaced by the literals to which they are bound by a structure called an *environment*, which maintains a mapping of variable names to values. The environment structure must be supplied as an argument to the `close` operation, i.e.:

```
Expr* close( const Environment& ) const;
```

For example, suppose the variable `e` points to an `Expr` tree representation of $(w + (x - 5))$ and that the environment `env` maps variables to values as follows:

```
{  x  →  2.5,
   y  → -30.123 }
```

Then the execution of:

```
e->close(env);
```

should produce an `Expr` tree representation of $(w + (2.5 - 5))$. Note that you must design and implement an `Environment` class as one of the project deliverables.

- Reduction: Produces a simpler but equivalent expression by applying algebraic identities. The signature for this operation is:

```
Expr* reduce(void) const;
```

Your implementation must support all of the algebraic identities in **Table 1** and must reduce any expression that contains no variables with the literal that results from evaluating that expression.

$E + 0.0 \longrightarrow E$	$0.0 - E \longrightarrow -E$	$E * 0.0 \longrightarrow 0.0$
$E - 0.0 \longrightarrow E$	$0.0 * E \longrightarrow 0.0$	$E * 1.0 \longrightarrow E$
$0.0 + E \longrightarrow E$	$1.0 * E \longrightarrow E$	

Table 1: Table of algebraic identities. E is an arbitrary expression

Additional requirements:

- To minimize storage requirements, your solution must allow for two Expr trees to share sub-expressions without duplicating these. Thus, an Expr tree is more properly an Expr DAG (directed, acyclic graph).
- To prevent memory leaks, these classes should support (and your library code must properly use) the reference counting idiom that we discussed in class.
- To simplify grading, please name your classes as dictated by the list of names in **Table 2**.
- Your solution must include a separate .h and .cc file for each class and any other classes you see fit to invent to solve this problem.
- You must provide a “driver” program, which provides a function main that exercises each of the capabilities described herein.
- You must include a Makefile for recompiling these classes and linking them with the example main program that we provide.
- Please note that you are **not** required to write any software to construct terms from character input. Rather, you must construct them by writing little programs that explicitly allocate instances of these classes and inter-connect these instances appropriately.

Extra credit: Notice that each operation is declared `const`, suggesting that it should not modify the target expression (i.e., the expression upon which the operation is invoked). We will give extra credit to solutions that satisfy all of the above requirements and (in addition) are able to store `const` points to sub-expressions, so as to enable code like:

```
const Expr* e1 = new Variable("x");
const Expr* e2 = new Literal(2.0);
const Expr* e3 = new Add(e1, e2);
```

Hint: you will need to make use of the C++ type qualifier `mutable`.

Add	Cos	Environment	Exponentiate	Expr
Literal	Ln	Multiply	Negate	Power
Sin	Subtract	Variable		

Table 2: Naming conventions for classes in this assignment
