This was my experiment-with-advanced-Haskell project. Here, I created a simple, expression-based language that can be run from GHCI, which lets the user evaluate expressions involving a single variable x with the following features:

1. Trigonometric (sin, cos, tan), logarithmic (ln, logb where b is the base), derivatives, integrals, along with binary expressions that add, multiply, divide and exponentiate expressions together. See algExpr.hs for the full specification of the available expressions, including the precedence of the various operators.

2. Operations that simplify algebraic expressions. For now these include the ability to combine like terms, including those that are added and multiplied together (see below for more details), to evaluate derivatives of an arbitrary power, to clean up redundancies in algebraic expressions (e.g. something like turning $x + 0$ to x), and to do very basic integration (e.g. $\int x^n$, except that there is no "+C" term here as the language so far was only intended to compute just anti-derivatives).

3. Evaluation of expressions for some specified x. Note that if there is any integral in the expression, then the evaluation fails (as integration is very difficult to do symbolically)

Some example expressions that would work in my language are:

$$f(x) = e^x \sin^2(x) + \frac{d^2y}{dx^2}\left(x^3 * \cos(x) * x^2 * \cos(x)^3 * 3 * \cos(x) + 3e^{\sin(x)}\right) + \int x^3$$

which would be written as

```
e^x * sin(x)^2 + d^2y/dx^2(x^3*cos(x)*x^2*cos(x)^3*3*cos(x) + 3*e^(sin(x))) + I(x^3)
```

in my language. Note that there can be any amount of space (' ', not "\n" ☹) between terms that are added together, the parser will just filter those out of the input and still get the overall expression. Using some of the simple reductions and the integral evaluator (the derivative is really ugly), the above would be reduced to:

```
((((e^x)*(sin(x)^2.0))+d^2y/dx^2(((((x^5.0)*3.0)*(cos(x)^5.0))+(3.0*(e^sin(x))))))+((x^4.0)/4.0)
```

which is just

$$f(x) = e^x \sin^2(x) + \frac{d^2y}{dx^2}\left(3x^5 \cos^5 x + 3e^{\sin(x)}\right) + \frac{x^4}{4}$$

or as another example,

$$f(x) = x^2 + x + 2x + \frac{d^2y}{dx^2}(x^4) - 3x + 10x^2 + 0 * 1 * 3 + \ln(x) * \ln(x) * \cos(x) * \ln(x)^x$$

Would be reduced to (when evaluating the derivative and doing some addition and multiplication reduction):

```
(((((13.0*(x^2.0))+(x*3.0))-(3.0*x))+(10.0*(x^2.0)))+(cos(x)*(ln(x)^(x+2.0))))
```

Which is

$$f(x) = 13x^2 + 3x - 3x + 10x^2 + \cos(x)\ln(x)^{x+2}$$

Although not completely simplified, it is equivalent to the original expression! I think the reason it isn't here is because of the subtraction term (I did not implement negative numbers here). Anyways, I wrote the parser for this language using an existing homework assignment from CIS 194 as inspiration (I know there's a ParseC library out there, but I wanted to gain more experience with monads and applicative functors when doing this). The algorithm for simplification is straightforward for evaluating a derivative (just recurse to the appropriate subtrees), along with integration, but the simplification for the multiplication and the addition is a bit more involved. The algorithm I used was something as follows, using the following expression below as an example:

$$f(x) = x * 3 * 6 * 8 * 10 * x + x * 8 * x + x * 3 * 6 * x$$

The simplification was done one at a time – multiplication then addition. With the multiplication part, each subtree in the "+" (since * has higher precedence) is simplified separately. Let's use the left one as an example:

$$x * 3 * 6 * 8 * 10 * x$$

Here, the multiplication part of the tree was flattened to compose of each of the sub-expressions (the operators to each "*"):

$$[x, 3, 6, 8, 10, x]$$

Then an array of "merged" expressions was maintained. Initially, it was the first expression from the list:

$$[x] < -Merged\ terms$$
$$[3,6,8,10,x] < -Remaining\ terms$$

The next remaining term would look for all of its "like" terms in the merged terms. It would then "merge" with these terms and concatenate with the other merged terms – if it has no like terms, then it with the entire list of merged terms. For our example above, "3" is not "like" x so it would not combine with it. So we would have:

$$[x, 3] < -Merged\ terms$$
$$[6, 8, 10, x] < -Remaining\ terms$$

6 would then search through the merged terms and find 3 as its like. So we would have

$$foldl\ merge\ 6\ [3]$$

And the resulting merge would yield

$$[18]$$

While the remaining "non-like" term would be

$$[x]$$

So that the next set of merged terms is

$$[18, x]$$
$$[8, 10, x] < -Remaining\ terms$$

Then we would have 8 look for its like terms, merge with those (if any), and we'd repeat until we'd end up with a bunch of merged expressions. For our example,

$$[x^2, 1440]$$

would be what's left. We would then "combine" the merged expressions together, where a single expression would be itself, and a list of more than one expression would combine the relevant operator, here multiplication. In general, "liked" terms are terms that are structurally equivalent (have the same AST, or for commutative operations like addition and multiplication, have the same ordering of the AST), and are also "similar" (e.g. in multiplication, x^n and x can be considered like terms that, when multiplied, yield x^{n+1}).

For our example, we would get:

$$1440 * x^2$$

as our final result. The addition algorithm is similar.

Note that the above algorithm is not perfect and not very efficient, but it does do a decent job of simplifying easier, commonly found expressions.

In any case, the code is organized as follows:

1. algExpr.hs

   -This stores the type class for the algebraic expressions, and lists the exact expressions that the language will have (along with some type class instantiations)

2. parser.hs

   -This stores the definition of the Parser, along with some relevant type class instances so that it can be used like a monad.

3. parserUtils.hs

   -Stores helpful parser functions that could be shared across files.

4. algExprParser.hs

   -Stores the main parser for the expression language. The grammar for the language is also described here in the comments.

5. algExprReducUtils.hs

   -This stores common helper functions to aid reducing the expression, either by combining like terms, evaluating derivatives or evaluating integrals.

6. algExprAddMul.hs, algExprReducDeriv.hs, algExprReducInteg.hs

   -Stores the code for combining like terms in addition and multiplication, for evaluating derivatives, and for evaluating integrals.

7. algExprEval.hs

   -This will evaluate an algebraic expression for a given value of x (so long as there aren't any integrals in it). Note that the method evalExprFromString lets you directly enter an expression into GHCI along with an x value and see it evaluated.