Problem Set: Symbolic differentiation

Stuart M. Shieber

February 24, 2023

Solving an equation like $x^2 = x + 1$ NUMERICALLY yields a particular number as an approximation to the solution for x, for instance, 1.618. Solving the equation SYMBOLICALLY yields an expression representing the solution exactly, for instance, $\frac{1+\sqrt{5}}{2}$. (The golden ratio! See Exercise **??**.) The earliest computing devices were used to calculate numerically. Charles Babbage envisioned his analytical engine as a device for calculating numeric tables, and Ada Lovelace's famous program for Babbage's analytical engine numerically calculated Bernoulli numbers.

But Lovelace (Figure 1) was perhaps the first computer scientist to have the revolutionary idea that computers could be used for much more than numerical calculations.

Figure 1: A rare daguerrotype of Ada Lovelace (Augusta Ada King, Countess of Lovelace, 1815–1852) by Antoine Claudet, taken c. 1843, around the time she was engaged in writing her notes on the Babbage analytical engine. [Menabrea and Lovelace, 1843]

> The operating mechanism…might act upon other things besides *number*, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. [Menabrea and Lovelace, 1843, page 694]

One of the applications of the power of computers to transcend numerical calculation, which Lovelace immediately saw, was to engage in mathematics symbolically rather than numerically.

> It seems to us obvious, however, that where operations are so independent in their mode of acting, it must be easy by means of a few simple provisions and additions in arranging the mechanism, to bring out a *double* set of *results*, viz. – 1st, the *numerical magnitudes* which are the results of operations performed on *numerical data*. (These results are the *primary* object of the engine). 2ndly, the symbolical results to be attached to those numerical results, which symbolical results are not less the necessary and logical consequences of operations performed upon *symbolical data*, than are numerical results when the data are numerical. [Menabrea and Lovelace, 1843, page 694–5]

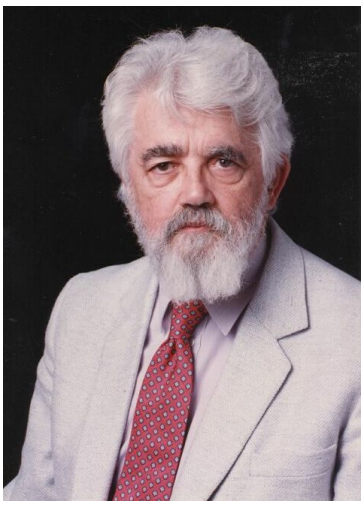

Figure 2: John McCarthy (1927–2011), one of the founders of (and coiner of the term) artificial intelligence. His LISP programming language was widely influential in the history of programming languages. He was awarded the Turing Award in 1971.

The first carrying out of symbolic mathematics by computer arose over a hundred years later, in the work of Turing-Award-winning computer scientist John McCarthy (Figure 2). In the summer of 1958, McCarthy made a major contribution to the field of programming languages. With the objective of writing a program that performed

symbolic differentiation (that is, the process of finding the derivative of a function) of algebraic expressions in an effective way, he noticed that some features that would have helped him to accomplish this task were absent in the programming languages of that time. This led him to the invention of the programming language LISP [McCarthy, 1960] and other ideas, such as the concept of list processing (from which LISP derives its name), recursion, and garbage collection, which are essential to modern programming languages.

McCarthy saw that the power of higher-order functional programming, together with the ability to manipulate structured data, make it possible to carry out such symbolic mathematics in an especially elegant manner. However, it was Jean Sammet (Figure 3) who first envisioned a full system devoted to symbolic mathematics more generally. Her FORMAC system [Sammet, 1993] ushered in a wave of symbolic mathematics systems that have made good on Lovelace's original observation. Nowadays, symbolic differentiation of algebraic expressions is a task that can be conveniently accomplished on modern mathematical packages, such as Mathematica and Maple.

This assignment focuses on using abstract data types to design your own mini-language – a mathematical expression language over which you'll perform symbolic mathematics by computing derivatives symbolically.

Figure 3: Jean Sammet (1928–2017), head of the FORMAC project to build "the first widely available programming language for symbolic mathematical computation to have significant practical usage" [Sammet, 1993]. She was awarded the Augusta Ada Lovelace Award in 1999 and the Computer Pioneer Award in 2009 for her work on FORMAC and (with Admiral Grace Hopper) the programming language COBOL.

1 A language for symbolic mathematics

In this section, your mission is to define a language of mathematical expressions representing functions of a single variable, and write code that can differentiate (that is, take the derivative of) and evaluate those expressions. Symbolic expressions consist of numbers, variables, and standard numeric functions (addition, subtraction, multiplication, division, exponentiation, negation, trigonometric functions, and so forth) applied to them.

It may have been a long time since you thought about differential calculus or trigonometric functions or taking a derivative. In fact, maybe you've never studied any of that. Have no fear! This problem set isn't really about calculus or trig. Rather, it's about *manipulating representations of expressions*. We give you all of the formulas you need to do the symbolic manipulations; all you need to do is represent them in OCaml.

1.1 Conceptual Overview

We want to be able to manipulate symbolic expressions such as $x^2 + \sin(-x)$, so we'll need a way of representing expressions as data

in OCaml. For that purpose, we use OCaml types to define the appropriate data structures. The `expression` data type allows for four different kinds of expressions: numbers, variables, and unary and binary operator expressions. For our purposes, only one variable (call it x) is needed, and it will be represented by the `Var` constructor for the `expression` type. Numbers are represented with the `Num` constructor, which takes a single float argument to specify which number is being denoted. Binary operator expressions, in which a binary operator like addition or division is applied to two subexpressions, is represented by the `Binop` constructor, and similarly for unary operators like sine or negation, which take only a single subexpression.

The `expression` data type can therefore be defined as follows (and as provided to you in the file `expressionLibrary.ml`):

```
(* Binary operators. *)
type binop = Add | Sub | Mul | Div | Pow ;;

(* Unary operators. *)
type unop = Sin | Cos | Ln | Neg ;;

(* Expressions *)
type expression =
  | Num of float
  | Var
  | Binop of binop * expression * expression
  | Unop of unop * expression ;;
```

For instance, the mathematical expression x^{-2} would be represented by this OCaml value:

```
Binop (Pow, Var, Unop (Neg, Num 2))
```

You can think of the data objects of this `expression` type as defining trees where nodes are the type constructors and the children of each node are the specific operator to use and the arguments of that constructor. These are just the abstract syntax trees of Section **??**3.3.

Although numeric expressions frequently make use of parentheses – and sometimes necessarily so, as in the case of the expression $(x+3)(x-1)$ – the `expression` data type definition has no provision for parenthesization. Why isn't that needed? It might be helpful to think about how this example would be represented.

1.2 Provided code

We have provided some functions to create and manipulate `expression` values. The `checkexp` function is contained in `expression.ml`. The others are contained in `expressionLibrary.ml`. Here, we provide a brief description of them and some example evaluations.

- `parse`: Translates a string in infix form (such as `"x^2 + sin(~x)"`) into an `expression` (treating `"x"` as the variable). (The function uses "`~`" for unary negation rather than "–", to make it distinct from binary subtraction.) The `parse` function parses according to the standard order of operations – so `"5+x*8"` will be read as `"5+(x*8)"`.

```
# parse ("5+x*8") ;;
- : ExpressionLibrary.expression =
Binop (Add, Num 5., Binop (Mul, Var, Num 8.))
```

- `to_string`: Returns a string representation of an expression in a readable form, using infix notation. This function adds parentheses around every binary operation so that the output is completely unambiguous.

```
# let exp = Binop (Add,
#                  Binop (Pow, Var, Num 2.0),
#                  Unop (Sin, Binop (Div, Var, Num 5.0))) ;;
val exp : ExpressionLibrary.expression =
  Binop (Add, Binop (Pow, Var, Num 2.), Unop (Sin, Binop (Div, Var,
  Num 5.)))
# to_string exp ;;
- : string = "((x^2.)+(sin((x/5.))))"
```

- `to_string_smart` : Returns a string representation of an expression in an even more readable form, only adding parentheses where needed to override associativity and precedence.

```
# to_string_smart exp ;;
- : string = "x^2.+sin(x/5.)"
```

- `rand_exp` : Takes a length l and returns a randomly generated `expression` of length at most 2^l. Useful for generating expressions for debugging purposes.

```
# let () = Random.init 2 (* for consistency *) ;;
# rand_exp 5 ;;
- : ExpressionLibrary.expression =
Binop (Mul, Unop (Ln, Num (-11.)), Binop (Sub, Var, Num (-17.)))
# rand_exp 5 ;;
- : ExpressionLibrary.expression = Binop (Mul, Num 4., Var)
```

- `rand_exp_str` : Takes a length l and returns a string representation of length at most 2^l.

```
# let () = Random.init 2 (* for consistency *) ;;
# rand_exp_str 5 ;;
- : string = "ln(~11.)*(x-~17.)"
# rand_exp_str 5 ;;
- : string = "4.*x"
```

- `checkexp` : Takes a string and a value and prints the results of calling every function to be tested except `find_zero`.

1.3 Simple expression manipulation

Start by implementing two functions that perform simple expression manipulation. The function `contains_var : expression -> bool` returns `true` when its argument contains a variable (that is, the constructor `Var`). The function `evaluate : expression -> float -> float` takes an expression and a numeric value for the variable in the expression and returns the numerical evaluation of the expression at that value.

As a helpful note, in testing `evaluate`, rather than testing that you get a particular `float` value, you may want to use `unit_test_within` to verify that the value is within a certain tolerance of the answer you expect. This is necessary to avoid small differences due to the imprecision of `float` arithmetic.

1.4 Symbolic differentation

Next, we want to develop a function that takes an expression `e` as its argument and returns an expression `e'` representing the derivative of the expression with respect to `x`. This process is referred to as symbolic differentiation.

When implementing this function, recall the chain rule from your calculus course:

$$(f(g(x)))' = f'(g(x)) \cdot g'(x)$$

Using the chain rule, we can write the derivatives for the other functions in our language, as shown in Figure 4.[1]

[1] If the kinds of notation used here are unfamiliar, the discussion in Section **??**B.1.4 may be helpful.

We've provided two cases for calculating the derivative of $f(x)^{g(x)}$, one for where $g(x)$ is an expression (h) that contains no variables, and one for the general case. The first is a special case of the second, but it is useful to treat them separately, because when the first case applies, the second case produces unnecessarily complicated expressions.

Your task is to implement the `derivative` function whose type is `expression -> expression`. The result of your function must be correct, but need not be expressed in the simplest form. Take advantage of this in order to keep the code in this part as short as possible.

Once you've implemented the `derivative` function, you should be able to calculate the symbolic derivative of various functions. Here's an example, calculating the derivative of $x^2 + 5$:

```
# to_string_smart (derivative (parse "x^2 + 5")) ;;
- : string = "2.*1.*x^(2.-1.)+0."
```

$$
\begin{aligned}
(f(x)+g(x))' &= f'(x)+g'(x)\\
(f(x)-g(x))' &= f'(x)-g'(x)\\
(f(x)\cdot g(x))' &= f'(x)\cdot g(x)+f(x)\cdot g'(x)\\
\left(\frac{f(x)}{g(x)}\right)' &= \frac{(f'(x)\cdot g(x)-f(x)\cdot g'(x))}{g(x)^2}\\
(\sin f(x))' &= f'(x)\cdot\cos f(x)\\
(\cos f(x))' &= f'(x)\cdot\sim\sin f(x)\\
(\ln f(x))' &= \frac{f'(x)}{f(x)}\\
(f(x)^h)' &= h\cdot f'(x)\cdot f(x)^{h-1}\\
&\qquad \text{where } h \text{ contains no variables}\\
(f(x)^{g(x)})' &= f(x)^{g(x)}\cdot\left(g'(x)\cdot\ln f(x)+\frac{f'(x)\cdot g(x)}{f(x)}\right)\\
(n)' &= 0 \qquad \text{where } n \text{ is any constant}\\
(x)' &= 1
\end{aligned}
$$

Figure 4: Rules for taking derivatives for a variety of expression types.

The result generated, $2\cdot 1\cdot x^{2-1}+0$, isn't in its simplest form, but it does correctly capture the derivative, $2\cdot x$.

To make your task easier, we have provided an outline of the function with many of the cases already filled in. We also provide a function, `checkexp`, which checks the functions you write in Problems 1–3 for a given input. The portions of the function that require your attention currently read `failwith "not implemented"`.

1.5 Zero-finding

An application of the derivative of a function is to numerically calculate the ZEROS of a function, the values of its argument that the function maps to zero. One way to do so numerically is Newton's method.

Newton's method to find a zero of a function f works by starting with a guess x_0 and repeatedly calculating better and better estimates of the zero $x_1, x_2, x_3, \ldots$. Starting with the initial guess x_0, Newton's method proceeds by calculating the x_n values according to the recurrence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad ,$$

continuing until a good enough x is found. In determining if an x is

"good enough", we make use of a small threshold ϵ below which we are satisfied with an estimate's closeness to a zero; in particular, we seek a value x such that $|f(x)| < \epsilon$, that is, the value that the expression evaluates to at x is "within ϵ of 0". (We are *not* using the more demanding requirement of seeking an x such that $|x - \hat{x}| < \epsilon$ for some $\hat{x}$ for which $f(\hat{x}) = 0$.)

Notice that the reestimation process in Newton's method crucially relies on the ability to evaluate both the function f itself and its derivative f'. Fortunately, you've already implemented functions to evaluate expressions and to calculate their derivatives.

Your remaining task is then to implement the function `find_zero : expression -> float -> float -> int -> float option`. This function should take an expression (representing f), a starting guess for the zero (x_0), a precision requirement (ϵ), and a limit on the number of times to repeat the reestimation process. It should return `None` if no zero was found within the desired precision by the time the limit was reached, and `Some x` if a zero was found at `x` within the desired precision. (You should return the first such value; there's no need to exhaust the maximum number of reestimations.) As an example of the use of `find_zero`, we can find the zero of $3x-1$, starting the search around 0 and looking for an estimate accurate to 0.0001 with a maximum of 100 reestimations:

```
# find_zero (parse "3 * x - 1") 0. 0.00001 100 ;;
- : float option = Some 0.333333333333333315
```

Note that there are cases where Newton's method will fail to produce a zero, such as for the function $x^{1/3}$. You are not responsible for finding a zero in those cases, but just for the correct implementation of Newton's method.

1.6 Challenge problem: Symbolic zero-finding

If you find yourself with plenty of time on your hands after completing the problem set to this point and submitting it successfully, feel free to try this week's extra challenge problem, which is completely optional but good for your karma.

The function you wrote above allows you to find the zero (or a zero) of most functions that can be represented with our expression language. This makes it quite powerful. However, in addition to numeric solving like this, Mathematica and many similar programs can perform symbolic algebra. These programs can solve equations using techniques similar to those you learned in middle and high school (as well as more advanced techniques for more complex equations) to get exact, rather than approximate answers. For example, given the

expression representing $3x - 1$, your `find_zero` function might return something like `0.33333`, depending on your value of ϵ. The exact solution, however is given by the expression `1/3`, and this answer can be found by a program that solves equations symbolically.

Performing the symbolic manipulations on complex expressions necessary to solve equations is quite difficult in general, and we do not expect you to handle the general case. However, there is one type of expression for which symbolic zero-finding is not so difficult. These are *expressions of degree one*, those that can be simplified to the form $a \cdot x + b$, where the highest exponent of the variable is 1. You likely learned how to solve equations of the form $a \cdot x + b = 0$ years ago, and can apply the same skills in writing a program to solve these.

Write a function, `find_zero_exact` which will exactly find the zero of degree one expressions. Your function should, given a valid input expression that has a zero, return `Some exp` where `exp` is an expression that contains no variables, evaluates to the zero of the given expression, and is exact. If the expression is not degree one or has no zero, it should return `None`.

You need not return the simplest expression, though it could be instructive to think about how to simplify results. For example, `find_zero_exact (parse "3*x-1")` might return `Binop (Div, Num 1., Num 3.)` or `Unop(Neg, Binop (Div, Num -1., Num 3.))` but should *not* return `Num 0.333333333` as this is not exact.

Note that degree-one expressions need not be as simple as $ax + b$. Something like $5x - 3 + 2(x - 8)$ is also a degree-one expression, since it can be simplified to $ax + b$ by distributing and simplifying. You may want to think about how to handle these types of expressions as well, and think more generally about how to determine whether an expression is of degree one.

Hint: You may want to start by writing a function that will crawl over an expression and distribute any multiplications or divisions, resulting in something of a form like $ax + b$ (or maybe $ax + bx + cx + d + e + f$ or similar).

References

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4): 184–195, April 1960. ISSN 0001-0782. DOI: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

Luigi Federico Menabrea and Ada Lovelace. Sketch of the analytical engine invented by charles babbage with notes by the translator. translated by ada lovelace. In Richard Taylor, editor, *Scientific Mem-*

oirs, volume 3, pages 666–731. Richard and John E. Taylor, London, 1843. URL http://nrs.harvard.edu/urn-3:FHCL.HOUGH:33047333.

Jean E. Sammet. The beginning and development of FORMAC (FORmula MAnipulation Compiler). In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 209–230, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. DOI: 10.1145/154766.155372. URL http://doi.acm.org/10.1145/154766.155372.

Image Credits

Version information: Commit c4f9d02 from 2023-02-23 by Brian Yu. CI build of I_TAG ().