CS51 PROBLEM SET 2: HIGHER-ORDER FUNCTIONAL PROGRAMMING AND SYMBOLIC DIFFERENTIATION

STUART M. SHIEBER

Introduction

This assignment focuses on programming in the functional programming paradigm, with special attention to the idiomatic use of higher-order functions like `map`, `fold`, and `filter`. In doing so, you will exercise important features of functional languages, such as recursion, pattern matching, and list processing, and you will gain the ability to design your own mini-language – a mathematical expression language over which you'll compute derivatives symbolically.

Setup. First, create your repository in GitHub Classroom for this homework by following this link. Then, follow the GitHub Classroom instructions found here.

For this assignment, you will be working in two different files:

- `mapfold.ml`: A self-contained series of exercises involving higher order functions.
- `expression.ml`: A number of functions that will allow you to perform symbolic differentiation of algebraic expressions. Support code can be found in `expressionLibrary.ml`.

Reminders.

Compilation errors: In order to submit your work to the course grading server, *your solution must compile against our test suite.* This problem set has two parts. If either part does not compile, you will receive *no credit* for that part of the problem set. However, you will still receive credit for any parts that do successfully compile. If there are problems that you are unable to solve, you must still write a function that matches the expected type signature, or your code will not compile. (When we provide stub code, that code will compile to begin with.) If you are having difficulty getting your code to compile, please visit office hours or post on Piazza. *Emailing your homework to your TF or the Head TFs is* not *a valid substitute for submitting to the course grading server. Please start early, and submit frequently, to ensure that you are able to submit before the deadline.*

Testing is required: As with the previous problem set, we ask that you explicitly add tests to your code in the files `mapfold_tests.ml` and `expression_tests.ml`.

Start early: Part 2 in particular may prove quite challenging, so we encourage you to start early and ask questions if you get stuck.

Date: February 6, 2018.

Design and style: Good design and style are important aspects of this problem set and all problem sets in the course. As such, design and style will be a significant portion of your grade. Please refer to the CS51 Style Guide or CSCI E-51 Style Guide to help ensure the quality of your code.

As in the previous problem set, your submission should have the following properties:

- Your code compiles without warnings.
- Your code adheres to the CS51 Style Guide or CSCI E-51 Style Guide.
- You have written one test per code path per function.
- Your functions have the correct names and type signatures. Do not change the signature of a function that we have provided.

1. Compilation with `make`

In your last problem set, we provided a Makefile that allowed you to compile your code without directly calling `ocamlbuild`, the compiler that takes your `.ml` files and turns them into executable `.byte` files.

For this problem set, a Makefile might again prove helpful in compiling your code, especially if you don't want to compile it all at once! This time, you will be given the opportunity to write a Makefile yourself. Take a look at the CS51 reference or CSCI E-51 reference for instructions on how to create a Makefile.

2. Higher order functional programming (`mapfold.ml`)

Mapping, folding, and filtering are important techniques in functional languages that allow the programmer to abstract out the details of traversing and manipulating lists. Each can be used to accomplish a great deal with very little code. In this part, you will create a number of functions using the higher-order functions `map`, `filter`, and `fold`. In OCaml, these functions are available as `List.map`, `List.filter`, `List.fold_right`, and `List.fold_left` in the `List` module.

The file `mapfold.ml` contains starter code for a set of functions that operate on lists. For each one, you are to provide the implementation of that function using the higher-order (mapping, folding, filtering) functions directly. The point is to use the higher-order functional programming paradigm idiomatically. The problems will be graded accordingly: a solution, even a working one, that does not use the higher-order functional paradigm, deploying these higher-order functions properly, will receive little or no credit. For instance, solutions that require you to change the `let` to a `let rec` show that you haven't assimilated the higher-order functional paradigm. However, you should feel free to use functions you've defined earlier to implement others later where appropriate.

Remember to provide unit tests for all of the functions in `mapfold.ml` in a file `mapfold_tests.ml`. For this purpose, you should write a function `test: unit -> unit` that executes a series of tests using the `assert: bool -> unit` function. The `assert` function simply returns `()` if its argument is `true`, but if it is `false` it raises an exception. We have included an example of its use in the starter code for `mapfold_tests.ml`. You should provide at least one test per code path (that is, every match case and if branch) for every function that you write on this problem set.

3. A language for symbolic mathematics (`expression.ml`)

In the summer of 1958, John McCarthy (recipient of the Turing Award in 1971) made a major contribution to the field of programming languages. With the objective of writing a program that performed symbolic differentiation 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 (published in *Communications of the ACM* in 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. Nowadays, symbolic differentiation of algebraic expressions is a task that can be conveniently accomplished on modern mathematical packages, such as Mathematica and Maple.

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.

3.1. 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 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 object:

```
  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. Such trees are called *abstract syntax trees* (or AST).

Although numeric expressions frequently make use of parentheses – and sometimes necessarily so, as in the case of the expression $(x + 3)(x - 1)$ – the 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.

3.2. **Compiling and testing.** The code you will be augmenting is in `expression.ml`.

You can compile and test from the terminal command line. To use this method, compile your code from the command line using `ocamlbuild` or `make`, for instance,

```
$ ocamlbuild expression.byte
```

Once you have done this, you can run the compiled code with

```
$ ./expression.byte
```

You should provide a complete set of unit tests for your code in the file `expression_tests.ml`, using `assert` to specify the tests. (For the adventuresome, we also provide a `verify` function in the CS51 module, which is a bit more pleasant for writing unit tests. Feel free to experiment with that instead.)

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") ;;
  - : 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 : 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 when there may be ambiguity.
  ```
  # 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 ;;
  - : expression =
  Binop (Mul, Unop (Ln, Num (-11.)), Binop (Sub, Var, Num (-17.)))
  # rand_exp 5 ;;
  - : 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`.

3.3. **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 `assert` that it is within a small threshold (conventionally referred to as ϵ or "epsilon") of the answer you expect. This is necessary to avoid small differences due to the imprecision of `float` arithmetic, and can save you a headache down the road.

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

FIGURE 1. Rules for taking derivatives for a variety of expression types.

3.3.1. *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 that, we can write the derivatives for the other functions in our language, as shown in Figure 1.

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.

To make your task easier, we have provided an outline of the function with many of the cases already filled in. We have also provide a function, `checkexp`, which checks the functions you write in Problems 2.1–2.3 for a given input. The portions of the function that require your attention currently read `failwith "not implemented"`.

3.4. **Zero-finding.** One application of the derivative of a function is to find zeros of a function. One way to do so is Newton's method. Your task is to implement the function `find_zero : expression -> float -> float -> int -> float option`.

This function should take an expression, a starting guess for the zero, a precision requirement, and a limit on the number of times to repeat the process. It should return `None` if no zero was found within the desired precision by the time the limit was reached, and `Some r` if a zero was found at `r` within the desired precision.

If the expression that `find_zero` is operating on represents $f(x)$ and the precision is ϵ, we are asking you to find 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* requiring you to find an x such that $|x - x_0| < \epsilon$ for some x_0 for which $f(x_0) = 0$.

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.

3.5. **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).

4. Submission

Before submitting, please estimate how much time you spent on the problem set by editing the lines in each file that look like

```
let minutes_spent_on_partX () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) this part of the problem set took you to complete. For example, if you spent 6 hours on a part, you should change the line to:

```
let minutes_spent_on_partX () : int = 360
```

Make sure your code still compiles. Then, to submit the problem set, follow the instructions found in the CS51 Reference or CSCI E-51 Reference.