CS 440 MP2

Overview

In this machine problem you will be writing and making use of higher order functions -- an important class of functions discussed in class. You'll also be writing functions that perform *symbolic manipulation* --- a step in the direction of implementing compilers/interpreters.

All the code you write for this machine problem will go into the "mp2.rkt" file. In that file you will find stubs for each of the programming exercises described below.

Exercises

Part 1: HOFs (and some utility functions) (25 points)

- `deep-map`: takes a function `fn` and list `lst`, and applies `fn` (recursively) to every element of `lst`, returning a list of the same "shape" as the original. Note that `lst` does not need to be a proper list, and the behavior of `deep-map` is unlike `map`, in that the latter will not recursively apply its argument function.

```
> (deep-map add1 (cons 1 2))
'(2 . 3)

> (deep-map string-length '("hello" ("how" . "are") (("you") "today?")))
'(5 (3 . 3) ((3) 6))
```

- `my-curry`: implement your own version of `curry`. Your version should behave identically to `curry`, but need only work on procedures with fixed arity. E.g.,

```
> ((my-curry even?) 2)
#t
> (((my-curry cons) 1) 2)
'(1 . 2)
> ((my-curry cons 1) 2)
'(1 . 2)
```

  For this exercise you may (and will need to) use the built-in `procedure-arity` function.

- `lookup`: treats a list of pairs as a lookup structure (aka "associative list"); when given a key and a list, it finds the pair whose `car` matches the key (using `equal?`) and returns the associated `cdr`. If no match exists, returns `#f`. E.g.,

```
> (lookup 'a '((a . apple) (b . bee) (c . cat)))
'apple

> (lookup 2 '((1 . "one") (2 "two" "three") (4 "four" "five")))
'("two" "three")
```

```
> (lookup 'foo '((a . apple) (2 . "two")))
#f
```

- `update`: updates or inserts a new pair into a lookup list (as used by `lookup`) reflecting a provided key/value mapping. (The location of a newly inserted pair doesn't matter.) E.g.,

```
> (update 'a 'apple '((b . bee) (c . cat)))
'((b . bee) (c . cat) (a . apple))

> (update 'a "auto" '((a . apple) (b . bee) (c . cat)))
'((a . "auto") (b . bee) (c . cat))

> (update 1 (list 100 200 300) '())
'((1 100 200 300))
```

- `make-object`: creates and returns a rudimentary "object" as a closure over an associative list containing attributes. The returned closure (i.e., function) responds to three "messages": `get`, `set`, and `update`.
    - `get` takes an attribute key, whose value is returned (if it exists).
    - `set` takes an attribute key and value, which are used to update the associative list
    - `update` takes an attribute key and a function which is used to update the associative list by applying it to the current value

  `make-object` takes one argument, the initial name of the object (associated with the attribute key `'name`).

  E.g.,

```
> (define obj (make-object "foo"))
> (obj 'get 'name)
"foo"
> (obj 'set 'name 'bar)
> (obj 'get 'name)
'bar
> (obj 'set 'x 42)
> (obj 'update 'x (lambda (n) (* n 100)))
> (obj 'get 'x)
4200
```

Part 2: Symbolic differentiation (10 points)

For this part you will start by implementing a function that performs symbolic differentiation -- i.e., that determines the derivative of a function with respect to some variable. The rules you need to implement are limited to:

1. $\frac{d}{dx}C = 0$, where C is a constant
2. $\frac{d}{dx}f = 0$, where f is independent of x
3. $\frac{d}{dx}x = 1$
4. $\frac{d}{dx}x^n = n \cdot x^{n-1}$
5. $\frac{d}{dx}(e_1 + e_2 + \cdots) = \frac{d}{dx}e_1 + \frac{d}{dx}e_2 + \cdots$
6. $\frac{d}{dx}(e_1 \cdot e_2) = e_1 \cdot \frac{d}{dx}e_2 + e_2 \cdot \frac{d}{dx}e_1$

Your function, named `diff`, will take a variable `var` and expression `exp` (expressed in prefix form as a list, using `+`, `*`, `^`, to denote addition, multiplication, and exponentiation), and return the derivative of `exp` with respect to `var`. Note that `diff` does not need to return the derivative in algebraically simplified form (i.e., so long as your answer is algebraically correct, it will do).

E.g.,

```
> (diff 'x 10)
0

> (diff 'y 'x)
0

> (diff 'x '(^ x 4))
'(* 4 (^ x 3))

> (diff 'y '(+ x y))
'(+ 0 1)

> (diff 'x '(* 3 x))
'(+ (* 3 1) (* x 0))

> (diff 'x '(+ (^ x 2) (* 5 x) 10))
'(+ (* 2 (^ x 1)) (+ (* 5 1) (* x 0)) 0)
```

Part 3: Meta-circular Evaluator (10 points)

In this part you will implement your own limited version of `eval`. Your version will only understand a very limited subset of Racket, with expressions of the following kind:

```
expr ::= (lambda (var) expr)     ; lambda (function) definition
       | (expr expr)             ; function application
       | var                     ; variable reference
```

Note that a `lambda` may only have one parameter.

Your evaluation function will be passed s-expressions of the above form (the base form will always be a lambda), and should return corresponding Racket values. Because your evaluator will use Racket features to represent and implement this subset of the Racket language, it is a so-called *meta-circular evaluator*.

To implement this, you will need to take apart the input to your evaluator and determine which form it is before building the corresponding Racket form. In order to implement variables (which are just symbols in the input), you will also need to keep track of their bindings in an *environment* -- we recommend using the associative list which you implemented above.

Here's the evaluator in action:

```
> ((my-eval '(lambda (x) x)) 10)
10

> (((my-eval '(lambda (f) (lambda (x) (f (f x))))) sqr) 5)
625
```

We've stubbed out portions of the evaluator for you as a guide. Feel free to delete/keep what you wish.

Part 4: Free Variables (10 points)

Lastly, you will implement a function `free`, which will take the same types of expressions understood by your evaluator, but will instead return a list of all the free variables found in those expressions (order doesn't matter).

You will likely be able to reuse much of your code from part 3!

Some sample results:

```
> (free 'a)
'(a)
> (free '(x y))
'(x y)
> (free '(lambda (x) x))
'()
> (free '(lambda (x) (x y)))
'(y)
> (free '(lambda (x) (w (lambda (w) (x y)))))
'(w y)
```

Testing

As before, we've provided you with test suites for most of the exercises in the "mp2-tests.rkt" source file.

Note that passing all the tests *does not guarantee full credit*! In particular, we will be hand-checking your symbolic differentiator and algebraic simplifier (described below).

Extra credit! (Up to 8 points)

If you're looking for a bit more practice/fun with symbolic manipulation, here it is.

Implement an algebraic simplification function named `simplify` for the differentiation function you wrote in part 2. At a minimum, your implementation should handle the following types of subexpressions:

- adding to 0
- multiplying by 1
- addition/multiplication involving only numeric operands

E.g., a properly implemented simplifier would work as follows:

```
> (simplify '(+ (* 2 x) 0))
'(* 2 x)

> (simplify '(* 1 (+ x 5)))
'(+ x 5)

> (simplify '(+ (* 0 x) (* 1 y)))
'y

> (simplify '(+ 1 (* 2 3) 4))
11
```

A more sophisticated simplifier would also handle exponentiation, combining like terms, etc. When all is said and done, `(simplify (diff expr))` would give us a much more satisfying symbolic differentiation experience. E.g.,

```
> (simplify (diff 'x '(+ (^ x 2) (* 5 x) (* 2 x) 10)))
'(+ (* 2 x) 7)
```

If you choose to tackle this exercise, complete the implementation of `simplify`. You should preface it with a comment that describes all the features of your implementation. You should also add (passing) test cases to `mp2-tests.rkt` that demonstrate its capabilities.

Submission

When you are done with your work, simply commit your changes and push them to our shared private GitHub repository. Please note that your submission date will be based on your most recent push (unless you let us know to grade an earlier version).