Homework 07：Scheme

1. Instructions

> Please download homework materials `hw07.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 3. The starter code for these problems is provided in `hw07.scm`, which is distributed as part of the homework materials in the `code` directory.

Submission: When you are done, submit your code to our Grader server as instructed in lab07 by `python submit.py --stuid <YOUR STUDENT ID> --stuname <YOUR NAME>`. You may submit more than once before the deadline; grader will record the highest score graded from your submissions. Check that you have successfully submitted your code and what your score is on Grader website.

> See lab07 for more instructions on on submitting assignments.

> **WARNING:** Do not modify `submit.py`!

Using Ok: If you have any questions about using Ok, please refer to this guide.

Readings: You might find the following references useful:

- Scheme Specification
- Scheme Built-in Procedure Reference

2. Scheme Editor

2.1 How to launch

In your `hw07` folder you will find a new editor. To run this editor, run `python editor`. This should pop up a window in your browser; if it does not, please navigate to localhost:31415 and you should see it.

Make sure to run `python ok` in a separate tab or window so that the editor keeps running.

2.2 Features

The `hw07.scm` file should already be open. You can edit this file and then run `Run` to run the code and get an interactive terminal or `Test` to `run` the ok tests.

`Environments` will help you diagram your code, and `Debug` works with environments so you can see where you are in it. We encourage you to try out all these features.

`Reformat` is incredibly useful for determining whether you have parenthesis based bugs in your code. You should be able to see after formatting if your code looks weird where the issue is.

> By default, the interpreter uses Lisp-style formatting, where the parens are all put on the end of the last line
>
> ```
> (define (f x)
> (if (> x 0)
>    x
>    (- x)))
> ```
>
> However, if you would prefer the close parens to be on their own lines as so
>
> ```
> (define (f x)
> (if (> x 0)
>    x
>    (- x)
> )
> )
> ```
>
> you can go to Settings and select the second option.

3. Required Problems

The following problems develop a system for symbolic differentiation of algebraic expressions. The `derive` Scheme procedure takes an algebraic expression and a variable and returns the derivative of the expression with respect to the variable. Symbolic differentiation is of special historical significance in Lisp. It was one of the motivating examples behind the development of the language. Differentiating is a recursive process that applies different rules to different kinds of expressions.

```
; derive returns the derivative of EXPR with respect to VAR
(define (derive expr var)
  (cond ((number? expr) 0)
        ((variable? expr) (if (same-variable? expr var) 1 0))
        ((sum? expr) (derive-sum expr var))
        ((product? expr) (derive-product expr var))
        ((exp? expr) (derive-exp expr var))
        (else 'Error)))
```

To implement the system, we will use the following data abstraction. Sums and products are lists, and they are simplified on construction:

```
; Variables are represented as symbols
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

; Numbers are compared with =
(define (=number? expr num)
```

```
  (and (number? expr) (= expr num)))

; Sums are represented as lists that start with +.
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
(define (sum? x)
  (and (list? x) (eq? (car x) '+)))
(define (first-operand s) (cadr s))
(define (second-operand s) (caddr s))

; Products are represented as lists that start with *.
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
(define (product? x)
  (and (list? x) (eq? (car x) '*)))
; You can access the operands from the expressions with
; first-operand and second-operand
(define (first-operand p) (cadr p))
(define (second-operand p) (caddr p))
```

> Note that we will not test whether your solutions to this question correctly apply the chain rule. For more info, check out the extensions section.

Problem 1: Derive Sum (100 pts)

Implement `derive-sum`, a procedure that differentiates a sum by summing the derivatives of the `first-operand` and `second-operand`. Use data abstraction for a sum.

> **Note:** the formula for the derivative of a sum is `(f(x) + g(x))' = f'(x) + g'(x)`

```
(define (derive-sum expr var)
  'YOUR-CODE-HERE
)
```

The tests for this section aren't exhaustive, but tests for later parts will fully test it.

Before you start, check your understanding by running

```
python ok --local -q derive-sum -u
```

To test your code, if you are in the local Scheme editor, hit `Test`. You can click on a case, press `Run`, and then use the `Debug` and `Environments` features to figure out why your code is not functioning correctly.

You can also test your code from the terminal by running

```
python ok --local -q derive-sum
```

Problem 2: Derive Product (100 pts)

> **Note:** the formula for the derivative of a product is `(f(x) g(x))' = f'(x) g(x) + f(x) g'(x)`

Implement `derive-product`, which applies the product rule to differentiate products. This means taking the first-operand and second-operand, and then summing the result of multiplying one by the derivative of the other.

> The `ok` tests expect the terms of the result in a particular order. First, multiply the derivative of the first-operand by the second-operand. Then, multiply the first-operand by the derivative of the second-operand. Sum these two terms to form the derivative of the original product. In other words, `f' g + f g'` not some other ordering

```
(define (derive-product expr var)
  'YOUR-CODE-HERE
)
```

Before you start, check your understanding by running

```
python ok --local -q derive-product -u
```

To test your code, if you are in the local Scheme editor, hit `Test`. You can click on a case, press `Run`, and then use the `Debug` and `Environments` features to figure out why your code is not functioning correctly.

You can also test your code from the terminal by running

```
python ok --local -q derive-product
```

Problem 3: Make Exp (200 pts)

Implement a data abstraction for exponentiation: a `base` raised to the power of an `exponent`. The `base` can be any expression, but assume that the `exponent` is a non-negative integer. You can simplify the cases when `exponent` is `0` or `1`, or when `base` is a number, by returning numbers from the constructor `make-exp`. In other cases, you can represent the exp as a triple `(^ base exponent)`.

> You may want to use the built-in procedure `expt`, which takes two number arguments and raises the first to the power of the second.

```
; Exponentiations are represented as lists that start with ^.
(define (make-exp base exponent)
  'YOUR-CODE-HERE
)

(define (exp? exp)
  'YOUR-CODE-HERE
)

(define x^2 (make-exp 'x 2))
(define x^3 (make-exp 'x 3))
```

Before you start, check your understanding by running

```
python ok --local -q make-exp -u
```

To test your code, if you are in the local Scheme editor, hit `Test`. You can click on a case, press `Run`, and then use the `Debug` and `Environments` features to figure out why your code is not functioning correctly.

You can also test your code from the terminal by running

```
python ok --local -q make-exp
```

Problem 4: Derive Exp (200 pts)

Implement `derive-exp`, which uses the power rule to derive exponents. Reduce the power of the exponent by one, and multiply the entire expression by the original exponent.

> **Note:** the formula for the derivative of an exponent is `[f(x)^(g(x))]' = g(x) * (f(x)^(g(x) - 1))`, if we ignore the chain rule, which we do for this problem

```
(define (derive-exp exp var)
  'YOUR-CODE-HERE
)
```

Before you start, check your understanding by running

```
python ok --local -q derive-exp -u
```

To test your code, if you are in the local Scheme editor, hit `Test`. You can click on a case, press `Run`, and then use the `Debug` and `Environments` features to figure out why your code is not functioning correctly.

You can also test your code from the terminal by running

```
python ok --local -q derive-exp
```

Extensions

There are many ways to extend this symbolic differentiation system. For example, you could simplify nested exponentiation expression such as `(^ (^ x 3) 2)`, products of exponents such as `(* (^ x 2) (^ x 3))`, and sums of products such as `(+ (* 2 x) (* 3 x))`. You could apply the chain rule when deriving exponents, so that expressions like `(derive '(^ (^ x y) 3) 'x)` are handled correctly. Enjoy!

After all, remember to submit your answers by

```
python submit.py --stuid [YOUR STUDENT ID] --stuname [YOUR NAME]
```