Instructions

> Please download homework materials `hw08.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 2. The starter code for these problems is provided in `hw08.scm`.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- Section 3.2

Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website.

Remember, you can use `ok` to test your code:

```
$ python ok  # test all functions
$ python ok -q <function>  # test single function
```

Problem 1: Pow (100pts)

Implement a procedure `pow` for raising the number `base` to the power of a *non-negative* integer `exp` for which the number of operations grows logarithmically, rather than linearly (the number of recursive calls should be much smaller than the input `exp`). For example, for `(pow 2 32)` should take 5 recursive calls rather than 32 recursive calls. Similarly, `(pow 2 64)` should take 6 recursive calls.

Hint: Consider the following observations:

1. $$x^{2y} = (x^y)^2$$
2. $$x^{2y+1} = x(x^y)^2$$

For example we see that 2^{32} is $(2^{16})^2$, 2^{16} is $(2^8)^2$, etc. You may use the built-in predicates `even?` and `odd?`. Scheme doesn't support iteration in the same manner as Python, so consider another way to solve this problem.

```
(define (square x) (* x x))

(define (pow base exp)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (pow 2 3)
; 8
```

Problem 2: Filter Lst (100 pts)

Write a procedure `filter-lst`, which takes a predicate `fn` and a list `lst`, and returns a new list containing only elements of the list that satisfy the predicate. The output should contain the elements in the same order that they appeared in the original list.

```
(define (filter-lst fn lst)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (filter-lst even? '(0 1 1 2 3 5 8))
; (0 2 8)
```

Problem 3: No Repeats (100 pts)

Implement `no-repeats`, which takes a list of numbers `s` as input and returns a list that has all of the unique elements of `s` in the order that they first appear, but no repeats. For example, `(no-repeats (list 5 4 5 4 2 2))` evaluates to `(5 4 2)`.

Hints: To test if two numbers are equal, use the `=` procedure. To test if two numbers are not equal, use the `not` procedure in combination with `=`. You may find it helpful to use the `filter-lst` procedure.

```
(define (no-repeats s)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (no-repeats (list 5 4 5 4 2 2))
; (5 4 2)
```

Problem 4: Substitute (100 pts)

Write a procedure `substitute` that takes three arguments: a list `s`, an `old` word, and a `new` word. It returns a list with the elements of `s`, but with every occurrence of `old` replaced by `new`, even within sub-lists.

Hint1: The built-in `pair?` predicate returns True if its argument is a `cons` pair.

Hint2: The `=` operator will only let you compare numbers, but using `equal?` or `eq?` will let you compare symbols as well as numbers. For more information, check out the Scheme Built-in Procedure Reference.

```
(define (substitute s old new)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (substitute '(a (a b c) d) 'b 'e)
; (a (a e c) d)
```

Problem 5: Sub All (100 pts)

Write `sub-all`, which takes a list `s`, a list of `old` words, and a list of `new` words; the last two lists must be the same length. It returns a list with the elements of s, but with each word that occurs in the second argument replaced by the corresponding word of the third argument. You may use `substitute` in your solution. Assume that `olds` and `news` have no elements in common.

```
(define (sub-all s olds news)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (sub-all '(a (a b c) d) '(b d) '(e f))
; (a (a e c) f)
```

Problem 6: Tree in Scheme (100 pts)

Let's design a tree data abstraction in scheme! As tree in python, a tree has a label and branches -- a list of trees.

Hint: Recall the function-based tree ADT in python.

```
(define (tree label branches)
  'YOUR-CODE-HERE
)

(define (label t)
  'YOUR-CODE-HERE
)

(define (branches t)
  'YOUR-CODE-HERE
)

(define (is-leaf t)
  'YOUR-CODE-HERE
)

; A tree for test

(define t1 (tree 1
  (list
    (tree 2
      (list
        (tree 5 nil)
          (tree 6 (list
            (tree 8 nil)))))
    (tree 3 nil)
    (tree 4
      (list
        (tree 7 nil))))))

;;; Tests
; scm> (label t1)
; 1
; scm> (label (car (branches t1)))
; 2
; scm> (is-leaf (car (cdr (branches t1))))
; #t
; scm> (branches (car (cdr (branches t1))))
; ()
```

Test your implementation with ok:

```
$ python ok -q tree
```

Problem 7: Label Sum (100 pts)

Using the data abstraction above, write a function that sums up the entries of a tree, assuming that the entries are all numbers.

Hint: You may want to use `map` with a helper function to sum list entries.

Note: Do not violate the abstraction barrier!

```
(define (label-sum t)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (label-sum t1)
; 36
```

Just for fun Problems

This section is out of scope for our course, so the problems below is optional. That is, the problems in this section **don't** count for your final score and **don't** have any deadline. Do it at any time if you want an extra challenge!

To check the correctness of your answer, you can submit your code to ok.

Problem 8: Derive (optional, 0 pts)

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:

```scheme
; 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 8.1: Derive Sum (optional, 0 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.

Problem 8.2: Derive Product (optional, 0 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
)
```

Problem 8.3: Make Exp (optional, 0 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 procedure `pow` implemented in problem 1 or 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))
```

Problem 8.4: Derive Exp (optional, 0 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
)
```

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!