Instructions

Please download homework materials `hw09.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 `hw09.scm`.

Note: You are supposed to finish Problem 4, 5, 6 after Wednesday's lecture about `Macros`.

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

Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here when you get stuck.

Tail Recursion

Recall from lecture that Scheme supports tail-call optimization. The example we used was factorial (shown in both Python and Scheme):

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

Notice that in this version of `factorial`, the return expressions both use recursive calls, and then use the values for more "work." In other words, the current frame needs to sit around, waiting for the recursive call to return with a value. Then the current frame can use that value to calculate the final answer.

As an example, consider a call to `fact(5)` (Shown with Scheme below). We make a new frame for the call, and in carrying out the body of the function, we hit the recursive case, where we want to multiply 5 by the return value of the call to `fact(4)`. Then we want to return this product as the answer to `fact(5)`. However, before calculating this product, we must wait for the call to `fact(4)`. The current frame stays while it waits. This is true for every successive recursive call, so by calling `fact(5)`, at one point we will have the frame of `fact(5)` as well as the frames of `fact(4)`, `fact(3)`, `fact(2)`, and `fact(1)`, all waiting for `fact(0)`.

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

Keeping all these frames around wastes a lot of space, so our goal is to come up with an implementation of factorial that uses a constant amount of space. We notice that in Python we can do this with a while loop:

```
def fact_while(n):
    total = 1
    while n > 0:
        total = total * n
        n = n - 1
    return total
```

However, Scheme doesn't have `for` and `while` constructs. No problem! Everything that can be written with while and `for` loops and also be written recursively. Instead of a variable, we introduce a new parameter to keep track of the total.

```
def fact(n):
    def fact_optimized(n, total):
        if n == 0:
            return total
        return fact_optimized(n - 1, total * n)
    return fact_optimized(n, 1)
```

```
(define (fact n)
    (define (fact-optimized n total)
        (if (= n 0)
            total
            (fact-optimized (- n 1) (* total n))))
    (fact-optimized n 1))
```

Why is this better? Consider calling `fact(n)` on based on this definition:

```
(fact 5)
(fact-optimized 5   1)
(fact-optimized 4   5)
(fact-optimized 3  20)
(fact-optimized 2  60)
(fact-optimized 1 120)
(fact-optimized 0 120)
120
```

Because Scheme supports tail-call optimization (note that Python **does not**), it knows when it no longer needs to keep around frames because there is no further calculation to do. Looking at the last line in `fact_optimized`, we notice that it returns the same thing that the recursive call does, no more work required. Scheme realizes that there is no reason to keep around a frame that has no work left to do, so it just has the return of the recursive call return directly to whatever called the current frame.

Therefore the last line in `fact_optimized` is a **tail-call**.

Macro

So far we've been able to define our own procedures in Scheme using the `define` special form. When we call these procedures, we have to follow the rules for evaluating call expressions, which involve evaluating all the operands.

We know that special form expressions do not follow the evaluation rules of call expressions. Instead, each special form has its own rules of evaluation, which may include not evaluating all the operands. Wouldn't it be cool if we could define our own special forms where we decide which operands are evaluated? Consider the following example where we attempt to write a function that evaluates a given expression twice:

```
scm> (define (twice f) (begin f f))
twice
scm> (twice (print 'woof))
woof
```

Since `twice` is a regular procedure, a call to `twice` will follow the same rules of evaluation as regular call expressions; first we evaluate the operator and then we evaluate the operands. That means that `woof` was printed when we evaluated the operand `(print 'woof)`. Inside the body of `twice`, the name `f` is bound to the value `undefined`, so the expression `(begin f f)` does nothing at all!

The problem here is clear: we need to prevent the given expression from evaluating until we're inside the body of the procedure. This is where the `define-macro` special form, which has identical syntax to the regular `define` form, comes in:

```
scm> (define-macro (twice f) (list 'begin f f))
twice
```

`define-macro` allows us to define what's known as a `macro`, which is simply a way for us to combine unevaluated input expressions together into another expression. When we call macros, the operands are not evaluated, but rather are treated as Scheme data. This means that any operands that are call expressions or special form expression are treated like lists.

If we call `(twice (print 'woof))`, `f` will actually be bound to the list `(print 'woof)` instead of the value `undefined`. Inside the body of `define-macro`, we can insert these expressions into a larger Scheme expression. In our case, we would want a `begin` expression that looks like the following:

```
(begin (print 'woof) (print 'woof))
```

As Scheme data, this expression is really just a list containing three elements: `begin` and `(print 'woof)` twice, which is exactly what `(list 'begin f f)` returns. Now, when we call `twice`, this list is evaluated as an expression and `(print 'woof)` is evaluated twice.

```
scm> (twice (print 'woof))
woof
woof
```

To recap, macros are called similarly to regular procedures, but the rules for evaluating them are different. We evaluated lambda procedures in the following way:

- Evaluate operator
- Evaluate operands

- Apply operator to operands, evaluating the body of the procedure

However, the rules for evaluating calls to macro procedures are:

- Evaluate operator
- Apply operator to unevaluated operands
- Evaluate the expression returned by the macro in the frame it was called in.

What Would Scheme Display

One thing to keep in mind when doing this question, builtins get rendered as such:

```
scm> +
#[+]
scm> list
#[list]
```

If evaluating an expression causes an error, type `SchemeError`. If nothing is displayed, type `Nothing`.

Use Ok to test your knowledge with the following "What Would Scheme Display?" questions:

```
python ok -q wwsd -u
```

```
scm> +
------
scm> list
------
scm> (define-macro (f x) (car x))
------
scm> (f (2 3 4)) ; type SchemeError for error, or Nothing for nothing
------
scm> (f (+ 2 3))
------
scm> (define x 2000)
------
scm> (f (x y z))
------
scm> (f (list 2 3 4))
------
scm> (f (quote (2 3 4)))
------
scm> (define quote 7000)
------
scm> (f (quote (2 3 4)))
------
scm> (define-macro (g x) (+ x 2))
------
scm> (g 2)
------
scm> (g (+ 2 3))
------
scm> (define-macro (h x) (list '+ x 2))
------
scm> (h (+ 2 3))
------
scm> (define-macro (if-else-5 condition consequent) `(if ,condition ,consequent 5))
------
scm> (if-else-5 #t 2)
------
scm> (if-else-5 #f 3)
------
scm> (if-else-5 #t (/ 1 0))
------
scm> (if-else-5 #f (/ 1 0))
------
scm> (if-else-5 (= 1 1) 2)
------
scm> '(1 x 3)
------
scm> (define x 2)
------
scm> `(1 x 3)
------
scm> `(1 ,x 3)
------
scm> '(1 ,x 3)
------
scm> `(,1 x 3)
------
scm> `,(+ 1 x 3)
------
scm> `(1 (,x) 3)
------
scm> `(1 ,(+ x 2) 3)
------
scm> (define y 3)
------
scm> `(x ,(* y x) y)
------
```

```
scm> `(1 ,(cons x (list y 4)) 5)
------
```

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: Count Change III (100 pts)

Write a procedure `make-change`, which takes in positive integers `total` and `biggest` and outputs a list of lists, in which each inner list contains positive numbers no larger than `biggest` that sum to `total`.

Note: Both outer list and inner lists should be descending ordered.

Hint: You may find Scheme built-in procedure `append` useful.

```
(define (make-change total biggest)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (make-change 2 2)
; ((2) (1 1))
; scm> (make-change 3 3)
; ((3) (2 1) (1 1 1))
; scm> (make-change 4 3)
; ((3 1) (2 2) (2 1 1) (1 1 1 1))
```

Problem 2: Find It! (100 pts)

Implement the Scheme procedure `find`, which takes a number `n` and a list of numbers `lst`. It returns the position (i.e. index) where `n` appears in `lst`. Assume that `n` appears exactly once in `lst`.

Note: Your solution should be tail-recursive.

```
(define (find n lst)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (find 1 '(1 2 3))
; 0
; scm> (find 2 '(1 2 3))
; 1
```

Problem 3: Find It! II (100 pts)

Implement the Scheme procedure `find-nest`, which takes a number `n` and a symbol `sym` that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to `n` by repeatedly applying `car` and `cdr` to the nested list. Assume that `n` appears exactly once in the nested list bound to `sym`.

```
(define (find-nest n sym)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (define a '(1 (2 3) ((4))))
; a
; scm> (find-nest 1 'a)
; (car a)
; scm> (find-nest 2 'a)
; (car (car (cdr a)))
```

Problem 4: Scheme Def (100 pts)

Note: You are supposed to finish this problem after Wednesday's lecture about `Macros`.

Implement `def`, which simulates a python `def` statement, allowing you to write code like `(def f(x y) (+ x y))`.

The above expression should create a function with parameters `x` and `y`, and body `(+ x y)`, then bind it to the name `f` in the current frame.

Note:

1. the previous is equivalent to `(def f (x y) (+ x y))`.
2. `body` stands for a single expression. Expressions like `(def f (x y) (print x) (print y)` is outside the scope of this problem, you don't need to care about it.

Hint: We strongly suggest doing the WWSD questions on macros first as understanding the rules of macro evaluation is key in writing macros.

```
(define-macro (def func args body)
  ''YOUR-CODE-HERE
)

;;; Tests
; scm> (def f(x y) (+ x y))
; f
; scm> (f 1 2)
; 3
```

Problem 5: K-Curry (100 pts)

Note: You are supposed to finish this problem after Wednesday's lecture about `Macros`.

Write the macro `k-curry`, which takes in a function `fn`, a list of the function's arguments `args`, a list of `k` values to substitute in for these arguments, and a sorted list of `k` non-negative `indices` specifying which arguments to pass values from `vals` into. `k-curry` will return a new function which takes in whatever arguments are remaining from `fn`'s list of arguments, with the arguments picked out by `indices` automatically filled in with the values from `vals`.

Hint: Write helper functions outside the macro, and use them inside the macro.

```
(define-macro (k-curry fn args vals indices)
  ''YOUR-CODE-HERE
)

;;; Tests
; scm> (define (f a b c d) (- (+ a c) (+ b d)))
; f
; scm> (define minus-six (k-curry f (a b c d) (2 4) (1 3)))
; minus-six
; scm> (minus-six 8 10) ; (- (+ 8 10) (+ 2 4))
; 12
```

Problem 6: Let *! (100 pts)

Note: You are supposed to finish this problem after Wednesday's lecture about `Macros`.

Let's meet a new special form `let*`, which has a form similar to `let`:

```
(let*
  ((name1 expr1)
   (name2 expr2)
   ...
   (nameN exprN))
  expr)
```

What `let*` does is similar to `let`, and the only difference is that the bindings are performed sequentially (one by one), i.e., `expr1` is evaluated first and `name1` is immediately bound to it, then `expr2` and `name2`, and so on. In contrast, with `let`, `expr1` through `exprN` are firstly evaluated without binding `name1`, ..., `nameN` to them; after that, the names are bound to the corresponding values.

Note:

1. `expr` stands for a single expression. Expressions like `(let* ((x 1) (y (+ x 1))) (print x) y)` is outside the scope of this problem, you don't need to care about it.
2. `expr` should always be evaled in the local frame.

Hint1: Use recursion, it is available in macro.

Hint2: You can use `let` to implement `let*`.

```
(define-macro (let* bindings expr)
  ''YOUR-CODE-HERE
)

;;; Tests
; scm> (define y 0)
; y
; scm> (let* ((x 1) (y (+ x 1))) y)
; 2
; scm> y
; 0
```

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 7: Find It! III (optional, 0pts)

Implement the `find-in-tree` procedure, which takes a Scheme tree `t`, a target value `goal` and returns a list containing all paths from the root of `t` to all node containing `goal`. If no such path exists, `find-in-tree` returns an empty list. The return list should be sorted by the end of each path, according to preorder traversal order.

```
(define (find-in-tree t goal)
  'YOUR-CODE-HERE
)

;;; Tests
; A tree for test
(define t1 (tree 1
  (list
    (tree 2
      (list
        (tree 3 nil)
        (tree 7 (list
          (tree 7 nil)))))
    (tree 3 nil)
    (tree 6
      (list
        (tree 7 nil))))))

; scm> (find-in-tree t1 0)
; ()
; scm> (find-in-tree t1 2)
; ((1 2))
; scm> (find-in-tree t1 3)
; ((1 2 3) (1 3))
; scm> (find-in-tree t1 7)
; ((1 2 7) (1 2 7 7) (1 6 7))
```

Problem 8: Infix (optional, 0pts)

One annoying thing about Scheme is that it can only understand arithmetic operations that are written in prefix notation. That is, if I want to evaluate an expression, the arithmetic operator must come first, which is different than in math class.

Let's leverage our skills to define a Scheme macro `infix` that accepts arithmetic operations with infix notation, which places operators between operands as you are used to. You only need to support the addition and multiplication operators `*` and `+`.

There are 2 ways to solve this problem -- calculate the value of the expression directly, or just adjust the order of operators and operands.

There are 4 test cases in our *Online Judge*, their requirements are detailed in the table below.

Test Case	Extra Restrictions on `expr`
8.1	Atomic operands are all numbers, `expr` is either an atomic operand or a three-element list `(expr op expr)`.
8.2	`expr` is either an atomic operand or a three-element list `(expr op expr)`.
8.3	Atomic operands are all numbers.
8.4	None.

If you choose to solve only 8.1 and 8.2, the solution can be *very simple* (less than 120 characters), give a try!

```
(define-macro (infix expr)
  ''YOUR-CODE-HERE
)

;;; Tests
; scm> (define x 5)
; x
; scm> (infix ((1 + 2) * (3 + 4)))
; 21
; scm> (infix ((1 + 2) * (3 + x)))
; 24
; scm> (infix (1 + 2 * 3 + 4))
; 11
; scm> (infix (1 + 2 * 3 + x))
; 12
```