# Lab 12: Programs as Data lab12.zip (lab12.zip)

*Due by 11:59pm on Wednesday, November 15.*

## Starter Files

Download lab12.zip (lab12.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

## Required Questions

Getting Started Videos

## Quasiquotation

Consult the drop-down if you need a refresher on Quasiquotation. It's okay to skip directly to the questions and refer back here should you get stuck.

Quasiquotation

### Q1: WWSD: Quasiquote

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

```
python3 ok -q wwsd-quasiquote -u
```

```
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 2) 3)

scm> (define y 3)

scm> `(x ,(* y x) y)

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

## Programs as Data

Consult the drop-down if you need a refresher on Programs as Data. It's okay to skip directly to the questions and refer back here should you get stuck.

Programs as Data

### Q2: If Program

In Scheme, the `if` special form allows us to evaluate one of two expressions based on a predicate. Write a program `if-program` that takes in the following parameters:

1. `predicate` : a quoted expression which will evaluate to the condition in our `if` - expression
2. `if-true` : a quoted expression which will evaluate to the value we want to return if `predicate` evaluates to true ( `#t` )
3. `if-false` : a quoted expression which will evaluate to the value we want to return if `predicate` evaluates to false ( `#f` )

The program returns a Scheme list that represents an `if` expression in the form: `(if <predicate> <if-true> <if-false>)`. Evaluating this expression returns the result of our `if` expression.

Here are some doctests to show this:

```
scm> (if-program '(= 0 0) '2 '3)
(if (= 0 0) 2 3)
scm> (eval (if-program '(= 0 0) '2 '3))
2
scm> (if-program '(= 1 0) '(print 3) '(print 5))
(if (= 1 0) (print 3) (print 5))
scm> (eval (if-program '(= 1 0) '(print 3) '(print 5)))
5
```

```
(define (if-program condition if-true if-false)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q if-program
```

## Q3: Exponential Powers

Implement the procedure `pow-expr`, which takes in a base `n` and a nonnegative integer `p`. The procedure should create a program as a list that, when passed into the `eval` procedure, evaluates the value `n^p`, or `n` to the `p`th power.

For example, the expression `(pow-expr 2 5)` returns a program as a list. When `eval` is called on that returned list, it should evaluate to `2^5`, which is `32`. We'll do this by building nested multiplication expressions!

```
scm> (define expr (pow-expr 5 1))
expr
scm> expr
(* 1 5)
scm> (eval expr)
5

scm> (define expr2 (pow-expr 5 2))
expr2
scm> expr2
(* (* 1 5) 5)
scm> (eval expr2)
25
```

**Hint:** Note that the expression returned by `(pow-expr 5 2)` is just the expression returned by `(pow-expr 5 1)` nested in another multiplication expression. There is an inherent *recursive* structure here, so what programming paradigm do you think we should use?

```
(define (pow-expr n p)
    'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q pow
```

## Q4: Swap

Implement `swap`, a procedure which takes a call expression `expr` and returns the same expression with its first two operands swapped if the **evaluated** value of the second operand is greater than the **evaluated** value of the first operand. Otherwise, it should just return the original expression.

For example, `(swap '(- 1 (+ 3 5) 7))` should return the expression `(- (+ 3 5) 1 7)` since `1` evaluates to `1`, `(+ 3 5)` evaluates to `8`, and `8 > 1`. Any operands after the first two should not be evaluated during the execution of the procedure, and they should be left unchanged in the final expression. You may assume that every operand evaluates to a number and that there are always at least two operands in `expr`.

**Hint:** Quasiquotation might not be the best way to approach this problem (look at the bindings in the `let` expression to see why this might be the case). What other methods of building lists could we use to solve this problem?

**Reminder:** Don't forget to **evaluate** the first two operands when comparing them!

```
(define (cddr s)
  (cdr (cdr s))
)

(define (cadr s)
  (car (cdr s))
)

(define (caddr s)
  (car (cddr s))
)

(define (swap expr)
    (let ((op (car expr))
          (first (car (cdr expr)))
          (second (caddr expr))
          (rest (cdr (cddr expr))))
          'YOUR-CODE-HERE
    )
)
```

Use Ok to test your code:

```
python3 ok -q swap
```