Detailed explanation for Week 12 Worksheet

Last updated Fall 2019 by Jemmy Zhou

apply-twice

Official Solution:

```
scm> (define-macro (apply-twice call-expr)
...     `(let ((operator ,(car call-expr))
...              operand ,(car (cdr call-expr)))
...            (operator (operator operand))))
...
scm> (define add-one (lambda (x) (+ x 1)))
scm> (apply-twice (add-one 1))
3
```

How it works:

1. We pass in the entire expression `(add-one 1)` as `call-expr`.
2. In the body of the apply-twice procedure, we build a `let` expression.
3. We eval `(car call-expr)` and `(car (cdr call-expr))`.
    1. `(car call-expr)` evaluates to the string `add-one`.
    2. `(car (cdr call-expr))` evaluates to the number `1`.
4. The body of the macro procedure thus evaluates to

    ```
    (let ((operator add-one)
          (operand 1))
         (operator (operator operand)))
    ```

5. Evaluating this `let` expression returns the value 3.
    1. When we evaluate the binding `(operator add-one)`, the procedure `add-one` is bound to `operator`.
    2. When we evaluate the binding `(operand 1)`, the number `1` is bound to `operand`.
    3. When we evaluate `(operator operand)`, `operator` evalutes to the procedure `add-one` while `operand` evalutes to 1.

Are these correct? Why?

1. Student Solution #1

    ```
    (define-macro (apply-twice call-expr)
    `(let ((operator (car ,call-expr))
            operand (car (cdr ,call-expr)))
          (operator (operator operand))))
    ```

2. Student Solution #2

    ```
    (define-macro (apply-twice call-expr)
    `(let ((operator ,(car call-expr))
            operand (car (cdr ,call-expr)))
          (operator (operator operand))))
    ```

3. Student Solution #3

```
(define-macro (apply-twice call-expr)
`(let ((operator ,(car call-expr))
        operand (car ,(cdr call-expr)))
      (operator (operator operand))))
```