Lab 12: Macros **lab12.zip (lab12.zip)**

Due by 11:59pm on Wednesday, November 16.

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.

Topics

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

Quasiquotation

Quasiquotation

Recall that the `quote` special form prevents the Scheme interpreter from executing a following expression. We saw that this helps us create complex lists more easily than repeatedly calling `cons` or trying to get the structure right with `list`. It seems like this form would come in handy if we are trying to construct complex Scheme expressions with many nested lists.

Consider that we rewrite the `twice` macro as follows:

```
(define-macro (twice f)
  '(begin f f))
```

This seems like it would have the same effect, but since the `quote` form prevents any evaluation, the resulting expression we create would actually be `(begin f f)`, which is not what we want.

The **quasiquote** allows us to construct literal lists in a similar way as quote, but also lets us specify if any sub-expression within the list should be evaluated.

At first glance, the quasiquote (which can be invoked with the backtick `` ` `` or the `quasiquote` special form) behaves exactly the same as `'` or `quote`. However, using quasiquotes gives you the ability to **unquote** (which can be invoked with the comma `,` or the `unquote` special form). This removes an expression from the quoted context, evaluates it, and places it back in.

By combining quasiquotes and unquoting, we can often save ourselves a lot of trouble when building macro expressions.

Here is how we could use quasiquoting to rewrite our previous example:

```
(define-macro (twice f)
  `(begin ,f ,f))
```

> Important Note: quasiquoting isn't necessarily related to macros, in fact it can be used in any situation where you want to build lists non-recursively and you know the structure ahead of time. For example, instead of writing `(list x y z)` you can write `` `(,x ,y ,z) `` for 100% equivalent behavior

Macros

Macros

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

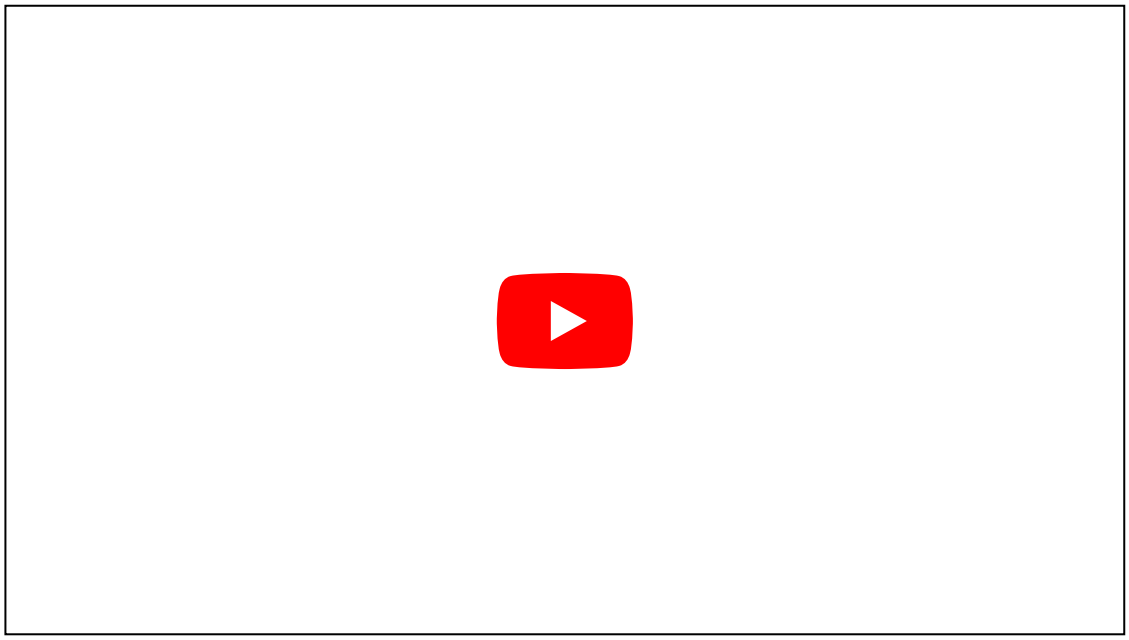
However, the rules for evaluating calls to macro procedures are:

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

Walkthrough video

We've provided a walkthrough video going over the contents of this week's lab.

To see these videos, you should be logged into your berkeley.edu email.



YouTube link (https://youtu.be/playlist?list=PLx38hZJ5RLZelFBIqH5_R2Cpzrx9lDCLS)

Required Questions

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 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)
------
```

Macros

Q2: If Macro Scheme

Use the `define-macro` special form to define a macro, `if-macro` ,which will take in the following parameters:

1. `condition` : an expression which will evaluate to the condition in our if expression

2. `if-true` : an expression which will evaluate to the value we want to return if true
3. `if-false` : an expression which will evaluate to the value we want to return if false

and evaluates the result of the if expression passed in.

```
scm> (if-macro (= 0 0) 2 3)
2
scm> (if-macro (= 1 0) (print 3) (print 5))
5
```

> If you're having a hard time with the question revisit Question 3 (If Program Scheme) from Discussion 11 (/disc/disc11/#q3). In the past discussion, we had to `'` our arguments, and we had to call `eval` on the return value of `if-function` to actually run the line of code we produced. Because we are using the `define-macro` special form, the operands are no longer quoted, and we no longer have the perform an extra call to `eval` !

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

Use Ok to test your code:

```
python3 ok -q if-macro
```

Q3: Or Macro

Implement `or-macro` , which takes in two expressions and `or` 's them together (applying short-circuiting rules). However, do this without using the `or` special form. You may also assume the name `v1` doesn't appear anywhere outside this macro. For a quick reminder on the short-circuiting rules for `or` take a look at slide 18 of Lecture 3 on Control (/assets/slides/03-Control_4pp.pdf).

The behavior of the `or` macro procedure is specified by the following doctests:

```
scm> (or-macro (print 'bork) (/ 1 0))
bork
scm> (or-macro (= 1 0) (+ 1 2))
3
```

```
(define-macro (or-macro expr1 expr2)
    `(let ((v1 ____________))
       (if _____ _____ _____))
)
```

Use Ok to test your code:

```
python3 ok -q or-macro
```

Q4: Replicate

Implement the `replicate` procedure, which takes in a value `x` and a number `n` and returns a list with `x` repeated `n` times. We've provided some examples for how `replicate` should function here:

```
scm> (replicate '(+ 1 2) 3)
((+ 1 2) (+ 1 2) (+ 1 2))

scm> (replicate (+ 1 2) 1)
(3)

scm> (replicate 'hi 0)
()
```

Then, write a macro that takes an expression `expr` and a number `n` and repeats the expression `n` times. For example, `(repeat-n expr 2)` should behave the same as `(twice expr)` from the introduction section of this worksheet. It's possible to pass in a combination as the second argument (e.g. `(+ 1 2)`) as long as it evaluates to a number. Be sure that you evaluate this expression in your macro so that you don't treat it as a list.

Complete the implementation for `repeat-n` so that its behavior matches the doctests below.

```
scm> (repeat-n (print '(resistance is futile)) 3)
(resistance is futile)
(resistance is futile)
(resistance is futile)

scm> (repeat-n (print (+ 3 3)) (+ 1 1))  ; Pass a call expression in as n
6
6
```

> **Hint**: Consider which method to build your list (`list`, `cons`, or `quasiquote`) might help make your implementation simpler.

```
(define (replicate x n)
    'YOUR-CODE-HERE
)

(define-macro (repeat-n expr n)
    'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q repeat-macro
```

Q5: List Comprehensions

Recall that list comprehensions in Python allow us to create lists out of iterables:

```
[<map-expression> for <name> in <iterable> if <conditional-expression>]
```

Define a procedure to implement list comprehensions in Scheme that can create lists out of lists. Specifically, we want a `list-of` program that can be called as follows:

```
(list-of <map-expression> 'for <name> 'in <list> 'if <conditional-expression>)
```

Note that the symbols for, in, and if must be quoted when calling list-of so that they will not be evaluated and cause an error.

Calling `list-of` will return a new list constructed by doing the following for each element in `<list>`:

- Bind `<name>` to the element.
- If `<conditional-expression>` evaluates to a truth-y value, evaluate `<map-expression>` and add it to the result list.

Here are some examples:

```
scm> (list-of '(* x x) 'for 'x 'in ''(3 4 5) 'if '(odd? x))
(map (lambda (x) (* x x)) (filter (lambda (x) (odd? x)) (quote (3 4 5))))

scm> (eval (list-of '(* x x) 'for 'x 'in ''(3 4 5) 'if '(odd? x)))
(9 25)

scm> (list-of ''hi 'for 'x 'in ''(1 2 3 4 5 6) 'if '(= (modulo x 3) 0))
(map (lambda (x) (quote hi)) (filter (lambda (x) (= (modulo x 3) 0)) (quote (1 2 3 4 5

scm> (eval (list-of ''hi 'for 'x 'in ''(1 2 3 4 5 6) 'if '(= (modulo x 3) 0)))
(hi hi)

scm> (eval (list-of '(car e) 'for 'e 'in ''((10) 11 (12) 13 (14 15)) 'if '(list? e)))
(10 12 14)
```

Hint: You may use the built-in `map` and `filter` procedures. Check out the Scheme Built-ins (/articles/scheme-builtins/) reference for more information.

You may also find it helpful to look at the expression returned by list-of in the example above.

```
(define (list-of map-expr for var in lst if filter-expr)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q list-comp-func
```

Q6: List Comprehension Macro

When we wrote out our original `list-of` program in Question 5, we had to do additional work to actually run the outputted program. Particularly, we needed to quote parameters and perform an extra call to `eval` to get the desired answer. To make things easier, we can use the `define-macro` special form to simplify the process.

As a reminder, the `define-macro` form changes the order of evaluation to be:

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

Use the `define-macro` special form to implement the `list-of-macro` macro that can be called as follows:

```
(list-of-macro <map-expression> for <name> in <list> if <conditional-expression>)
```

Calling `list-of-macro` will return a new list constructed by doing the following for each element in `<list>`:

- Bind `<name>` to the element.
- If `<conditional-expression>` evaluates to a truth-y value, evaluate `<map-expression>` and add it to the result list.

Here are some examples:

```
scm> (list-of-macro (* x x) for x in '(3 4 5) if (odd? x))
(9 25)

scm> (list-of-macro 'hi for x in '(1 2 3 4 5 6) if (= (modulo x 3) 0))
(hi hi)

scm> (list-of-macro (car e) for e in '((10) 11 (12) 13 (14 15)) if (list? e))
(10 12 14)
```

> *Hint:* Again, you may use the built-in `map` and `filter` procedures. Check out the Scheme Built-ins (/articles/scheme-builtins/) reference for more information.

```
(define-macro (list-of-macro map-expr for var in lst if filter-expr)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q list-comp
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```