Macro Review - Scheme

Macros

Previously, we've seen how we can create macro-like functions in Python using f-strings. Now let's talk about real macros, in Scheme. 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.

In the scheme project, we saw 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!

We have a problem here: 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:

1. Evaluate operator

2. Evaluate operands
3. Apply operator to operands, evaluating the body of the procedure

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

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

are swapped from 'normal' form!

Q5: Shapeshifting Macros

guided walkthrough

Q5: Shapeshifting Macros (Tutorial)

When writing macros in Scheme, we often create a list of symbols that evaluates to a desired Scheme expression. In this question, we'll practice different methods of creating such Scheme lists.

We have executed the following code to define `x` and `y` in our current environment.

```
(define x '(+ 1 1))
(define y '(+ 2 3))
```

x & y as variables have been bound as lists

We want to use `x` and `y` to build a list that represents the following expression:

```
(begin (+ 1 1) (+ 2 3))
```

remember begin just goes through and evaluates everything left to right & returns the last evaluated term

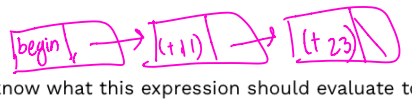
(1) What would be the result of calling `eval` on a quoted version of the expression above?

```
(eval '(begin (+ 1 1) (+ 2 3)))
```

so result should be whatever this evaluates to

Your Answer:

5



(2) Now that we know what this expression should evaluate to, let's build our scheme list.

How would we construct the scheme list for the expression `(begin (+ 1 1) (+ 2 3))` using quasiquotation?

make a list w/ these vals using quasiquoting

x y

Your Answer:

`(begin ,x ,y)

(3) How would we construct this scheme list using the `list` special form?

(list _ _ _ _)

we want list to look same as above but use list instead of quote to make list

we want to evaluate x & y so that we get their actual vals & not just the letters x and y

Your Answer:

(list 'begin x y)

(4) How would we construct this scheme list using the `cons` special form?

same as above but now with cons!

Remember, with cons if you want a single element list, you need to pass in nil as the second arg!

(cons 1 nil) → 1

(cons 1) → error!

Your Answer:

(cons 'begin (cons x (cons y nil)))

(cons (+ 1 1) nil)

2

SOLUTION:

(1) 5

(2) `(begin ,x ,y)

(3) (list 'begin x y)

(4) (cons 'begin (cons x (cons y nil)))

Q6: Max Macro

Q6: Max Macro (Tutorial)

Define the macro `max`, which takes in two expressions `expr1` and `expr2` and returns the maximum of their values. If they have the same value, return the first expression.

```
scm> (max 5 10)
10
scm> (max 12 12)
12
scm> (max 100 99)
100
```

Link in chat! please write your solution/ideas there!!

WORK TIME: 10 mins

1. First, try using quasiquotation to implement this macro procedure.

Your Answer

```
1   (define-macro (max expr1 expr2)
2       'YOUR-CODE-HERE
3       )
4
```

Hint: think about what expression you would want to call eval on so that the max of expr1 & expr2 is returned! (maybe think about how you could use an if statement to accomplish this)

4:42

2. Now, try writing this macro using the `list` special form.

Your Answer

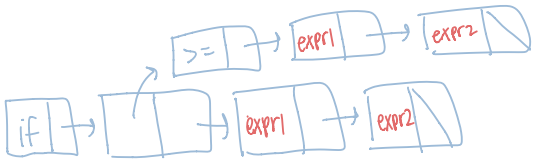
```
1   (define-macro (max expr1 expr2)
2       'YOUR-CODE-HERE
3       )
4
```

if expr1 >= expr2:
    expr1
else:
    expr2

3. Finally, write this macro using the `cons` special form.

Your Answer

```
1   (define-macro (max expr1 expr2)
2       'YOUR-CODE-HERE
3       )
4
```



SOLUTION:

①
```
(define-macro (max expr1 expr2)
  `(if (>= ,expr1 ,expr2) ,expr1 ,expr2)
)
```

remember the comma acts like an unquote & makes sure that our list replaces the word expr1 with the actual parameter that's been passed in

②
```
(define-macro (max expr1 expr2)
  (list 'if (list '>= expr1 expr2) expr1 expr2)
)
```

we don't want to evaluate the 'if' special form so quoting it ensures that we add the word if to our list

③
```
(define-macro (max expr1 expr2)
  (cons 'if (cons (cons '>= (cons expr1 (cons expr2 nil))) (cons expr1 (cons expr2 nil))))
)
```

main takeaway is cons makes writing macros very convoluted & hard to read!

Q7: When Macro

Q7: When Macro (Tutorial)

Using macros, let's make a new special form, `when`, that has the following structure:

```
(when <condition>
      (<expr1> <expr2> <expr3> ...))
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire `when` expression evaluates to `okay`.

→ if condition is true

→ if condition is false

```
scm> (when (= 1 0) ((/ 1 0) 'error))
okay
scm> (when (= 1 1) ((print 6) (print 1) 'a))
6
1
a
```

↓ condition = false

↓ condition = true

WORK TIME: 6 min.

4:55

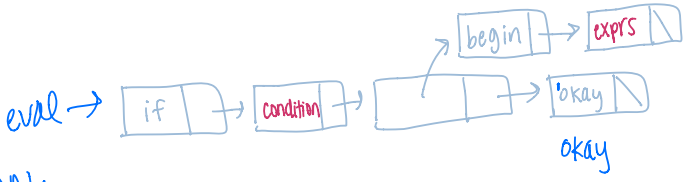
Hint: start by drawing out what the list should look like

Your Answer

```
1   (define-macro (when condition exprs)
2     'YOUR-CODE-HERE
3     )
4
```

(begin _ _ _)

Link in chat! ↳ please write your solution/ideas there!!



SOLUTION:

```
(define-macro (when condition exprs)
  '(if ,condition ,(cons 'begin exprs) 'okay)
)
```

quote makes it so that begin is not evaluated (we want the word begin itself)

we don't want the word condition, we want the condition parameter that was passed in

this makes it so that everything inside the blue cons is evaluated

scheme doesn't have any identifier (any way to recognize the word okay) so we have to quote it so that when eval is called on the resulting list scheme doesn't try to evaluate the word okay (the quasiquote just makes it so that we don't try to evaluate it before putting it in the list)