**Macro Procedures**

*Objectives: Understand what it means to treat an expression as data, the differences compared to regular Scheme procedures, and strategies for writing macro procedures.*

*Many credits to Chris Allsman and Jen Thakar*

**List Construction**

To get started, we’ll review a few concepts with lists in Scheme. These aren’t all the ways to create lists in Scheme, but they’ll be referenced later in this guide.

**Using List**

When drawing a box and pointer diagram for a list expression, the first step is to count the number of elements being passed in. This will tell us how many pairs will be included in our diagram. We’ll use an example to demonstrate this: (list 4 5 6)

We’ve passed in three elements to list, so we’ll start off by drawing three connected pairs with nil at the end:

At this point, we can fill in the specified values in the three open spots:

4

5

6

What if our list was a bit more complex? (list 8 (list 3 4))

Since there are two elements being passed in, we’ll draw two connected pairs:

There were two elements passed into list: 8 and another list. As 8 was the first element, we place it in the first spot. We use a pointer to the list containing 3 and 4 in the second spot to represent the nested list as our second element.

8

3

4

**Using Quotes**

We can also create lists with single quotes! Remember that single quotes return everything to the right of the quote, unevaluated. For example:

scm> ‘(1 2 3)

(1 2 3)

Creates the following list:

1

2

3

**Using Quasiquotes and Unquotes**

Quasiquotes (backtick symbol `) work very similarly to quotes. The key difference is that they give us the ability to use unquotes (comma symbol ,). When we place a quasiquote outside an expression, we still get a list of all the elements, though any elements that are unquoted are evaluated. For example:

scm> (define x 6)

x

scm> (define y 1)

y

scm> `(,x ,y a)

(6 1 a)

6

1

a

**Input and Output**

Using the regular Scheme procedures we’ve seen so far, we could always say that (foo 15) was equivalent to (foo (\* 3 5)). Why is this? Well, to evaluate any call expression to a regular Scheme procedure, we have the following set of rules:

1. Evaluate operator to a regular procedure
2. Evaluate operand expressions
3. Apply the procedure to evaluated operand(s)

For the first call expression, to evaluate it we would evaluate the foo function, evaluate 15 to the value 15, and then apply foo to 15. For the second expression, we would evaluate the foo function, evaluate (\* 3 5) to the value 15, and then apply foo to 15. The key takeaway here is that for regular Scheme procedures, our operands are always evaluated to values. Since during the apply step the body of the procedure is evaluated as well, the output of our procedure will always be a value as well. We can visualize this with the following diagram:

**Input**

**Output**

**Procedures**

Expressions

Values

Values

Calculations

eval

The body of our procedure can do whatever we like--most of the time it’s some sort of calculation--though what’s important is that our procedures take values as input, do some work, and return values as output.

Up until now, we’ve treated values as data. We could manipulate those values, use them to calculate things, or whatever we like. Expressions are automatically evaluated in Scheme, so in order to work with with the expressions themselves (instead of what they evaluate to) we need to halt this evaluation and store the expression itself as data.

Since everything in Scheme besides primitives are lists, we can store those expressions as lists. Meaning, the expression (+ 1 2) is stored like the following:

+

1

2

Now we’ll take a look at macro procedures! Instead of the rules of evaluating call expression for regular procedures as seen above, we’ll use the following rules:

1. Evaluate operator to a macro procedure
2. Apply macro procedure to the operand expressions
3. Evaluate the expression returned by the macro procedure **in the frame it was called in**

We can visualize this with the following diagram:

**Input**

**Output**

**Macro Procedures**

Expressions

Construct a list of data

Expressions

Values

eval

Macro procedures take expressions as input, similar to the regular procedures, although the key difference is they are not evaluated before running the body of the procedure. When we write macro procedures, instead of thinking “what should the behavior of my procedure be?” think “how do I construct a list that represents the behavior of my procedure?”.

**What we can do with macros**

The only reason the special forms in Scheme are special is because they all have unique evaluation rules. For example, if we tried to call ‘if’ as a procedure, we’d end up evaluating if as the operator, we’d evaluate the [if-true] and [if-false] expressions as the operands, and then apply the function to the operands. Doing this causes us to lose the invariants that define if statements because we only want at most one of the clauses to be evaluated, never both of them at the same time.

With macros, we can take advantage of the fact that none of the operands get evaluated when it’s called and create our own unique evaluation rules. We can control which operand expressions are evaluated and when, which means we can even design our own special forms! We’ll walk through a few examples on how to do exactly this in the next section.

**Approaching Macro Procedures**

Before we get started, here’s a rough outline on the steps to writing a macro procedure:

1. Take the data that you’re given (could be lists, or other expressions) and draw out the box and pointer diagram to represent it if applicable.
2. Write the Scheme code that accomplishes what the procedure should do
3. Use the body of the macro procedure to construct that code in a list. It represents the code that’ll eventually be evaluated.

**Examples**

The rest of this guide will be walking through example code writing questions.

**make-and**

Let’s take a look at make-and, which is a macro procedure that behaves like our and special form and accepts two arguments, returning either the first false-y value or the last truth-y value.

scm> (make-and 4 5)

5

scm> (make-and #f (/ 1 0))

#f

1. Input? We’re given two arguments, arg1 and arg2.
2. Behavior? We want it to work like our and special form, so let’s write it that way (note that there are multiple solutions to this problem).

(and arg1 arg2)

1. Create the list! The and expression from step 2 contains three elements (the ‘and symbol, arg1, and arg2). So we can create a list containing just that:

(list ‘and arg1 arg2)

We also could have used quasiquotes and unquotes to represent the same list:

`(and ,arg1 ,arg2)

Whichever implementation you choose can be directly returned from the macro procedure, as it’ll be evaluated implicity to behave just like the regular Scheme code we wrote at the beginning. Our final solution will look something like the following:

|  |  |
| --- | --- |
| (define-macro (make-and arg1 arg2)  (list ‘and arg1 arg2)  ) | (define-macro (make-and arg1 arg2)  `(and ,arg1 ,arg2)  ) |

**apply-twice**

We already have the and special form in Scheme, so this example wasn’t as exciting. Let’s try out apply-twice, which takes in a call expression call-expr and returns the result of calling the operator twice on the operand. For example:

scm> (define (add-one x) (+ x 1))

add-one

scm> (apply-twice (add-one 3))

5 ;; The result of calling (add-one (add-one 3))

1. Input? A Scheme list called call-expr containing an operator and a single operand. Since it’s a list, we’ll draw it out:

add-one

1

1. Behavior? We want to return the result of calling (add-one (add-one 3))
2. Create the list! The expression we want has two elements: the first element being the operator of call-expr and the second being another list, containing the operator and the operand of call-expr.

add-one

add-one

1

How can we construct it? Well we can get the operator by asking for the (car call-expr), and for the second element, one way would be to create a new list containing the (car call-expr) and the (car (cdr call-expr)):

(list (car call-expr) (list (car call-expr) (car (cdr call-expr))))

There’s a small modification we can make here. Notice how the second element is the same as the input list we were given, so we can simply create a list containing our operator and call-expr.

(list (car call-expr) call-expr)

Our resulting macro procedure then looks like this:

(define (apply-twice call-expr)

(list (car call-expr) call-expr)

)

Why did apply-twice have to be a macro procedure instead of a regular one? Well, remember that our regular procedures will always evaluate their operands before executing the body of the function. So if apply-twice were a regular procedure and we made the following call:

(apply-twice (add-one 3))

We would enter the body of the function with call-expr bound to the value 4. This isn’t useful to us, because we have no way of knowing what the operator or original operand was, so we don’t have the information we need to apply procedure twice. That’s why we’re using a macro, because we can take the data we were given and manipulate it to create the expression we want.

**cond-false**

Let’s take a look at an example that may seem more difficult that the previous ones, though can still be broken down into the three steps to writing macro procedures as we’ve seen before. We’d like to implement cond-false, which works similarly to the built-in special form cond, though instead will evaluate only the second sub-expression of the first falsey condition (whereas cond evaluates the second sub-expression to the first truthy condition). Our input will also be a **list of clauses** instead of a variable number of clauses, to make the problem a bit easier. Feel free to define any helper functions or use any built-ins necessary. Here are a few examples of how cond-false should function:

scm> (cond-false ( ((= 1 2) (\* 3 4))

((= 3 3) (\* 8 9)) )

)

12

scm> (cond-false ( ((= 5 5) (print 5))

((even? 7) (print 7)) )

)

7

scm> (cond-false ( (#f ‘sun)

((/ 1 0) ‘moon) )

)

sun

To get started, remember the preliminary steps we discussed earlier in this guide. First identify your input and draw out the box and pointer(s) if necessary, ensure that you understand the behavior of the macro (write out the equivalent scheme code for one of the doctests, or your own), and finally construct the scheme list that represents that behavior. Try it out yourself first, and we’ll walk through it together.

Let’s first start out with those three preliminary steps:

1. Input? A list of clauses, where each clause has two elements: a condition and an if-true expression. If the condition is false, we want to evaluate the if-true expression.
2. Behavior? Our macro should behave similarly to a cond expression if we just instead modify all the conditional expressions to be instead (eq? #f [condition]). For example, the first doctest should produce the same result as the following:

scm> (cond ((eq? #f (= 1 2)) (\* 3 4))

((eq? #f (= 3 3)) (\* 8 9))

)

12

This way, the expression will evaluate the sub-expression of the first clause since (eq? #f (= 1 2)) evaluates to false.

1. Now we’d like to construct the list to produce this modified cond expression. This can get a little bit tricky, since we would need to split up clauses in different parts and recombine them into a single expression. Below, bolded are the expressions within clauses, and unbolded is elements that we’ve added:

(cond ((eq? #f **(= 1 2)**) **(\* 3 4)**)

((eq? #f **(= 3 3)**) **(\* 8 9)**)

)

The first thing that we can notice is that each clause in our cond expression contains two elements, the first being (eq? #f [condition]) and the second being the corresponding [if-true] expression. Let’s try to solve a smaller problem, that is simply constructing a list of all of the (eq? #f [condition]) expressions for each condition in clauses.

Note that all we want to do is create a new list, so there’s no need to write this helper procedure as a macro, it can just be a regular scheme list construction question.

Input: ((= 1 2) (= 3 3))

Output: ((eq? #f (= 1 2)) (eq? #f (= 3 3)))

(define (convert-false c)

(if (null? lst) nil

(cons (list 'eq? #f (car c)) (convert-false (cdr c)))

)

)

Alternatively, if instead of a list of conditions we assumed we passed in the entire clauses list, we could write construct-clause to return the same clauses list but with the modified conditions.

Input: ( ((= 1 2) (\* 3 4)) ((= 3 3) (\* 8 9)) )

Output: ( ( (eq? #f (= 1 2)) (\* 3 4) )

( (eq? #f (= 3 3)) (\* 8 9) ) )

(define (construct-clause clauses)

(if (null? clauses) nil

(cons

(cons

(list 'eq? #f (car (car clauses)))

(cdr (car clauses))

)

(construct-clause (cdr clauses))

)

)

)

In both cases we had the code segment (list 'eq? #f [condition]) which constructs a list that therefore stops evaluation of the condition. Alternatively, it would seem to be equivalent if we had bypassed the list-creation and directly coded the expression (eq? #f [condition]), therefore evaluating it in the frame of the helper function instead of in the frame of the macro procedure. This difference becomes most apparent for the following doctest we saw earlier:

scm> (cond-false ( (#f ‘sun)

((/ 1 0) ‘moon) )

)

If we had instead evaluated both of the conditions in this cond-false expression within the frame of the helper function, we’d lose the sense of short-circuiting. We’d end up evaluating (/ 1 0) and throwing a zero-division error when instead we should only be evaluating the conditions up to and including the first falsey one we find, which in this case only ends up being the first condition, #f.

From here, we’ve essentially computed all that we need to construct the cond expression we’d like. One thing to notice is that every cond expression is a scheme list where the first element is ‘cond and the subsequent elements are the clauses. By using a cons statement, we can create a well-formed list where the first element is ‘cond and the second element will be our desired clauses.

(define-macro (cond-false clauses)

(cons ‘cond … )

)

For example, given this list we will get the following and an abbreviated box and pointer diagram is included below:

scm> (cons ‘cond ‘(( (eq? #f (= 1 2)) (\* 3 4) )

( (eq? #f (= 3 3)) (\* 8 9)) ) )

(cond ((eq? #f (= 1 2)) (\* 3 4))

((eq? #f (= 3 3)) (\* 8 9)))

cond

first clause

second clause

The two possible helper procedures we saw earlier each have different approaches to constructing our desired list of clauses. We’ll first walk through the process for convert-false (took a list of conditions as input) and then we’ll look at how we could’ve used the convert-clause.

Our helper procedure convert-false was solely taking care of constructing the first element (the (eq? #f [condition]) part) of each of the conditions in our clauses, we now have a list containing all the first elements. To get a list containing all the second elements? (i.e., the [if-true] parts)? Since for each clause the [if-true] portion is the car of the cdr of the clause, we can get a list of all the [if-true] portions by utilizing map:

(map cadr clauses)

Remember that when we defined convert-false, we assumed that the input would be a list of all of our conditions. We can apply a similar trick as we did before using map to accomplish this, and call:

(convert-false (map car clauses))

to get all of our properly converted conditions. By zipping this list with our [if-true] expressions, we’ve now constructed our final clauses expression to be included in the cond expression:

(define-macro (cond-false clauses)

(cons

'cond

(zip (convert-false (map car clauses)) (map cadr clauses))

)

)

Now if we wanted to instead use convert-clause, since it essentially took care of all the work to create the list of updated clauses, we can simply define our macro as the following:

(define-macro (cond-false clauses)

(cons 'cond (convert-clause clauses))

)

And there we have it! Now we have two implementations of a fully functional cond-false special form! A few key takeaways from this problem:

1. It was very important that we understood what scheme code we wanted to construct. This was what lead us to see that our problem reduced to figuring out a way to modify each condition to be (eq? #f [condition]).
2. We took advantage of our pre-existing special forms (cond) to make our problem a bit easier, as we reduced the problem of creating the cond-false special form to calling cond with modified conditions.
3. We had to ensure that our desired evaluation rules were followed, meaning we had to make sure that only the conditions up to and including the first falsey condition were evaluated as well as only the sub-expression corresponding to said first falsey condition.
4. Our macro returned **a list** to be evaluated in the place where said macro was called. This is why it’s helpful to draw out the scheme code in box and pointer notation before writing the body of the macro.