Guide to Macros (Scheme)

Formatted by: Bryan Tong, CS61A Spring 2018
Credit to: Chris Allsman, Sequoia Eyzaguirre

Introduction

Don't know what a macro is? Don't worry - this is the first semester macros have been officially taught, so they're a bit daunting to actually find resources on too.

Thus, this will be a complete walkthrough of macros from scratch, although you should already be familiar with Scheme and its specifics. If you feel shaky on Scheme, functional programming, and quoting, you should refresh yourself [here](#).

This is a guide built off work done by CSM, Chris Allsman, and Sequoia Eyzaguirre. This guide is intended to be a complete guide to macros (relative to Scheme) from the student's perspective, though there is a video lecture Professor DeNero [gave here](#) as well as an introduction on the [discussion worksheet](#) that are worth checking out first.

We'll cover the same examples here (credit to Chris Allsman for all of these), but this will be more comprehensive than the discussion. Lecture should still not be skipped - definitely watch it in combination to this document!

Note: This guide is still a WIP, and is heavily influenced by the original Macro's guide for teaching. I plan to add more of Sequoia's notes into it soon and add some original examples.

Part 1

Expressions as Data

You've probably heard "Expressions as Data" from a TA as a way to motivate why we use Scheme, but it's likely not clear what that even means, so let's look at an example. As a reminder, on the semantics of terminology, expressions evaluate to values. So 1 + 2 is an expression (in Python), which evaluates to the value 3.

With that in mind, let's look at a list in Scheme: '(+ 1 2). Unsurprisingly, this evaluates to the list (+ 1 2), due to the existence of the quote.

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

However, we know by now that (+ 1 2) is an expression in Scheme! So what happens when we evaluate this expression, defined as "a"?

```
scm> (define a '(+ 1 2))
a
scm> (eval a)
3
```

We get the value we would expect when evaluating the original expression! This might be obvious, but what this means is that all expressions (other than atoms and names) in Scheme are really just lists.

So what? Well, this is actually incredibly powerful - this is a unique feature in Scheme versus what we've seen in Python.   Lists are things we can create ourselves, just like any other form of data. Hence, Scheme allows us to construct our own expressions within a procedure!

Why would we actually use this in reality though? Well, what if we want to define a procedure involving two arguments that returns an *expression* adding together its two arguments. Highlighting expression because key concept: returning (+ 1 2), not a simplified return value of 3, for this example problem.

We could do something like this:

Note that after evaluating the inside function, we're essentially calling (eval (+ 1 2)).

We can even make expressions for special forms, or nested expressions, if we wanted!
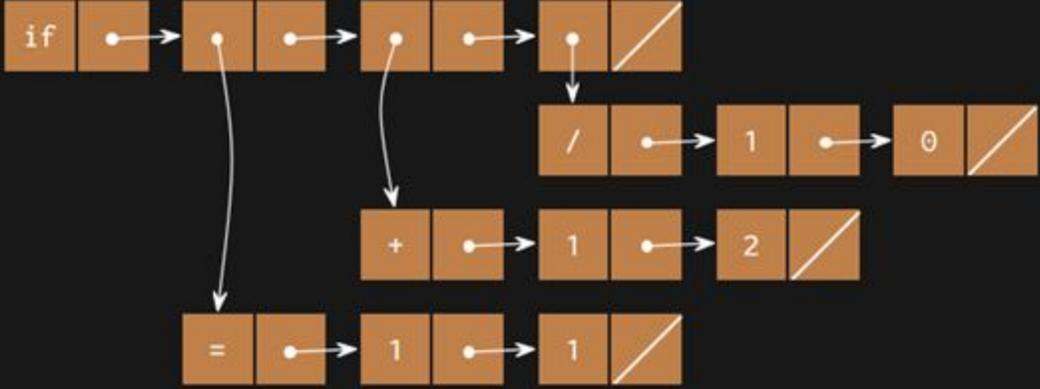
Part 2

Macros

Now we've seen how to write expressions inside a Scheme procedure by creating a list, and how we are able to return an expression instead of immediately evaluating two arguments. What do we actually do with this though?!

Well, consider this other example. Let's say we want to write a procedure called double, which will takes in an expression and evaluate it twice. Hypothetically, if we wrote:

```
(double (print 'bork))
```

We'd want the output to be:

```
bork
bork
```

(By the way, printing in Scheme works very similarly to printing in Python, except in Scheme we return an undefined value. Undefined values are not displayed in the interpreter, just like NoneType values are not displayed in Python.)

Anyway, this seems simple - right? Let's just evaluate whatever we pass in twice. Here's a first attempt:
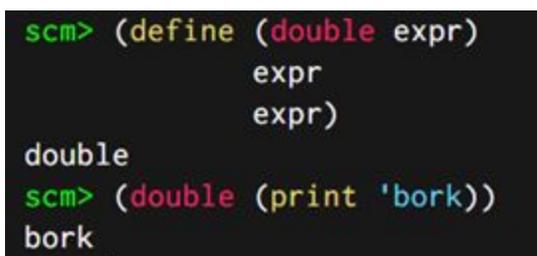
```
scm> (define (double expr)
              expr
              expr)
double
scm> (double (print 'bork))
bork
```

Well that doesn't work. We just print bork once -- why is this?!

Let's revisit the rules of Scheme evaluation: when calling double, we evaluate the call to print and print bork. But now expr is bound to an undefined value, so when we evaluate that twice, nothing actually happen in double! So now our goal is to try to prevent evaluation of expr before we apply double. Let's try quoting our input, then?

This won't work either - the issue being, after quoting the expression, we create a list. So when evaluating expr inside the function, we don't interpret it as an expression, we interpret it as data. But thankfully, because of the section before, we know that expressions and data are one and the same in Scheme! We just have to force Scheme to recognize this list as an expression and evaluate it. So let's redefine double and try one more time.

```
scm> (define (double expr)
             (eval expr)
             (eval expr))
double
scm> (double '(print 'bork))
bork
bork
```

Yay it worked! In fact, we can improve the composition a bit by using the begin special form. Let's create one giant expression and evaluate that rather than duplicating the same call to eval. The expression we want to create looks like (begin expr expr). So with that in mind, we get:

```
scm> (define (double expr)
             (eval (list 'begin expr expr)))
double
scm> (double '(print 'bork))
bork
bork
```

This may not look any better or clearer, but hold on for a minute. We defined the body of double with an `eval` call on this list starting with the quoted `'begin` because of the format of the list we want to call this `eval` on. Remember, the `expr` that we're taking in is also of a quoted form: `'print 'bork))`.
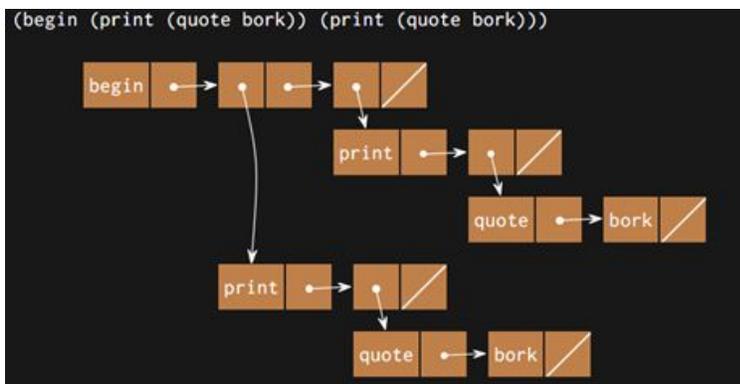
This is a box-and-pointer diagram showing what this list actually looks like. Notice that the structure of both `expr` sections is the same, but the *cdr* section does not point to the second expr, but rather another list with a null cdr.

Ok, so this kinda works. Though, there's still an issue, and the above diagram is a bit complex. Consider that we wanted the call to double to look like `(double (print 'bork))`; yet, to get this to work, we have to unintuitively write `(double '(print 'bork))`

How do we possibly fix this? By some surprise magic? Yes -- in the form of macros.

Part 3

What are Macros?

You might have in section, with some exasperation and hand-waving by a TA, that macros are basically "code that writes other code."

This isn't wrong! However, once diagrammed out, that actually essentially just look likes what we saw in the example above. That is, a macro constructs an expression (which again, is just a list) and evaluates it to get a specific value. There is a *major* thing to note here however, being that macros have two key properties that procedures do not:

1) Macros <u>do not evaluate their inputs</u>. You can think of it like a call to a macro will have all of its inputs automatically implicitly quoted. That's useful in this scenario - it means that we can make a call like we wanted above, since the call to a macro will automatically quote `(print 'bork)` and prevent its evaluation. Hence, we *actually pass in expressions* to macros, instead of the *values those expressions evaluate to.*

2) Macros <u>implicitly evaluate whatever they return</u>. In a macro, you can directly return an expression and the macro will evaluate that expression. If we weren't using macros, we can still call `eval` on whatever we returned, as we did above. It's just cool to note that this is rather convenient for these purposes, instead.

If there's a major takeaway, it's that the first point is the reason macros are so powerful and allows us to uniquely perform operations such as shown in the above examples.

Now, let's move on to some hands on macro implementation of the `double` function.

```
scm> (define-macro (double expr) (list 'begin expr expr))
double
scm>  (double (print 'bork))
bork
bork
```

It works - and we didn't have to quote our input! The code defining this macro, written with `define-macro` to specify this type, is also super clean. Now both our function is clean and our input feels natural - yay!

Part 4

Quasiquoting

Here's another nifty feature of Scheme that's worth learning about and is a newer concept to 61A.

Assume I wanted to return a function with no parameters via defining it through a macro. Thus, we can use this expression as the function's body alone. Let's look at a (surprisingly, perfectly valid) implementation of that:

```
scm> (define-macro (make-func expr)
                   (list 'lambda (list) expr))
make-func
scm> (make-func (/ 1 0))
(lambda () (/ 1 0))
```

It works, though it's a little frustrating to write the expression we're creating in the macro that way. Wouldn't it be easier to write something like `(lambda () expr)`, a more familiar syntax to us? However, then our issue is that by quoting the entire thing, we would not evaluate `expr`, so our return value would contain the *symbol* expr instead of the actual expression (e.g. `(/ 1 0)`)

```
scm> (define-macro (make-func expr)
                   '(lambda () expr))
make-func
scm> (make-func (/ 1 0))
(lambda () expr)
```

This is where *quasiquoting* comes into play - surprise! Instead of using a quote ('), we can use a quasiquote (`), aka the non-caps ~ wave-dash key, if it's unclear..

(con't on next page)

Normally, a quasiquote works just like a normal quote.

```scheme
scm> `(+ a b)
(+ a b)
```

However, a quasiquote also allows you to evaluate specific expressions by putting a comma in front of that expression. This is called *unquoting* the expression.

```scheme
scm> (define a 2)
a
scm> (define b 2)
b
scm> `(+ ,a ,b)
(+ 2 2)
```

So, we could define the make-func macro above as follows:

```scheme
scm> (define-macro (make-func expr)
                    `(lambda () ,expr))
make-func
scm> (make-func (+ 1 2))
(lambda () (+ 1 2))
```

Or, we could define the earlier double macro as:

```scheme
scm> (define-macro (double exp)
                    `(begin ,exp ,exp))
double
scm> (double (print 'bork))
bork
bork
```

Pretty nifty, right?

Part 5

Motivating Macros

You might have mastered macros now, yet still be wondering why we even need them. Isn't life simpler with a few less random concepts sprinkled into 61A?

Let's play devil's advocate for a moment. After all, if we wanted to print `bork` twice, we could have just written `(begin (print 'bork) (print 'bork))` in the first place. To even bother caring about macros and these minute concepts, it's very important to motivate why we care about macros. This is definitely a difficult concept, so let's go through some tangible examples to clearly illustrate the foundations behind macros.

In general, the power of macros comes from the fact that we do not evaluate the inputs to a macro when passing them in. This has a variety of implications for what we do with them - here's a few. *(The following examples are straight from Chris Allsman's original guide, hence the language explaining them might sound a bit different. s/o to Chris for these awesome examples)*

1) Rearranging expressions.

From lecture, we talked about how we might want to create some syntax similar to list comprehensions. That is, it would be nice to be able to write something like

```
(for (* x 2) x '(1 2 3))
```

Which would return

```
(2 4 6)
```

The issue, of course, is that the name `x` is not defined, and thus we would not be able to evaluate either `(* x 2)` or `x`. Even if we could, we do not want to evaluate these expressions on their own: rather, we want to use those to create a function that we map on to each of the values in the input list. So, we can write a macro like

```
scm> (define-macro (for expr var lst)
                   `(map (lambda (,var) ,expr) ,lst))
for
scm> (for (* x 2) x '(1 2 3))
(2 4 6)
```

Again, we could have just written `(map (lambda (x) (* x 2) '(1 2 3))` instead, but macros offer us a convenient way to make a new type of expression that we can rearrange into something that Scheme already supports. This is at its core is a type of sugaring, but the results can be rather sweet!

To re-emphasize, the only reason this is possible is because we don't evaluate the expressions when calling the macro. This allows us to rearrange those expressions when plugging them into an already-supported expression

2) Implementing special forms

Let's say you've just built a shiny new Scheme interpreter and... whoops, you forgot to implement the cond special form! Not to worry, because you did remember to implement macros.

Really, this type of application is similar to the problem we faced above. We'll create a new type of expression (a cond expression) and translate that into something our Scheme interpreter can recognize. But the conceptual difference is here, if we evaluated every sub-expression when we evaluate the cond expression, things would be even worse than usual. For special forms, you are not supposed to automatically evaluate each sub-expression. Rather, you only evaluate a subset of them depending on the specific special form you're working with. So if the cond expression acted like a normal call expression, we'd end up evaluating something we potentially don't want to.

Here's one potential implementation of cond as a macro (assuming we always have an else case and two other suites)

```
scm> (define-macro (cond-macro if-suite elif-suite else-suite)
                    (list 'if (car if-suite)
                              (car (cdr if-suite))
                              (list 'if (car elif-suite)
                                        (car (cdr elif-suite))
                                        (car (cdr else-suite)))))
cond-macro
scm> (cond-macro ((= 1 2) 1)
                 ((= 2 2) 2)
                 (else (/ 1 0)))
2
```

For another example, see the or macro on Discussion 9.

3) Lazy evaluation

What do we do in the case of lazy evaluation, where we don't want to evaluate something until it's actually forced, as with streams? We probably can't implement this with a procedure, because we don't want to evaluate elements of a stream as we create it. The problem is, again, we don't want to evaluate arguments at the point of making a call. Sounds like it's time for macros to shine again! Here's an implementation of `cons-stream` using macros (or well, con-stream, because you can't overwrite cons-stream since it's a special form)

```
scm> (define-macro (con-stream first rest)
                    `(cons ,first (delay ,rest)))
con-stream
scm> (define (a) 2)
a
scm> (define str (con-stream 1 (a)))
str
scm> (define (a) 1)
a
scm> (cdr-stream str)
1
```

4) Creating variable length expressions

In many of these cases it is possible, if rather difficult, to directly write out the expression a macro creates rather than having a macro create it for you. But what if we wanted to print out bork 100 times? That would be incredibly tedious to write if you were writing out the begin expression yourself! But what if we had a macro do it for us? You can see an example of that in the discussion worksheet with the first question! With some ingenuity, you could also do some fun stuff, like make your own special forms with any number of sub-expressions

Ultimately, maybe the best reason to use macros is the same reason it's good to use functions – abstraction! Sure, you can do this stuff without a macro, but you would have to repeat a lot of work and acquaint yourself with a lot of messy implementation details that you wouldn't have to worry about if you just used macros.

Thanks for taking the time to read this - good luck on the final!

Last updated: 4/26/2018