Introduction

So you are going over the content for the lectures you skipped (hey, you're going to watch the webcasts!) and you come across macros. You aren't quite sure what they are - you think it might be something about keyboard shortcuts? Or maybe those little cookies with the filling in the middle. Perhaps the president of France? But when you actually do watch the lecture, you realize how confusing they can be. But that might just be because you were watching at 2x speed.

Do not fear, because this guide is here to help you make sense of Scheme macros! This will be a pretty bottom up introduction (assuming you have NO idea what macros are), but it should be helpful even if you have some sense of what they are. This document is rather similar to the examples from lecture as well as on the discussion worksheet. We'll cover some of the same examples here, but this will be more comprehensive than the discussion. The explanation of this topic in lecture is really good, but if you want the "cliff notes" version this document should help. It also emphasizes some other aspects of macros which may have not been as apparent there.

And remember - even if macros are hard right now, you'll get them! We wouldn't put them in the course if learning them was not doable or valuable.

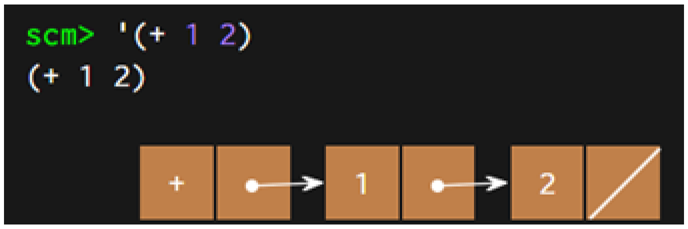
After that, there are also a couple of examples on where we might use macros. I think it's helpful to think about where macros are useful in order to understand their properties!

Part 1

Expressions as Data

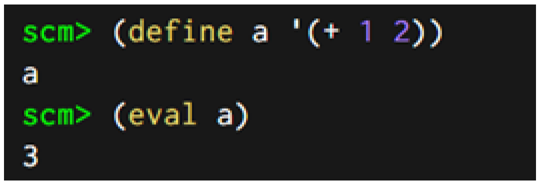
You may have heard “Expressions as Data” as a way to motivate why we use Scheme, but maybe you don’t know exactly what that means. Let’s look at an example. As a reminder, and so we’re clear with terminology, expressions evaluate to values. So 1 + 2 is an expression (in Python), which evaluates to the value 3.

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



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

But wait, (+ 1 2) is an expression in scheme! So what if we try to evaluate that expression?

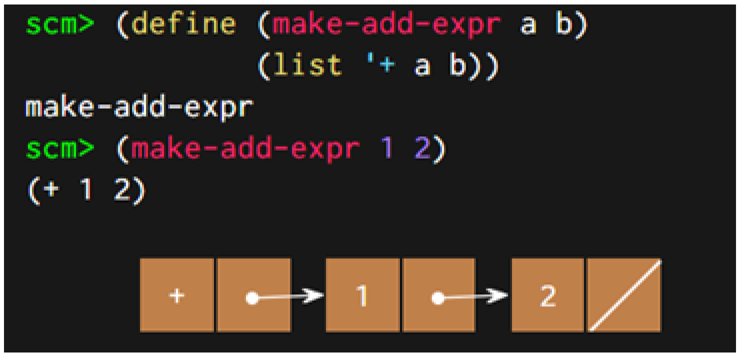


```
scm> (define a '(+ 1 2))
a
scm> (eval a)
3
```

We get the value we would expect when evaluating the original expression! So what this means is all expressions (other than atoms and names) in Scheme are really just lists. This is incredibly powerful! Lists are things we can create ourselves (they’re just data). So Scheme allows us to construct our own expressions within a procedure!

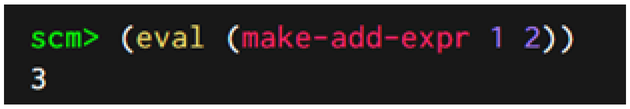
Taking a quick step back - why is this so cool? Well, in Python, even though we could write out expressions, we couldn’t actually return expressions in functions, or pass them into functions, or assign them to variables. We could only do those things with **values**. In Scheme, we can pass **expressions themselves** into functions, or return expressions, or assign expressions to variables, by quoting the expression. This allows us to do some really cool things that just aren’t possible in Python! And this is all due to the fact that Scheme is constructed in such a way that expressions are really just lists!

For the practical benefits of this: let's say we want to define a procedure that takes in two arguments and returns an *expression* adding together its two arguments. We could do something like this:



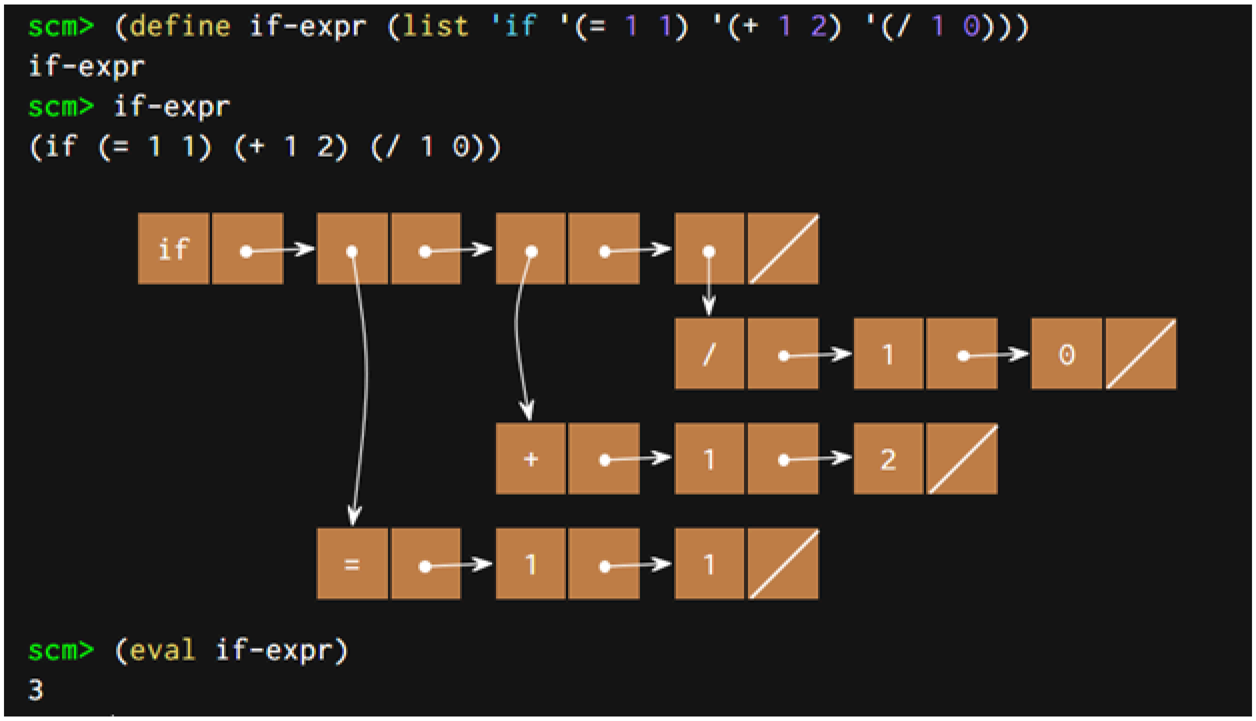
```
scm> (define (make-add-expr a b)
             (list '+ a b))
make-add-expr
scm> (make-add-expr 1 2)
(+ 1 2)
```

And we could use it as follows:



```
scm> (eval (make-add-expr 1 2))
3
```

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



```
scm> (define if-expr (list 'if '(= 1 1) '(+ 1 2) '(/ 1 0)))
if-expr
scm> if-expr
(if (= 1 1) (+ 1 2) (/ 1 0))
```

```
scm> (eval if-expr)
3
```

Macros

Ok great, so we can write our own expressions inside a procedure by creating a list! But why do we care about that? Well, let's take a look at another example (unrelated to the previous examples, for now). Let's say we want to write a procedure, double, which takes in an expression and evaluates it twice. So if we wrote

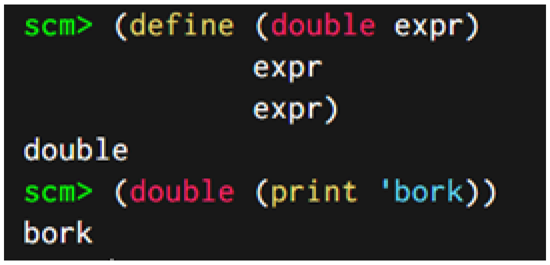
```
(double (print ‘bork))
```

We'd want the output to be

```
bork
bork
```

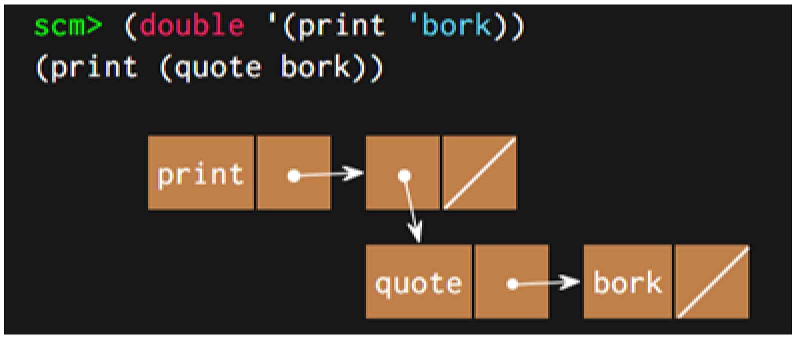
(By the way, printing in Scheme works very similarly to printing in Python, except in Scheme we return an undefined value. Undefined values are not displayed in the interpreter, just like NoneType values are not displayed in Python.)

Ok, seems simple enough, let's just evaluate whatever we pass in twice. Here's a first attempt:



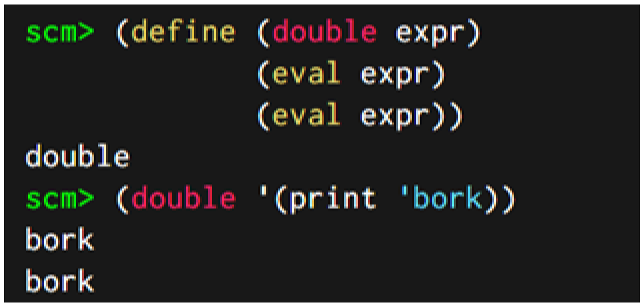
```
scm> (define (double expr)
                expr
                expr)
double
scm> (double (print 'bork))
bork
```

Well that doesn't work. We just print bork once! Once we remember the golden rules of evaluation, we realize the issue: when calling double, we evaluate the call to print (the operand) and print bork. But now `expr` is bound to an undefined value, so when we evaluate that twice, nothing happens! So now our goal is to try to prevent evaluation of `expr` before we apply double. Let's quote our input then!



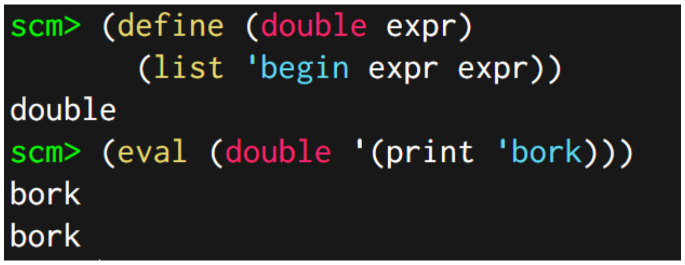
```
scm> (double '(print 'bork))
(print (quote bork))
```

Hmm, well this isn't helpful either. The issue is, 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.



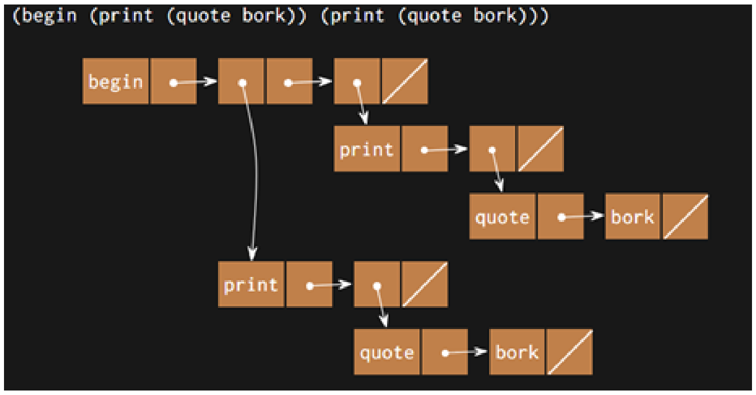
```
scm> (define (double expr)
               (eval expr)
               (eval expr))
double
scm> (double '(print 'bork))
bork
bork
```

Yay it worked! But there's one other way we can think about this. Instead of calling eval within the body of the function, what if I just have double return an expression that we can evaluate outside of the function - like we did with `make-add-expr` earlier. That might look something like this:



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

As an aside, the reason I write the body of double the way I did is because the list I ultimately want to call eval on looks like



```
(begin (print (quote bork)) (print (quote bork)))
```

Ok, so we got something working. But there's an issue: we wanted the call to double to look like `(double (print 'bork))`, but to get this to work, we have to write `(eval (double '(print 'bork)))`. How do we fix this? The answer is: MACROS!!!! (Wow it only took like 5 pages to get here)

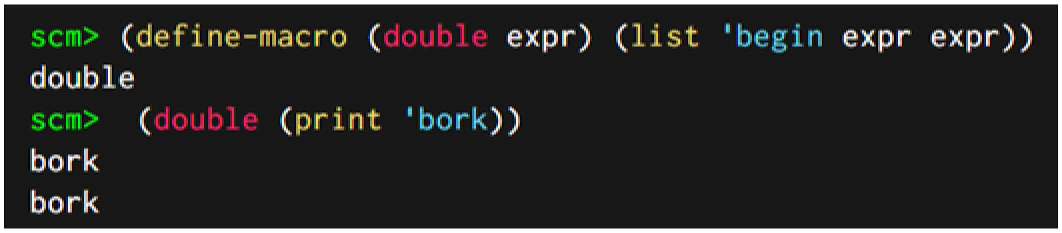
You might have heard macros described as "code that writes other code". Which is true! But what that actually looks like is pretty much what we just saw in the example above. That is, a macro constructs an expression (which again, is just a list) and evaluates it to get a specific value. But macros have two key properties that procedures do not:

1) Macros do not evaluate their inputs. You can think of it like a call to a macro will have all of its inputs implicitly quoted. That's great! That means we can make a call like we wanted above, since the call to a macro will automatically quote (print 'bork) and prevent its evaluation. So we truly pass in expressions to macros, instead of the values those expressions evaluate to.

2) Macros implicitly evaluate whatever they return. That means, in a macro, you can directly return an expression (and again, expressions are data, so you could just return a list here representing an expression!) and the macro will evaluate that expression. If we weren't in a macro we could of course call eval on whatever we returned, as we did above, but this is rather convenient, wouldn't you say?

Of these two features, the first one (not evaluating its inputs) is the reason macros are so powerful, and the reason it allows us to do stuff we otherwise couldn't.

Alright, with all that in mind, let's take a stab at implementing double using a macro.

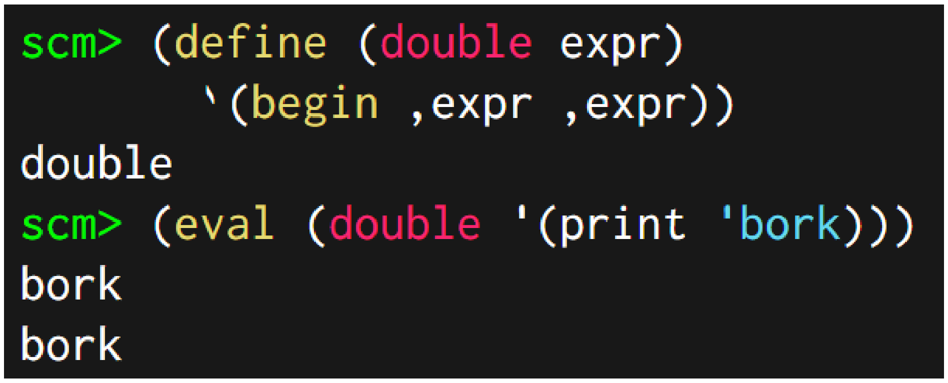


```
scm> (define-macro (double expr) (list 'begin expr expr))
double
scm>  (double (print 'bork))
bork
bork
```

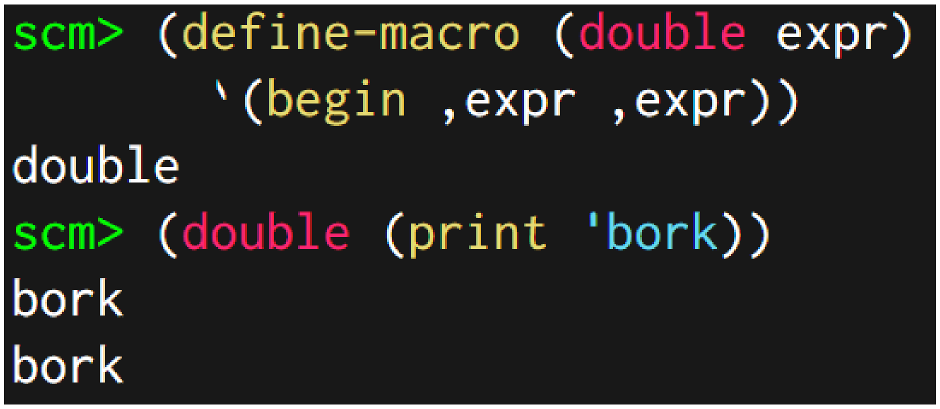
Yay it works! And we didn't have to quote our input! (As you can see, creating a macro works just like creating a function except we use define-macro instead of define).

Isn't Scheme cool???

To be very explicit about it, the following two blocks of code are functionally equivalent:



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

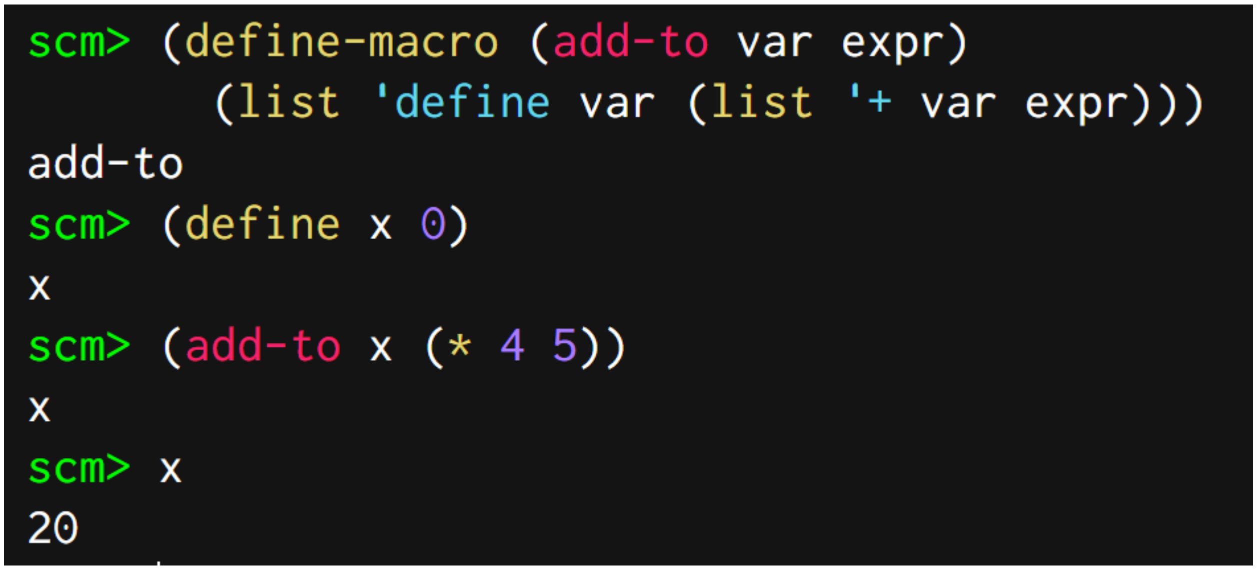


```
scm> (define-macro (double expr)
        `(begin ,expr ,expr))
double
scm> (double (print 'bork))
bork
bork
```

You can think through a macro as if it was a function which automatically quotes its inputs and automatically calls eval on its output

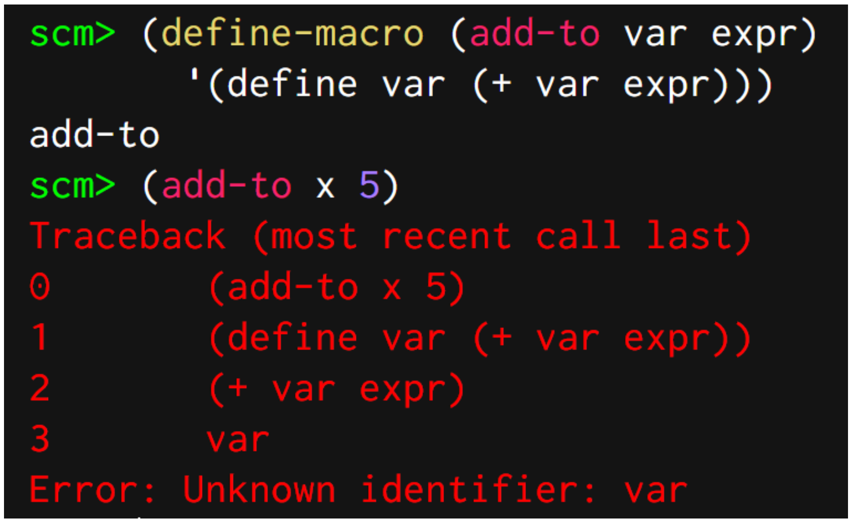
Quasiquoting

And now for a feature of Scheme that you'll wish you had earlier.
Let's say I wanted to make a macro that takes in a variable and an expression, and adds the value of that expression to that variable. Here's a (correct) implementation:



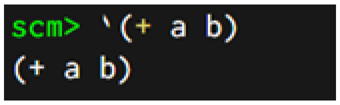
```
scm> (define-macro (add-to var expr)
       (list 'define var (list '+ var expr)))
add-to
scm> (define x 0)
x
scm> (add-to x (* 4 5))
x
scm> x
20
```

Yay it works! But it's a little cumbersome to write the expression we're creating in the macro that way…. It would be a lot easier to write something like `'(define var (+ var expr))`. The issue with that is by quoting the entire thing, we would not evaluate expr or var, so our return value would contain the *symbols* expr and var instead of the actual expression.



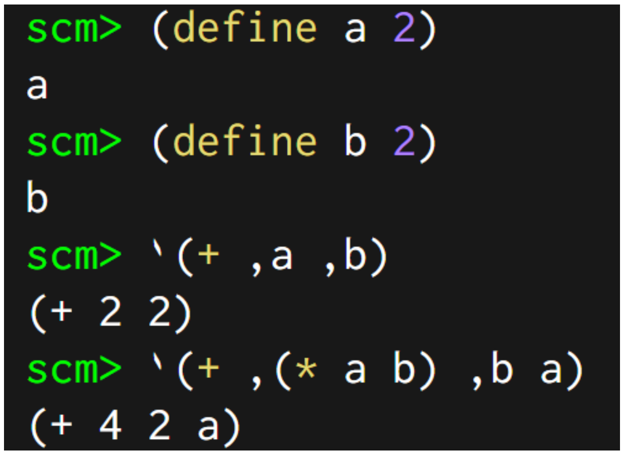
```
scm> (define-macro (add-to var expr)
       '(define var (+ var expr)))
add-to
scm> (add-to x 5)
Traceback (most recent call last)
0       (add-to x 5)
1       (define var (+ var expr))
2       (+ var expr)
3       var
Error: Unknown identifier: var
```

Here, the macro returns the expression `(define var (+ var expr))`. When trying to evaluate it, it attempts to evaluate `(+ var expr)` but var and expr aren't defined in the global frame!
If only there was a way to evaluate specific parts of an expression when we quote it. Well, Scheme's got your back! Instead of using a quote ('), we can use a quasiquote (`). Normally, a quasiquote works just like a normal quote.



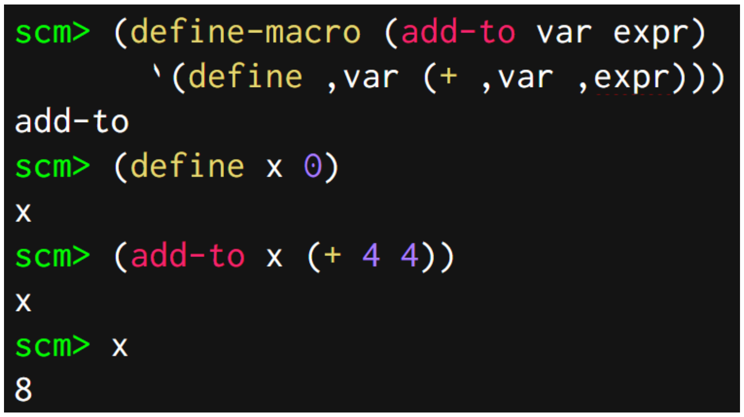
```
scm> `(+ a b)
(+ a b)
```

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



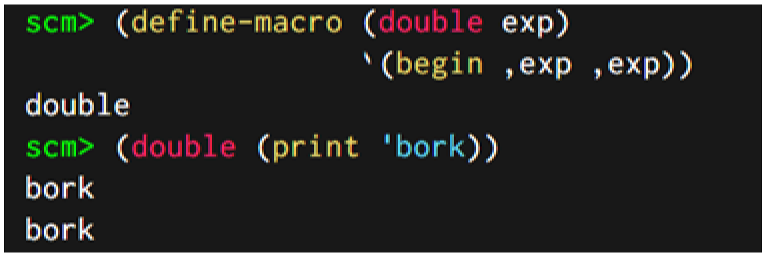
```
scm> (define a 2)
a
scm> (define b 2)
b
scm> `(+ ,a ,b)
(+ 2 2)
scm> `(+ ,(* a b) ,b a)
(+ 4 2 a)
```

So, we could define the add-to macro above as follows:



```
scm> (define-macro (add-to var expr)
       `(define ,var (+ ,var ,expr)))
add-to
scm> (define x 0)
x
scm> (add-to x (+ 4 4))
x
scm> x
8
```

Or, we could define the earlier double macro as



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

Wow!!!! Isn't Scheme so cool!!!!!!!!!!

Part 2

Uses for Macros

Ok, so by this point you're a master of macros! But you still might be asking yourself why we even need them. After all, if we wanted to print bork twice, we could have just written `(begin (print 'bork) (print 'bork))` in the first place.

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 are just a few.

1) Rearranging expressions.

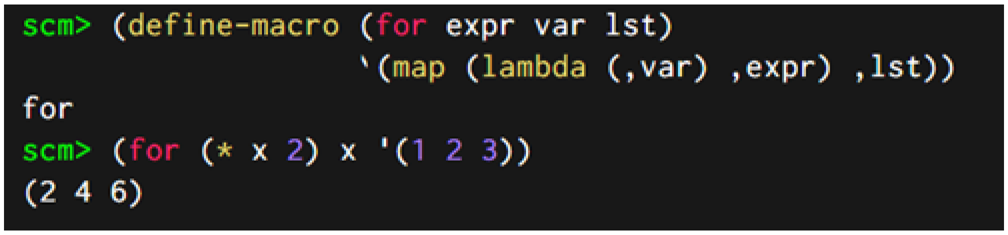
From lecture, we talked about how we might want to create some syntax similar to list comprehensions. That is, it would be nice to be able to write something like

```
(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 that final expression `(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.

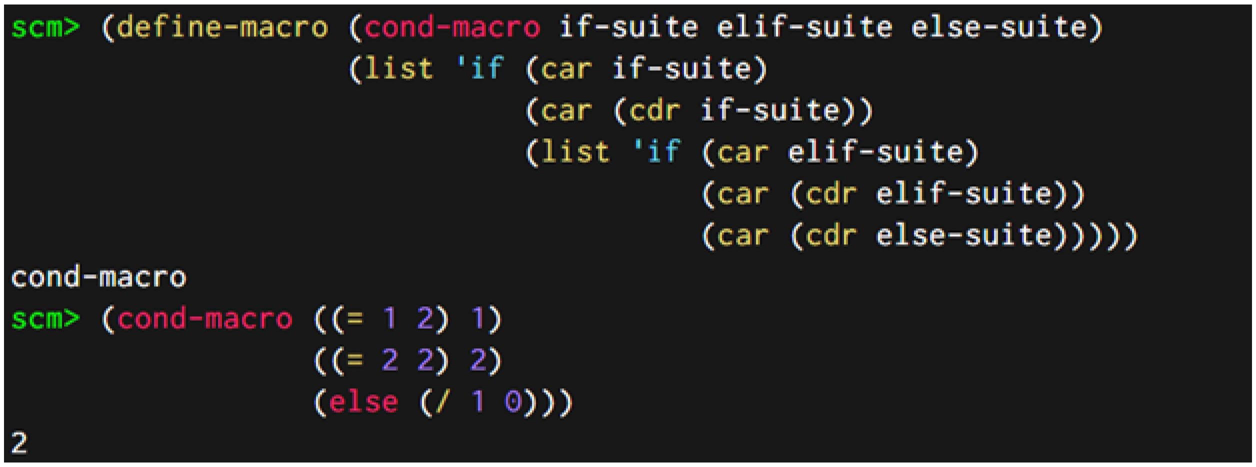
To re-emphasize, the only reason this is possible is because we don't evaluate the expressions when calling the macro. This allows us to rearrange those expressions when plugging them into an already-supported expression. **When writing macros, you should return an *expression* already supported by scheme that has equivalent behavior to the macro you're writing.**

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
```

3) 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 `repeat-n`! 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.

Wow. Aren't macros just so great?