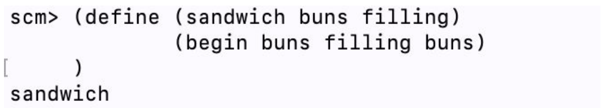
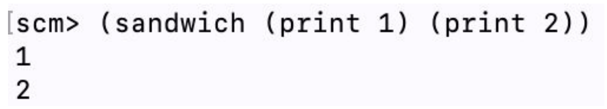
Amit Sant
amitsant2000@berkeley.edu

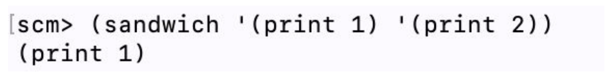
Macros

Most simply put, a macro is a way to have a function take in **expressions as inputs**

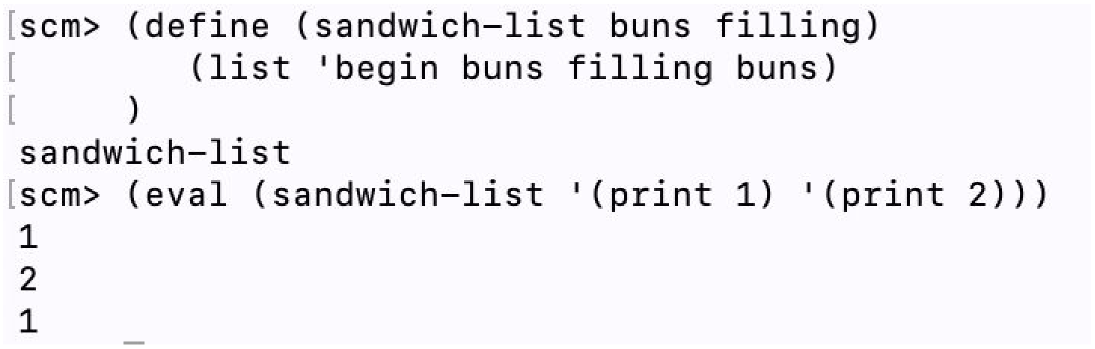
We cannot do this with regular define expressions alone:

Consider the function sandwich which is supposed to run the 'buns' expression before and after the 'filling' expression	 `scm> (define (sandwich buns filling)`
	`(begin buns filling buns)`
	`[ )`
	`sandwich`

We can't just put in expressions as parameters as they would evaluate before the function is even called	 `[scm> (sandwich (print 1) (print 2))`
	`1`
	`2`

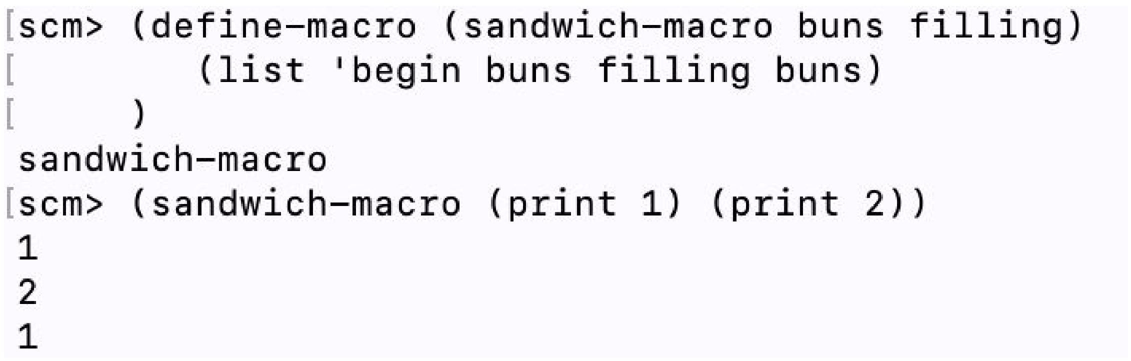
If we quote the expressions, they won't evaluate in the begin statement and it just ends up returning the 'buns' expression as it is the last element in the 'begin' statement	 `[scm> (sandwich '(print 1) '(print 2))` `(print 1)`

The solution is to have the function **return the entire body as a list and then eval it:**



```
[scm> (define (sandwich-list buns filling)
[         (list 'begin buns filling buns)
[     )
 sandwich-list
[scm> (eval (sandwich-list '(print 1) '(print 2)))
 1
 2
 1
```

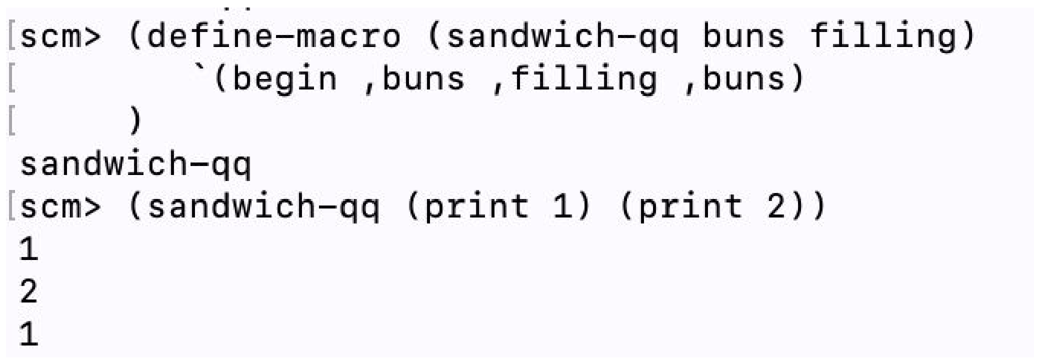
However, this is quite tedious as we have to 1) quote all the inputs and 2) eval the result;
Macros do both of these implicitly:



```
[scm> (define-macro (sandwich-macro buns filling)
[         (list 'begin buns filling buns)
[     )
 sandwich-macro
[scm> (sandwich-macro (print 1) (print 2))
 1
 2
 1
```

Amit Sant
amitsant2000@berkeley.edu

Instead of using lists, we can also use quasiquotes to get the same result:



```
[scm> (define-macro (sandwich-qq buns filling)
[         `(begin ,buns ,filling ,buns)
[      )
sandwich-qq
[scm> (sandwich-qq (print 1) (print 2))
1
2
1
```

Note that we have to unquote buns and filling as they were already quoted implicitly

Practice

Write a macro that does the same thing as the built-in if without using it *Hint: Use 'and' and 'or'*:

(define-macro (if cond t-suite f-suite)

    ____

)

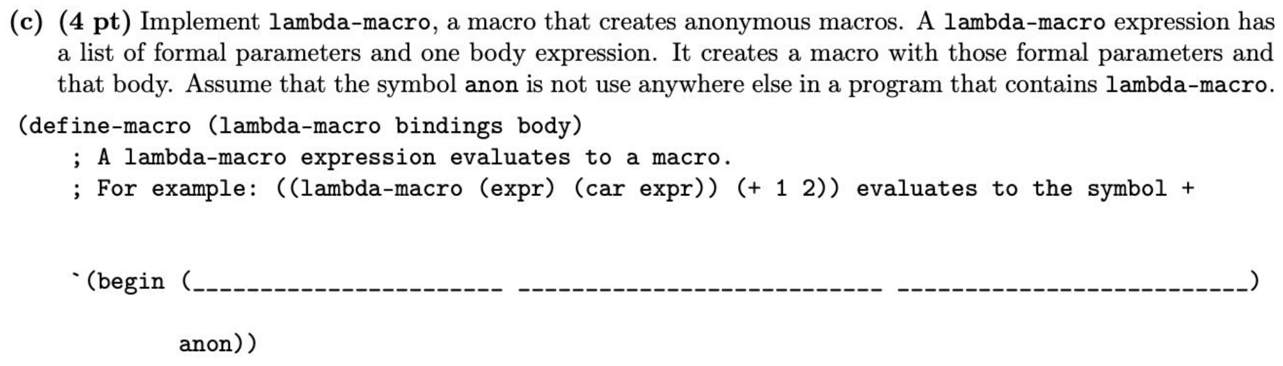
Write a macro that takes in a name a list of params and a body [stored as a list. ex: ((print 1) (print 2))] and creates a function with them

(define-macro (create-function name params body)

    ____

)

Now here's an exam problem from the Spring 2018 finals:



(c) **(4 pt)** Implement `lambda-macro`, a macro that creates anonymous macros. A `lambda-macro` expression has a list of formal parameters and one body expression. It creates a macro with those formal parameters and that body. Assume that the symbol `anon` is not use anywhere else in a program that contains `lambda-macro`.

```
(define-macro (lambda-macro bindings body)
    ; A lambda-macro expression evaluates to a macro.
    ; For example: ((lambda-macro (expr) (car expr)) (+ 1 2)) evaluates to the symbol +


    `(begin (______________________ _________________________ _________________________)

            anon))
```