

## A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

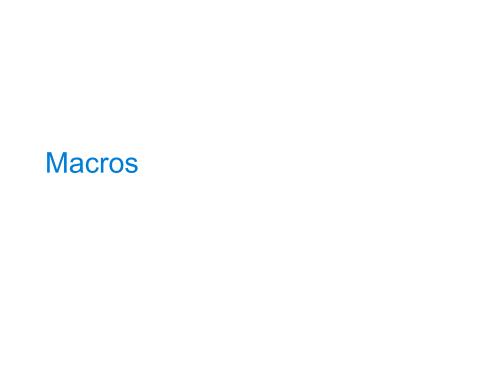
The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

(Demo)

```
scm> (+12)
3
scm> (list + 1 2)
(#[+] 1 2)
scm> (list '+ 1 2)
(+12)
scm> (+12)
scm> (list '+ 1 (+ 2 3))
(+15)
                                                          scm> (fact 5)
                                                          120
define (fact n)
 (if (= n 0) 1 (* n (fact (- n 1)))))
                                                          scm> (fact-exp 5)
                                                          (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
define (fact-exp n)
                                                          scm> (eval (fact-exp 5))
 (if (= n 0) 1 (list '* n (fact-exp (- n 1)))))
                                                          120
(define (fib n)
 (if (<= n 1) n (+ (fib (- n 2)) (fib (- n 1)))))
(define (fib-exp n)∏
 (if (<= n 1) n (list '+ (fib-exp (- n 2)) (fib-exp (- n 1)))))
                                 scm> (fib 6)
                                 8
                                 scm> (fib-exp 6)
                                 (+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1)) (+ (+ 0 1) (+ 1 (
                                 + 0 1)))))
```



## Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

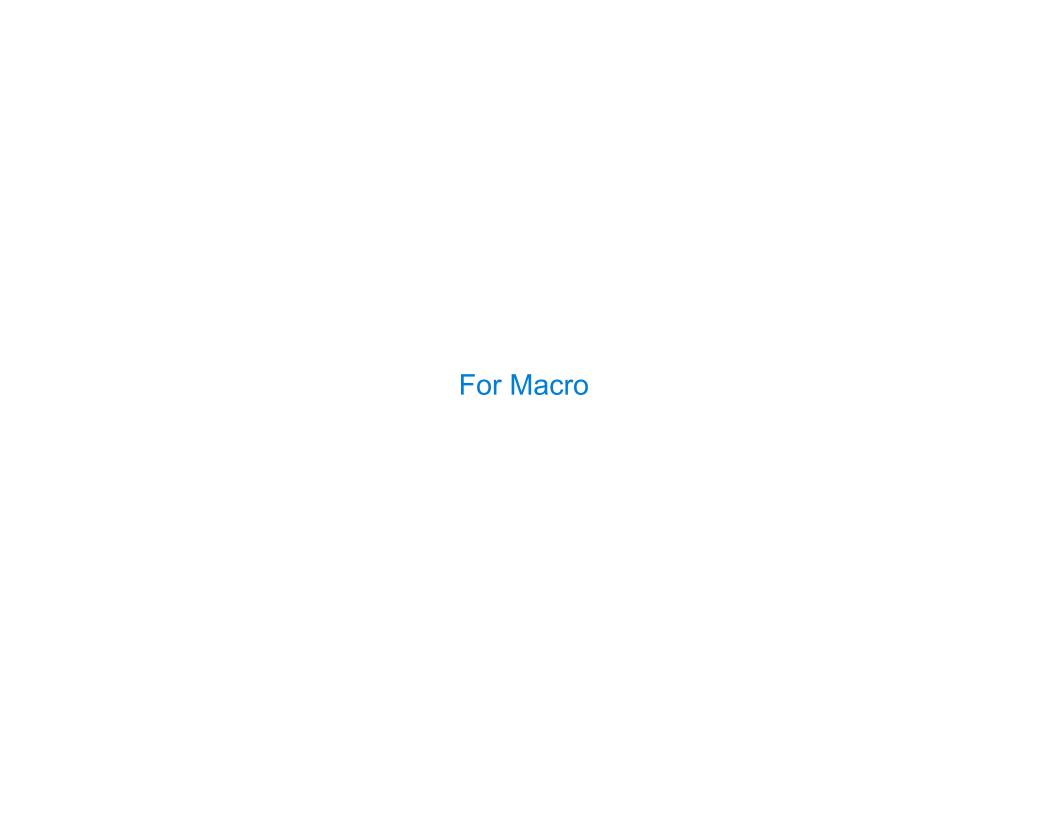
(Demo)

```
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
(begin None None) The value of (print 2) is evaluated and shown instead.
scm> (twice '(print 2)) To not evaluate it, a quote can be used.
(begin (print 2) (print 2))
scm> (eval (twice '(print 2))) However, eval is still needed to finish the work.
2
                                     Using define-macro can save the job.
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2 2
```

Now let's see another application.

```
scm> (define (check val) (if val 'passed 'failed))
check
scm> (define x -2)
x
scm> (check (> x 0))
failed
We want check to tell what expression is failed. -> define-macro
```

If you want to see what expressions define-macro creates, just delete -macro and quote the arg.



## **Discussion Question**

Define a macro that evaluates an expression for each value in a sequence

8

**Quasi-Quotation** 

(Demo)

```
scm> (define b 2)
b
scm> '(a b c)
(a b c)
scm> `(a b c)
(a b c)
scm> `(a ¬b c) unquote part of the expression
(a 2 c)
scm> `(a ,(+ b 5) c)
(a 7 c)
scm> `(a b ,c)
Traceback (most recent call last):
        (quasiquote (a b (unquote c)))
Error: unknown identifier: c
scm> '(a ,b c)
(a (unquote b) c) Quote cannot do so.
scm> (define expr '(* x x))
expr
scm> `(lambda (x) ,expr)
                           Quasi-quote makes it more convenient to create some expressions.
(lambda (x) (* x x))
```

We can use quasi-quote to rewrite check, so that it is more readable.