

## Programs as Data

---

## Announcements

## Programs as Data

## 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
```

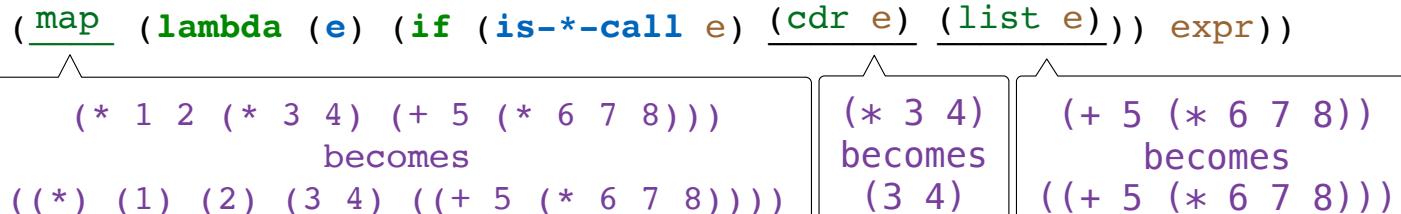
In such a language, it is straightforward to write a program that writes a program

(Demo)

## Discussion Question: Automatically Simplifying Code

```
scm> (* 1 2 (* 3 (* 4)) (+ 5 (* 6 (* 7 8))))  
8184  
scm> (flatten-nested-* '(* 1 2 (* 3 (* 4)) (+ 5 (* 6 (* 7 8)))))  
(* 1 2 3 4 (+ 5 (* 6 7 8)))  
scm> (* 1 2 3 4 (+ 5 (* 6 7 8)))  
8184  
scm> (eval (flatten-nested-* '(* 1 2 (* 3 (* 4)) (+ 5 (* 6 (* 7 8))))))  
8184  
(define (is-*-call expr) (and (list? expr) (equal? '* (car expr)))) ; E.g., (* 3 4)  
(define (flatten-nested-* expr) ; Return an equivalent expression with no nested calls to *  
(if (not (list? expr)) expr  
(let ((expr (map flatten-nested-* expr))) ; Now expr is (* 1 2 (* 3 4) (+ 5 (* 6 7 8)))  
(if (is-*-call expr)  
(apply append  
expr))))
```

result of applying append:  
(\* 1 2 3 4 (+ 5 (\* 6 7 8)))



## Discussion Question: Printing Evaluations

Define `print_evals`, which takes a Scheme expression `expr` that contains only numbers, `+`, `*`, `>`, `if` and parentheses. It prints all of the expressions that are evaluated during the evaluation of `expr` and their values. Print in the **order that evaluation completes**.

Assume every `if` expression has three sub-expressions: predicate, consequence, & alternative.

```
scm> (define expr '(* 2 (if (> 2 (+ 1 1)) (+ 3 4) (* 5 6))))
expr
scm> (eval expr)
60
scm> (print-evals expr)
* => #[*]
2 => 2
> => #[>]
2 => 2
+ => #[+]
1 => 1
1 => 1
(+ 1 1) => 2
(> 2 (+ 1 1)) => #f
* => #[*]
5 => 5
6 => 6
(* 5 6) => 30
(if (> 2 (+ 1 1)) (+ 3 4) (* 5 6)) => 30
(* 2 (if (> 2 (+ 1 1)) (+ 3 4) (* 5 6))) => 60
          (define (print-evals expr)
            (if (list? expr)
                (if (equal? (car expr) 'if)
                    (begin
                      (print-evals (car (cdr expr)))
                      (if (eval (car (cdr expr)))
                          (print-evals (car (cdr (cdr expr)))))
                          (print-evals (car (cdr (cdr (cdr (cdr expr))))))))
                      (map print-evals expr)
                    )))
                (print expr '=> (eval expr))))
```

# Quasiquotation

## Quasiquotation

---

There are two ways to quote an expression

Quote:        '(a b)    =>    (a b)

Quasiquote: ` (a b)    =>    (a b)

Parts of a quasiquoted expression can be unquoted with , to evaluate sub-expressions

(define b 4)

Quasiquote: ` (a ,(+ b 1))    =>    (a 5)

Quasiquotation is particularly convenient for generating Scheme expressions:

(define (make-add-lambda n) ` (lambda (d) (+ d ,n)))

(make-add-lambda 2)    => (lambda (d) (+ d 2))

## Discussion Question: Fact-Exp

---

Use quasiquotation to define **fact-expr**, a procedure that takes an integer n and returns a nested multiplication **expression** that evaluates to n **factorial**.

```
scm> (fact-expr 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

```
(define (fact-expr n)
  (if (= n 0) 1 `(* _____ ,n _____ , (fact-expr (- n 1)) _____ )))
```