CS61A Lecture 12

2011-07-11

Colleen Lewis

(calc) review

(scheme-1) has lambda but NOT define

Remember calc-apply?

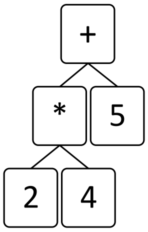
```
STk> (calc-apply '+ '(1 2 3))
6
STk> (calc-apply '* '(2 4 3))
24
STk> (calc-apply '/ '(10 2))
5
STk> (calc-apply '- '(9 2 3 1))
3
```

(calc) read-eval-print loop

```
(define (calc)
  (display "calc: ")
  (flush)
  (print (calc-eval (read)))
  (calc))
```

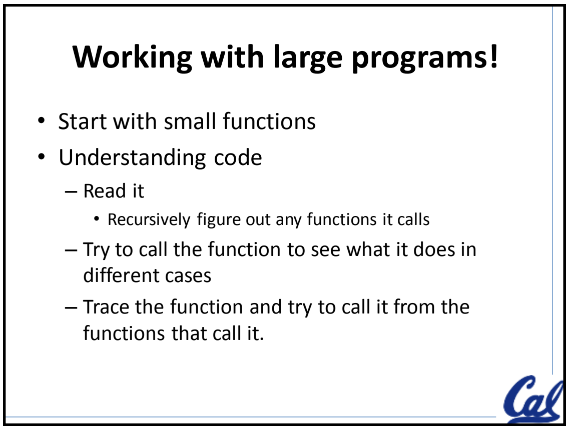
calc-eval

```
STk> (calc)
calc: (+ (* 2 4) 5)
13
(define (calc-eval exp)
  (cond
     ((number? exp) exp)
     ((list? exp)
         (calc-apply
            (car exp)
            (map calc-eval (cdr exp))))
    (else (error “Calc: bad exp”))))
```



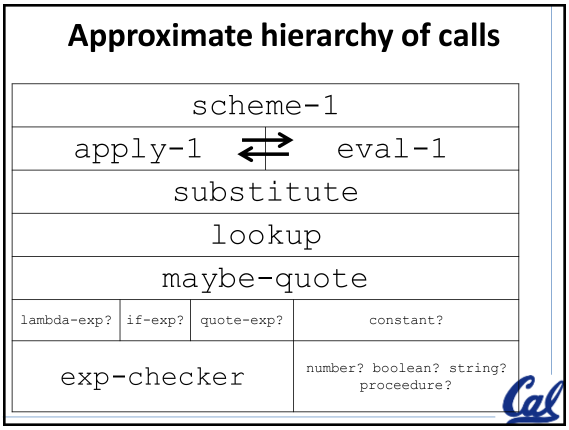
(scheme-1)
DOES NOT HAVE **DEFINE!**

```
STk> (scheme-1)

Scheme-1: (lambda (x) (* x x))
(lambda (x) (* x x))

Scheme-1: ((lambda (x) (* x x)) 3)
9
```



Working with large programs!

- Start with small functions
- Understanding code
  - Read it
    - Recursively figure out any functions it calls
  - Try to call the function to see what it does in different cases
  - Trace the function and try to call it from the functions that call it.



Approximate hierarchy of calls

scheme-1

apply-1 ⇄ eval-1

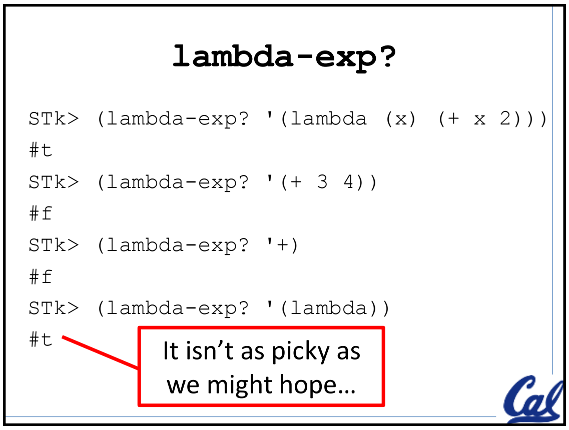
substitute

lookup

maybe-quote

lambda-exp? | if-exp? | quote-exp? | constant?

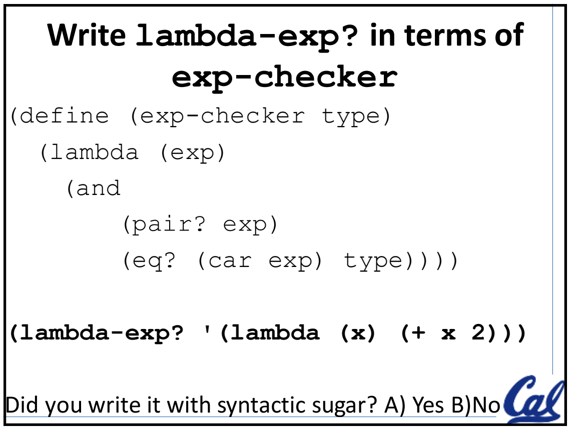
exp-checker | number? boolean? string? proceedure?



lambda-exp?

```
STk> (lambda-exp? '(lambda (x) (+ x 2)))
#t
STk> (lambda-exp? '(+ 3 4))
#f
STk> (lambda-exp? '+)
#f
STk> (lambda-exp? '(lambda))
#t
```

It isn't as picky as we might hope…

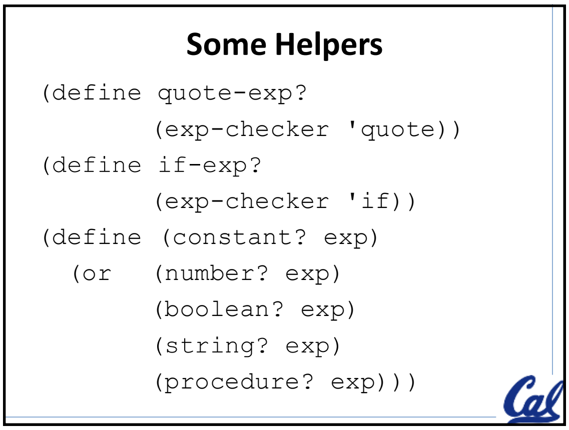


Write lambda-exp? in terms of exp-checker

```
(define (exp-checker type)
  (lambda (exp)
    (and
        (pair? exp)
        (eq? (car exp) type))))
```

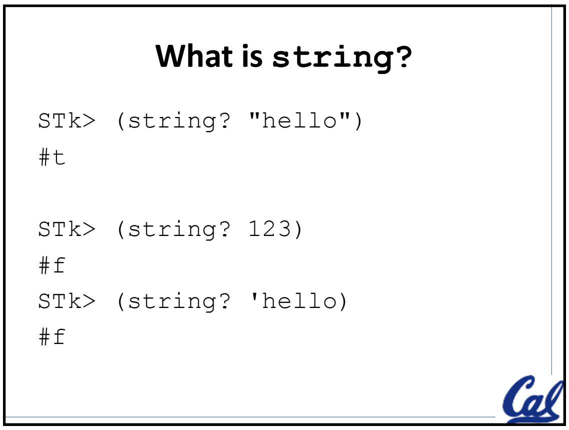
```
(lambda-exp? '(lambda (x) (+ x 2)))
```

Did you write it with syntactic sugar? A) Yes B)No



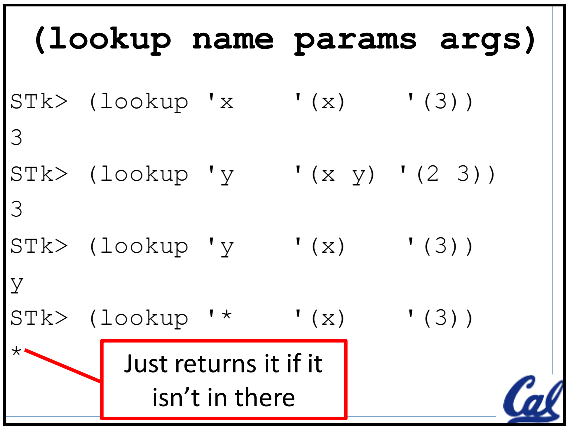
Some Helpers

```
(define quote-exp?
        (exp-checker 'quote))
(define if-exp?
        (exp-checker 'if))
(define (constant? exp)
   (or  (number? exp)
        (boolean? exp)
        (string? exp)
        (procedure? exp)))
```



What is string?

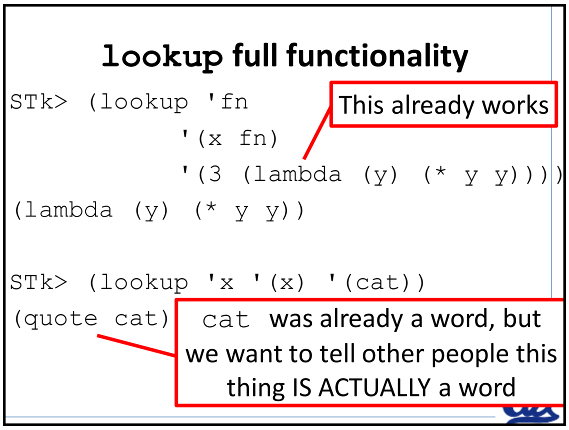
```
STk> (string? "hello")
#t

STk> (string? 123)
#f
STk> (string? 'hello)
#f
```



(lookup name params args)

```
STk> (lookup 'x    '(x)    '(3))
3
STk> (lookup 'y    '(x y) '(2 3))
3
STk> (lookup 'y    '(x)    '(3))
y
STk> (lookup '*    '(x)    '(3))
*
```

Just returns it if it isn't in there

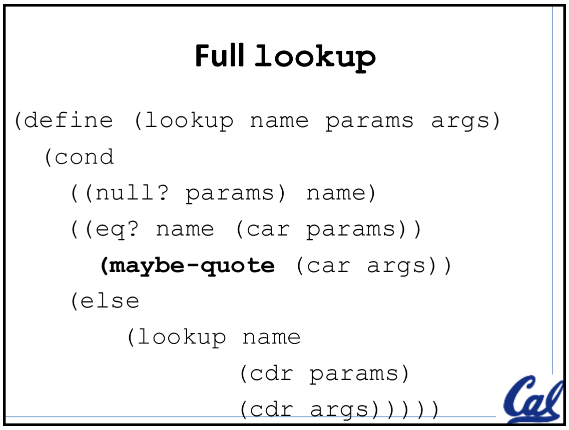


lookup full functionality

```
STk> (lookup 'fn
             '(x fn)
             '(3 (lambda (y) (* y y))))
(lambda (y) (* y y))

STk> (lookup 'x '(x) '(cat))
(quote cat)
```

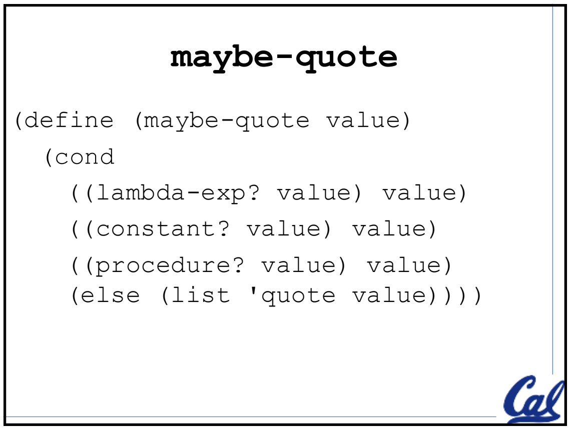
This already works

cat was already a word, but we want to tell other people this thing IS ACTUALLY a word



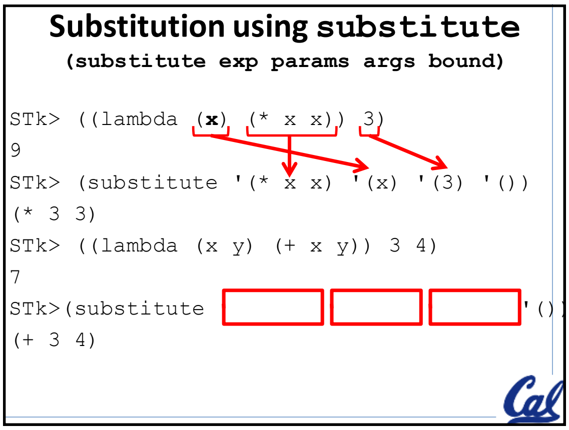
Full lookup

```
(define (lookup name params args)
  (cond
    ((null? params) name)
    ((eq? name (car params))
      (maybe-quote (car args))
    (else
        (lookup name
                (cdr params)
                (cdr args)))))
```



maybe-quote

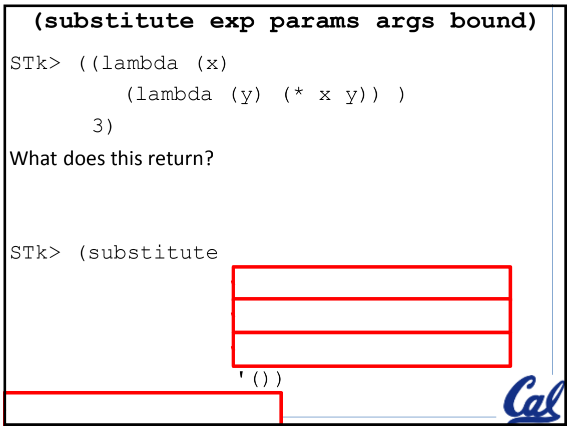
```
(define (maybe-quote value)
  (cond
    ((lambda-exp? value) value)
    ((constant? value) value)
    ((procedure? value) value)
    (else (list 'quote value))))
```



Substitution using substitute

(substitute exp params args bound)

```
STk> ((lambda (x) (* x x)) 3)
9
STk> (substitute '(* x x) '(x) '(3) '())
(* 3 3)
STk> ((lambda (x y) (+ x y)) 3 4)
7
STk>(substitute                      '())
(+ 3 4)
```

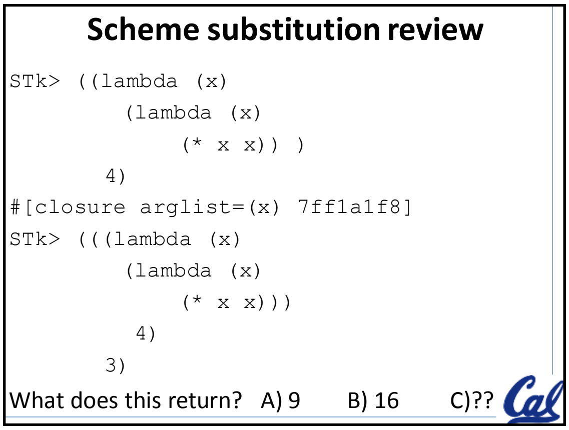


(substitute exp params args bound)

```
STk> ((lambda (x)
        (lambda (y) (* x y)) )
      3)
```

What does this return?

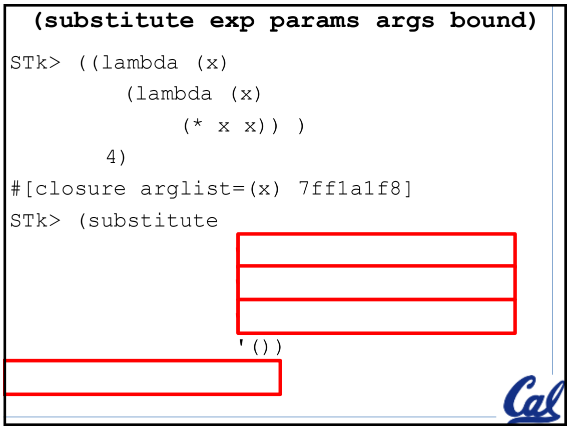
```
STk> (substitute



                       '())
```



Scheme substitution review

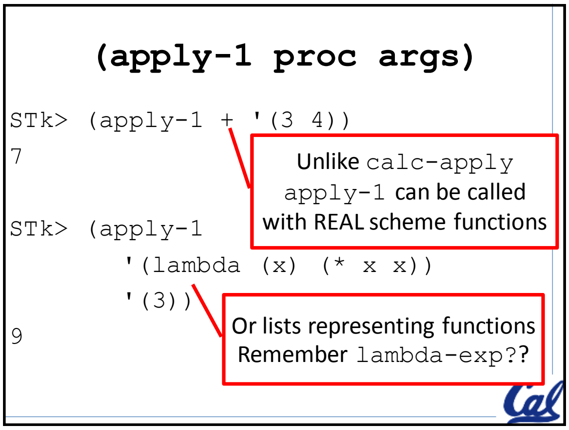
```
STk> ((lambda (x)
         (lambda (x)
              (* x x)) )
       4)
#[closure arglist=(x) 7ff1a1f8]
STk> (((lambda (x)
         (lambda (x)
              (* x x)))
          4)
       3)
```

What does this return? A) 9 B) 16 C)??



(substitute exp params args bound)

```
STk> ((lambda (x)
          (lambda (x)
               (* x x)) )
         4)
#[closure arglist=(x) 7ff1a1f8]
STk> (substitute
                      '())
```

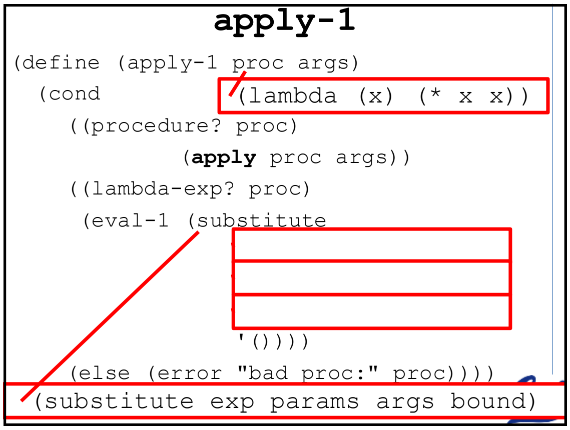


(apply-1 proc args)

```
STk> (apply-1 + '(3 4))
7

STk> (apply-1
        '(lambda (x) (* x x))
        '(3))
9
```

Unlike calc-apply apply-1 can be called with REAL scheme functions

Or lists representing functions Remember lambda-exp??

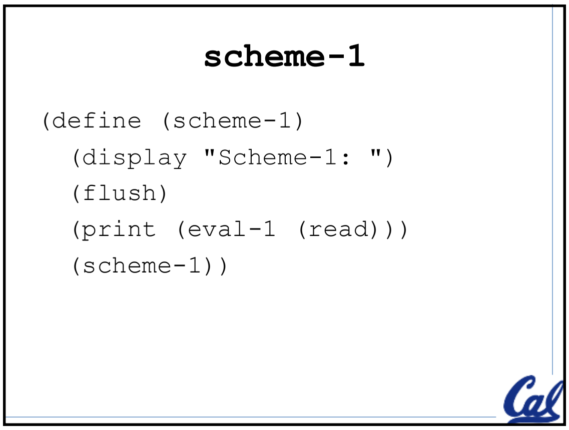


apply-1

```
(define (apply-1 proc args)
  (cond
    ((procedure? proc)
             (apply proc args))
    ((lambda-exp? proc)
     (eval-1 (substitute
                     '()))))
    (else (error "bad proc:" proc))))
```

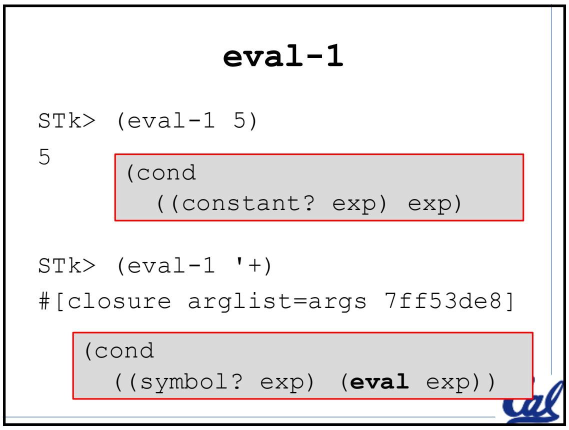
(lambda (x) (* x x))

(substitute exp params args bound)



scheme-1

```
(define (scheme-1)
  (display "Scheme-1: ")
  (flush)
  (print (eval-1 (read)))
  (scheme-1))
```



eval-1

```
STk> (eval-1 5)
5
```

```
(cond
  ((constant? exp) exp)
```

```
STk> (eval-1 '+)
#[closure arglist=args 7ff53de8]
```

```
(cond
  ((symbol? exp) (eval exp))
```

eval-1

```
STk> (eval-1 '(if (> 3 4) 5 7))
7
```

```
(cond
  ((if-exp? exp)
     (if (eval-1 (cadr exp))
         (eval-1 (caddr exp))
         (eval-1 (cadddr exp)))))
```

eval-1

```
STk> (eval-1 'x)
*** Error:
    unbound variable: x
Current eval stack:
__________________
  0    x
  1    (eval exp)
```

Things like + are quoted: '+ when they are passed to `eval-1` so this assumes x will be a variable not a word.

eval-1 with words

```
STk> (eval-1 '(quote x))
x

STk> (eval-1 (quote (quote x)))
x

STk> (eval-1 ''x)
x
```

```
(cond
  ((quote-exp? exp)(cadr exp))
```

These are all equivalent!

eval-1

```
STk> (eval-1 '(lambda (x) (* x x)))
(lambda (x) (* x x))
```

```
(cond
  ((lambda-exp? exp) exp)
```

```
(define (eval-1 exp)
  (cond
    ((constant? exp) exp)
    ((symbol? exp) (eval exp))
    ((quote-exp? exp) (cadr exp))
    ((if-exp? exp)
     (if (eval-1 (cadr exp))
         (eval-1 (caddr exp))
         (eval-1 (cadddr exp))))
    ((lambda-exp? exp) exp)
    ((pair? exp)__________________
    (else (error "?!?" exp))))
```

(cond ((pair? exp) ____)

```
STk> (eval-1 '(+ 2 3))
5

STk> (eval-1 '(+ (- 3 1) 5))
7

STk> (eval-1 '((lambda (x) (* x x)) 3))
9
```

(cond ((pair? exp) ____)

```
(cond
  ((pair? exp)
    (apply-1
        (eval-1 (car exp))
        (map eval-1 (cdr exp))))

STk> (eval-1 '(+ 2 3))
STk> (eval-1 '(+ (- 3 1) 5))
STk> (eval-1 '((lambda (x) (* x x)) 3))
```

lambda-exp? Solution

```
(define (exp-checker type)
  (lambda (exp)
    (and
        (pair? exp)
        (eq? (car exp) type))))

(define (lambda-exp? exp)
        ((exp-checker 'lambda) exp))
(define lambda-exp?
        (exp-checker 'lambda))
```

Write lookup (some functionality missing)

```
(define (lookup name params args)
  (cond
    ((null? params) name)
    ((eq? name (car params))
              (car args))
    (else
        (lookup name
              (cdr params)
              (cdr args)))))
```

Substitution using substitute

(substitute exp params args bound)

```
STk> ((lambda (x) (* x x)) 3)
9
STk> (substitute '(* x x) '(x) '(3) '())
(* 3 3)
STk> ((lambda (x y) (+ x y)) 3 4)
7
STk>(substitute '(+ x y) '(x y) '(3 4) '())
(+ 3 4)
```

(substitute exp params args bound)

```
STk> ((lambda (x)
        (lambda (y) (* x y)) )
      3)
```

What does this return?

#[closure arglist=(y) 7ff1cc48]

A procedure that takes argument y and adds 3 to it

```
STk> (substitute
                 '(lambda (y) (* x y))
                 '(x)
                 '(3)
                 '())
(lambda (y) (* 3 y))
```

(substitute exp params args bound)

```
STk> ((lambda (x)
        (lambda (x)
            (* x x)) )
      4)
#[closure arglist=(x) 7ff1a1f8]
STk> (substitute
                 '(lambda (x) (* x x))
                 '(x)
                 '(4)
                 '())
(lambda (x) (* x x))
```

A recursive call will be made where bound will be '(x)