TODAY

• Make a calculator program
  – To better understand how the Scheme interpreter works
  – STEP 1: calc-apply
  – STEP 2: list versus quote (Scheme primitives)
  – STEP 3: read (Scheme primitive)
  – STEP 4: read-print loop
  – STEP 5: read-eval-print loop
  – STEP 6: calc-eval
• deep-map

STEP 1: 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

(define (calc-apply fn-wd arg-list)
  (cond
    ((equal? fn-wd '+)
     (add-up-stuff-in arg-list))
    ((equal? fn-wd '-)
     (subtract-stuff-in arg-list))
    ((equal? fn-wd '*)
     (multiply-stuff-in arg-list))
    ((equal? fn-wd '/)
     (divide-stuff-in arg-list))
    (else
     (error "Calc: bad op: " fn-wd)))))

add-up-stuff-in

(define (add-up-stuff-in lst)
  (accumulate + 0 lst))

STk> (accumulate + 0 '(1 2 4))
.. -> + with args = (4 0)
.. <- + with args = 4
.. -> + with args = (2 4)
.. <- + with args = 6
.. -> + with args = (1 6)
.. <- + with returns 7
7

STEP 2: list versus quote

STk> '(1 2 +)
(1 2 +)
STk> (list 1 2 +)
(1 2 #[closure arglist=arglist 7ff53de8])
STEP 3: Demo (read)

I typed this!

After I hit return,
Scheme printed this

I didn’t have to quote words

' is really syntactic sugar for quote
(a special form)

Demo (read)

STk> (define a (read))
hello
a
STk> a
hello
STk>

Demo (read)

STk> (define b (read))
(+ 1 2)
b
STk> b
(+ 1 2)
STk> (car b)
+
STk> (list-ref b 1)
1

Not:
#|closure arglist=ARGS 7ff53de8|

Woah! read figured out it was a list within a list.

Demo (read)

STk> (define c (read))
(+ 3 (+ 1 2))
c
STk> (list-ref c 2)
(+ 1 2)
STk> (car c)
+

Demo (read)

STk> (define d (read))
(+ 3)
d
STk> d
(+ 3)

read Summary

• Prompts user for input
• NOT a function
• Whatever the user types it returns
  – They can type words (without quotes)
  – They can type numbers
  – They can type lists
    • If it looks like a list it waits for you to put necessary close parentheses
STEP 4: (read-print)

(define (read-print)
 (display "type here: ")
 (flush)
 (print (read))
 (read-print))

display: prints stuff
print: prints stuff on a new line
flush: Waits for user input
read-print: recursive call (infinite loop)

STk> (read-print)
type here: 4
4
type here: hi
hi
type here: (+ 1 2)
(+ 1 2)
type here: (+ (+ 3 4) 2)
(+ (+ 3 4) 2)
type here: Infinite loop!

I’m typing HERE not at STk>

(STk> (calc)
calc: 1
1
calc: (+ 2 3)
5
calc: (+ 2 (* 3 4))
14

STEP 5: Read-Eval-Print Loop

(calc) demo

STk> (calc)
calc: 1
1
calc: (+ 2 3)
5
calc: (+ 2 (* 3 4))
14

(calc) demo – it doesn’t have variables or “real” functions

calc: +
*** Error:
Calc: bad expression: +
Current eval stack:
STk> (calc)
calc: x
*** Error:
Calc: bad expression: x
Current eval stack:

(calc) read-eval-print loop

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

Representing Math

Translating to Scheme
(+ 1 2)
car: +
cdr: (2)
Representing Math in Scheme

\[(+ (* 2 4) 5)\]
\[\text{car: +} \]
\[\text{cdr: } ((* 2 4) 5)\]

Representing Math in Scheme

\[+\]
\[*\]
\[5\]
\[\text{car: } +\]
\[\text{cdr: } ((* 2 4) 5)\]

How many open parens?
A) 1  B) 2  C) 3  D) 4  E) 5

Remember the \texttt{(calc)} read-eval-print loop?

\begin{verbatim}
(define (calc)
  (display "calc: ")
  (flush)
  (print \texttt{(calc-eval (read))})
  (calc))
\end{verbatim}

\texttt{calc-eval basic case}

STk> (calc)
\texttt{calc: 1}
1
\begin{verbatim}
(define (calc-evl exp)
  (cond
   ((number? exp) exp)
   ((list? exp) _______
    (else (error "Calc: bad exp")))))
\end{verbatim}

\texttt{calc-eval}

STk> (calc)
\texttt{calc: (+ 1 2)}
3
\begin{verbatim}
(define (calc-evl exp)
  (cond
   ((number? exp) exp)
   ((list? exp)
    (calc-evl (car exp))
    (map calc-evl (cdr exp)))
   (else (error "Calc: bad exp"))))
\end{verbatim}

\texttt{calc-eval}

STk> (calc)
\texttt{calc: (+ (* 2 4) 5)}
40
\begin{verbatim}
(define (calc-evl exp)
  (cond
   ((number? exp) exp)
   ((list? exp)
    (calc-evl (car exp))
    (map calc-evl (cdr exp)))
   (else (error "Calc: bad exp"))))
\end{verbatim}
**calc-eval**

```
STk> (calc)
calc: (+ (* 2 4) 5)
40
```

```scheme
(define (calc-eval exp)
  (cond
   ((number? exp) exp)
   ((list? exp)
    (calc-eval
     (car exp))
    (map calc-eval (cdr exp))))
   (else (error "Calc: bad exp"))))
```

```
(calculate exp)
Works for bigger trees!
```

```
+ 1 2
* 4 5
```

```
+ 3 * 2 2
```

```
+ 3 3
```

**deep-map**

```
STk> (map square '(1 2 3))
(1 4 9)
STk> (deep-map square '(1 2 3))
(1 4 9)
STk> (deep-map square '((3 . 4) (5 6)))
((9 . 16) (25 36))
STk> (deep-map square 3)
9
STk> (deep-map square '())
()
```

```
Remember map? Meet deep-map
```

```
draw '((3 . 4) (5 6))
```

```
deep-map base cases
```

```
STk> (deep-map square 3)
9
STk> (deep-map square '())
()
(define (deep-map fn arg)
  (cond
   ((null? arg) '())
   ((pair? arg) _____________
    (else (fn arg))))
```

```
How many pairs?
A) 1  B) 2  C) 3  D) 4  E) 5
```
(deep-map sq '((3 . 4) (5 6)))

(map)
(define (map fn seq)
  (if (null? seq)
    '()
    (cons (fn (car seq))
      (map fn (cdr seq)))))

Representing Math in Scheme SOLUTION
(+ (* 3 (+ 2 1) 4) 5)
  How many open parens?
A) 1  B) 2  C) 3  D) 4  E) 5

SOLUTION

Draw `((3 . 4) (5 6))

Deep-map solution
(define (deep-map fn arg)
  (cond
    ((null? arg) '())
    ((pair? arg)
      (cons
        (deep-map fn (car arg))
        (deep-map fn (cdr arg))))
    (else (fn arg)))))