Computer Science I

Professor Tom Ellman
Lecture 5

Summing a List of Numbers

```
(define sum (lambda (x) …?…))

Welcome to DrScheme
> (sum '(1 2 3))
6
> (sum '(18  21  36))
75
> (sum '(7))
7
> (sum '())
0
```

Define a Separate Procedure for Each Possible Length of the List

```
(define sum-zero  (lambda (x) 0))

(define sum-one   (lambda (x) (+ (car x) 0)))

(define sum-two   (lambda (x) (+ (car x)
                                  (+ (car (cdr x)) 0))))

(define sum-three (lambda (x) (+ (car x)
                                  (+ (car (cdr x))
                                     (+ (car (cdr (cdr x))) 0)))))
… Etc. ...
```

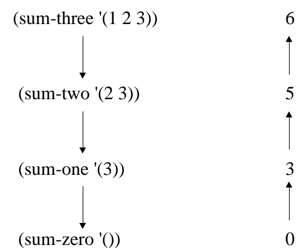
Problems with this Approach

- We might not know the length of our longest list of numbers.
- We don't know how many different sum procedures we need to write.
- Nearly all of the procedures follow a single general pattern.
- We are wasting effort in writing definitions that fit the pattern over and over.

Defining Each Sum Procedure in Terms of a Simpler One

```
(define sum-zero  (lambda (x) 0))

(define sum-one   (lambda (x) (+ (car x) (sum-zero (cdr x)))))

(define sum-two   (lambda (x) (+ (car x) (sum-one  (cdr x)))))

(define sum-three (lambda (x) (+ (car x) (sum-two  (cdr x)))))

… Etc. ...
```

Procedure Calls and Return Values



```
(sum-three '(1 2 3))

((lambda (x) (+ (car x) (sum-two  (cdr x)))) '(1 2 3))

(+ (car '(1 2 3)) (sum-two  (cdr '(1 2 3))))

(+ 1 ((lambda (x) (+ (car x) (sum-one  (cdr x))))  '(2 3)))

(+ 1 (+ (car '(2 3)) (sum-one  (cdr '(2 3)))))

(+ 1 (+ 2 ((lambda (x) (+ (car x) (sum-zero (cdr x)))) '(3))))

(+ 1 (+ 2 (+ (car '(3)) (sum-zero (cdr '(3))))))
```

```
(+ 1 (+ 2 (+ (car '(3)) (sum-zero (cdr '(3))))))

(+ 1 (+ 2 (+ 3 (sum-zero '()))))

(+ 1 (+ 2 (+ 3 ((lambda (x) 0) '()))))

(+ 1 (+ 2 (+ 3 0)))

(+ 1 (+ 2 3))

(+ 1 5)

6
```

Split the Problem into Two Cases

- Notice that SUM-ZERO is different from all of the others.
- Notice that SUM-ONE, SUM-TWO, and SUM-THREE … are nearly identical.
- Let's use (IF …) to handle these two groups separately.

```
(define sum (lambda (x) (if (null?  x)
                        <Answer for Zero Length List>
                        <Answer for Non-Zero Length List>)))
```

⬇

```
(define sum (lambda (x) (if (null?  x)
                        0
                        <Answer for Non-Zero Length List>)))
```

⬇

```
(define sum (lambda (x) (if (null?  x)
                        0
                        (+  (car x)  <Answer for (cdr x)> ))))
```

Wishful Thinking

- Let's pretend that we have already written a procedure called "SUM" that works for lists of all possible lengths.
- We shall use the "SUM" procedure to compute "<Answer for (cdr x)>" in our procedure definition.

Recursive Definition of "SUM"

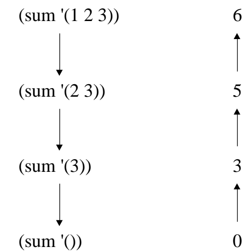
```
(define sum (lambda (x) (if (null?  x)
                        0
                        (+  (car x)  (sum (cdr x))))))
```

The procedure SUM is defined by a lambda expression that contains a reference to the SUM procedure itself!!!!!!!!!!!!!

Isn't this a Circular Definition?

- Not quite.
- To evaluate "(SUM X)" we may need to evaluate "(SUM (CDR X))".
- To evaluate "(SUM (CDR X))" we may need to evaluate "(SUM (CDR (CDR X)))".
- …Etc. ...
- Eventually we need only to evaluate the expression "(SUM '())", which is zero.

Procedure Calls and Return Values




```
(sum '(1 2 3))

((lambda (x) (if (null? x)
                 0
                 (+ (car x) (sum (cdr x)))))
 '(1 2 3))

(if (null? '(1 2 3))
    0
    (+ (car '(1 2 3)) (sum (cdr '(1 2 3)))))

(+ (car '(1 2 3)) (sum (cdr '(1 2 3))))

(+ 1 (sum '(2 3)))
```

```
(+ 1 (sum '(2 3)))

(+ 1 ((lambda (x) (if (null? x)
                      0
                      (+ (car x) (sum (cdr x)))))
      '(2 3)))

(+ 1 (if (null? '(2 3))
         0
         (+ (car '(2 3)) (sum (cdr '(2 3))))))

(+ 1 (+ (car '(2 3)) (sum (cdr '(2 3)))))

(+ 1 (+ 2 (sum '(3))))
```

```
(+ 1 (+ 2 (sum '(3))))

(+ 1 (+ 2 ((lambda (x) (if (null? x)
                           0
                           (+ (car x) (sum (cdr x)))))
           '(3))))

(+ 1 (+ 2 (if (null? '(3))
              0
              (+ (car '(3)) (sum (cdr '(3)))))))

(+ 1 (+ 2 (+ (car '(3)) (sum (cdr '(3))))))

(+ 1 (+ 2 (+ 3 (sum '()))))
```

```
(+ 1 (+ 2 (+ 3 (sum '()))))

(+ 1 (+ 2 (+ 3 ((lambda (x) (if (null? x)
                                0
                                (+ (car x) (sum (cdr x)))))
                '()))))

(+ 1 (+ 2 (+ 3 (if (null? '())
                   0
                   (+ (car '()) (sum (cdr '()))))))))

(+ 1 (+ 2 (+ 3  0)))

6
```

Recursive Definition of "SUM"

Stopping Condition

Base Case

Inductive Case

```
(define sum
  (lambda (x)  (if  (null?  x)
                    0
                    (+  (car x)  (sum (cdr x)))  )))
```

Increment All Numbers on a List

```
(define increment (lambda (x)  (+ x 1)))

(define increment-all (lambda (x) …?…))

Welcome to DrScheme
> (increment-all '(1 2 3))
(2  3  4)
> (increment-all '(18  21  36))
(19  22  37)
> (increment-all '())
()
```

Defining Each Increment Procedure in Terms of a Simpler One

```
(define increment-zero (lambda (x) '()))

(define increment-one (lambda (x) (cons (increment (car x))
                                        (increment-zero (cdr x)))))

(define increment-two (lambda (x) (cons (increment (car x))
                                        (increment-one (cdr x)))))

(define increment-three (lambda (x) (cons (increment (car x))
                                          (increment-two (cdr x)))))

… Etc. ...
```

Split the Problem into Two Cases

```
(define increment-all
    (lambda (x)
        (if  (null?  x)
            '()
            (cons (increment (car x))
                  <Answer for (cdr x)>))))
```

Recursive Definition of "INCREMENT-ALL"

```
(define increment-all
    (lambda (x)
        (if  (null?  x)
            '()
            (cons (increment (car x))
                  (increment-all (cdr x))))))
```

Procedure Calls and Return Values

(increment-all '(1 2 3))	(2 3 4)
↓	↑
(increment-all '(2 3))	(3 4)
↓	↑
(increment-all '(3))	(4)
↓	↑
(increment-all '())	( )

```
(increment-all '(1 2 3))

((lambda (x)  (if (null? x)
                  '()
                  (cons (increment (car x)) (increment-all (cdr x)))))
 '(1 2 3))

(if (null? '(1 2 3))
    '()
    (cons (increment (car '(1 2 3))) (increment-all (cdr '(1 2 3)))))

(cons (increment (car '(1 2 3))) (increment-all (cdr '(1 2 3))))

(cons 2 (increment-all '(2 3)))
```

```
(cons 2 (increment-all '(2 3)))

...

(cons 2 (cons 3 (increment-all '(3))))

...

(cons 2 (cons 3 (cons 4 (increment-all '()))))

(cons 2 (cons 3 (cons 4 '())))

(2 3 4)
```

Flip Each Element of a List

```
(define flip (lambda (x) (cons (car (cdr x)) (cons (car x) '()))))

(define flip-all (lambda (x) …?…))

Welcome to DrScheme
> (flip '(fred ethel))
(ethel fred)
> (flip-all '((fred ethel) (lucy rickey) (bill kenneth)))
'((ethel fred) (rickey lucy) (kenneth bill)))
> (flip-all '((1  2) (3  4) (5  6)))
((2  1) (4  3) ( 6  5))
> (flip-all '())
()
```

Defining Each Flip Procedure in Terms of a Simpler One

```
(define flip-zero (lambda (x) '()))

(define flip-one (lambda (x) (cons (flip (car x)) (flip-zero (cdr x)))))

(define flip-two (lambda (x) (cons (flip (car x)) (flip-one (cdr x)))))

(define flip-three (lambda (x) (cons (flip (car x)) (flip-two (cdr x)))))

… Etc. ...
```

Split the Problem into Two Cases

```
(define flip-all
      (lambda (x)
            (if  (null?  x)
                 '()
                 (cons (flip (car x))
                          <Answer for (cdr x)>))))
```

Recursive Definition of “FLIP-ALL”

```
(define flip-all
      (lambda (x)
            (if  (null?  x)
                 '()
                 (cons (flip (car x))
                          (flip-all (cdr x))))))
```

Procedure Calls and Return Values

(flip-all '((f e) (r l) (b k))) → ((e f) (l r) (k b))

(flip-all '((r l) (b k))) → ((l r) (k b))

(flip-all '((b k))) → ((k b))

(flip-all '()) → ( )

The "FACTORIAL" Procedure

```
(define factorial (lambda (n) …?…))

Welcome to DrScheme
> (factorial 4)
24
> (* 4 (* 3 (* 2 (* 1 1))))
24
>(factorial 3)
6
> (* 3 (* 2 (* 1 1)))
6
> (factorial 1)
1
> (factorial 0)
1
```

Defining Each Factorial Procedure in Terms of a Simpler One

```
(define factorial-zero (lambda (n) 1))

(define factorial-one (lambda (n) (* 1 (factorial-zero (- n 1)))))

(define factorial-two (lambda (n) (* 2 (factorial-one (- n 1)))))

(define factorial-three (lambda (n) (* 3 (factorial-two (- n 1)))))

… Etc. ...
```

Split the Problem into Two Cases

```
(define factorial
     (lambda (n)
          (if (= n 0)
              1
              (* n <Answer for (- n 1)>))))
```

Recursive Definition of "FACTORIAL"

```
(define factorial
     (lambda (n)
          (if (= n 0)
              1
              (* n (factorial (- n 1))))))
```

Procedure Calls and Return Values

(factorial 3) → 6

(factorial 2) → 2

(factorial 1) → 1

(factorial 0) → 1