Warm-up: Functions with conditional expressions

Class time

(define class-time?
  (lambda (time)
    ...?...))

> (class-time? 830)
#f
> (class-time? 1000)
#t
> (class-time? 1250)
#f

This works, but it's bad coding style; it makes the function longer and more confusing than it needs to be.
Class time

(define class-time?
  (lambda (time)
    (and (>= time 900)
         (<= time 1015))))

This states the function very briefly: It will return true exactly when the two tests are true.

Lists of specific lengths

*Singleton*: One element, e.g., '(a)

In other words, a list whose rest is '().

(define singleton?
  (lambda (x)
    (and (list? x)
         (null? (rest x)))))

*Doubleton*: Two elements, e.g., '(a b)

In other words, a list whose rest is a singleton.

(define doubleton?
  (lambda (x)
    (and (list? x)
         (singleton? (rest x)))))

*Tripleton*: Three elements, e.g., '(a b c)
**Tripleton**: Three elements, e.g., '(a b c)
In other words, a list whose rest is a *doubleton*.

```
(define tripleton?
  (lambda (x)
    (and (list? x)
      (doubleton? (rest x))))
```

**Recursion on flat lists**

**Summing a list of numbers**

```
(define sum (lambda (x) ...?...))
```

> (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)
    (+ (first x)
      0)))
```

```
(define sum-two
  (lambda (x)
    (+ (first x)
      (+ (first (rest x))
        0))))
```

```
(define sum-three
  (lambda (x)
    (+ (first x)
      (+ (first (rest x))
        (+ (first (rest (rest x)))
          0)))))
```

Etc.
Problem with this approach

Nearly all of the procedures follow a single general pattern.

We're 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)
    (+ (first x) (sum-zero (rest x)))))
(define sum-two
  (lambda (x)
    (+ (first x) (sum-one (rest x)))))
(define sum-three
  (lambda (x)
    (+ (first x) (sum-two (rest x)))))

... Etc. ...
```

sum-three: Procedure calls and return values

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sum-three '(1 2 3))</td>
<td>6</td>
</tr>
<tr>
<td>(sum-two '(2 3))</td>
<td>5</td>
</tr>
<tr>
<td>(sum-one '(3))</td>
<td>3</td>
</tr>
<tr>
<td>(sum-zero '())</td>
<td>0</td>
</tr>
</tbody>
</table>

sum-three: Evaluation

```
(sum-three '(1 2 3))
((lambda (x)
    (+ (first x)
      (sum-two (rest x)))
  '(1 2 3))
(+ (first '(1 2 3))
  (sum-two (rest '(1 2 3))))
(+ 1
  ((lambda (x)
      (+ (first x)
        (sum-one (rest x)))
    '(2 3)))
(+ 1
  (+ (first '(2 3))
    (sum-one (rest '(2 3)))))
```
**sum-three: Evaluation, 2**

```
(+ 1
  (+ (first '(2 3))
      (sum-one (rest '(2 3)))))
(+ 1
  (+ 2 ((lambda (x)
        (+ (first x)
            (sum-zero (rest x))))
        '(3))))
(+ 1
  (+ 2
      (+ (first '(3))
          (sum-zero (rest '(3))))))
```

**sum-three: Evaluation, 3**

```
(+ 1
  (+ 2
      (+ (first '(3))
          (sum-zero (rest '(3)))))
(+ 1 (+ 2 (+ 3 (sum-zero '())))
(+ 1 (+ 2 (+ 3 ((lambda (x) 0) '())))
(+ 1 (+ 2 (+ 3 0)))
(+ 1 (+ 2 3))
(+ 1 5)
6
```

**Problem with this case**

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.

**Split the problem into two cases**

Notice that:

- `sum-zero` is different from all of the others.
- `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>))))

Wishful thinking

Let’s pretend that we have already written a procedure called “sum” that works for lists of all possible lengths.

We’ll use the sum procedure to compute <answer for (rest x)> in our procedure definition.

Recursive definition of sum

(define sum
  (lambda (x)
    (if (null? x)
        0
        (+ (first x)
            (sum (rest x))))))

Isn’t this a circular definition?

Not quite.

The procedure sum is defined by a lambda expression that contains a reference to the sum procedure itself!
To evaluate \((\text{sum } x)\), we may need to evaluate \((\text{sum } (\text{rest } x))\).

To evaluate \((\text{sum } (\text{rest } x))\), we may need to evaluate \((\text{sum } (\text{rest } (\text{rest } x)))\).

...Etc. ...

How does it stop?
You need to provide a stopping condition – the base case.
Eventually we need only to evaluate the expression \((\text{sum } '()\)), which is zero.

**sum: Procedure calls and return values**

\[
\begin{array}{c}
(\text{sum } '((1 2 3))) & 6 \\
(\text{sum } '((2 3))) & 5 \\
(\text{sum } '((3))) & 3 \\
(\text{sum } '()) & 0 \\
\end{array}
\]

**Spoiler:** You don’t need to take my word for it!

(require racket/trace)
(define sum ...)
(trace sum)
(\text{sum } '((1 2 3 4)))
**sum: Evaluation**

(sum '(1 2 3))

((lambda (x)
  (if (null? x)
    0
    (+ (first x) (sum (rest x)))))
 '(1 2 3))

(if (null? '(1 2 3))
  0
  (+ (first '(1 2 3))
    (sum (rest '(1 2 3)))))

(+ (first '(1 2 3))
  (sum (rest '(1 2 3)))))

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

**sum: Evaluation, 2**

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

(+ 1
  ((lambda (x)
    (if (null? x)
      0
      (+ (first x) (sum (rest x)))))
   '(2 3)))

(if (null? '(2 3))
  0
  (+ (first '(2 3))
    (sum (rest '(2 3)))))

**sum: Evaluation, 3**

(+ 1
  (if (null? '(2 3))
    0
    (+ (first '(2 3))
      (sum (rest '(2 3)))))

(+ 1
  (+ (first '(2 3))
    (sum (rest '(2 3)))))

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

(+ 1 (+ 2
  ((lambda (x)
    (if (null? x)
      0
      (+ (first x) (sum (rest x)))))
   '(3)))))

**sum: Evaluation, 4**

(+ 1 (+ 2
  ((lambda (x)
    (if (null? x)
      0
      (+ (first x) (sum (rest x)))))
   '(3)))

(if (null? '(3))
  0
  (+ (first '(3))
    (sum (rest '(3)))))

(+ 1 (+ 2
  (+ (first '(3))
    (sum (rest '(3)))))))

(+ 1 (+ 2 (+ 3 (sum '()))))
**sum**: Evaluation, 4

\[ (+1 + 2 (+3 (\text{sum }')))) \]

\[ (+1 + 2 (+3 ((\lambda(x) \quad (\text{if } \text{null? } x) \quad 0 \quad (+ (\text{first } x) (\text{sum } (\text{rest } x))))')())) \]

\[ (+1 + 2 (+3 (\text{if } \text{null? } '()) \quad 0 \quad (+ (\text{first } '()) \quad (\text{sum } (\text{rest } '()))))) \]

\[ (+1 + 2 (+3 0)) \]

6

**Recursive definition of sum**

Stopping condition

Base case

Recursive case

(define sum
  (\lambda(x) \quad (\text{if } \text{null? } x) \quad 0 \quad (+ (\text{first } x) (\text{sum } (\text{rest } x))))))

**Problem: Square all numbers on a list**

> (square-all '(1 2 3))

(1 4 9)

> (square-all '())

()
Defining each square procedure in terms of a simpler one

(define square-zero
  (lambda (x) '()))
(define square-one
  (lambda (x)
    (cons (square (first x))
          (square-zero (rest x)))))
(define square-two
  (lambda (x)
    (cons (square (first x))
          (square-one (rest x)))))
(define square-three
  (lambda (x)
    (cons (square (first x))
          (square-two (rest x)))))

... Etc. ...

Split the problem into two cases

(define square-all
  (lambda (x)
    (if (null? x)
        ;; Base case:
        ;;   List is empty; return the empty list
        '()
      ;; Recursive case:
      ;;   List has at least one element; square it and
      ;;   cons it onto the front of your answer.
      (cons (square (first x))
            (square-all (rest x)))))

Recursive definition of square-all

(define square-all
  (lambda (x)
    (if (null? x)
        ;; Base case:
        ;;   List is empty; return the empty list
        '()
      ;; Recursive case:
      ;;   List has at least one element; square it and
      ;;   cons it onto the front of your answer.
      (cons (square (first x))
            (square-all (rest x))))))

(tester '(square-all '(1 2 3 4 5)))
(tester '(square-all '(-1 2 -3)))

Square-all: Procedure calls and return values

(square-all '(1 2 3)) (1 4 9)
(square-all '(2 3)) (4 9)
(square-all '(3)) (9)
(square-all '()) ()
We can do a similar thing for the cube function:

```
(define cube
  (lambda (x)
    (* x x x)))
(define cube-all
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (cube (first lst))
              (cube-all (rest lst))))))
(tester '(cube-all '(1 2 3 4)))
```

In fact, we can do the same thing for *any* procedure proc:

```
(define proc-all
  (lambda (proc lst)
    (if (null? lst)
        ;; Base case: List is empty; return empty list.
        '()
        ;; Recursive case: List is non-empty. Cons result
        ;; of applying proc to first of lst...
        (cons (proc (first lst))
              ;; ...together with the rest of the answer.
              (proc-all proc (rest lst))))))
(define test-proc (lambda (x) (* x 100)))
(tester '(proc-all test-proc '(1 2 3)))
```

For proc-all, proc is a function being passed as a parameter to another function!

The particular abstraction we just defined is so convenient it's part of the programming language – it's the map function.

Flat recursion design pattern

```
(define flat-recursion
  (lambda (x)
    (if (stop-cond? x)
        base-case
        (rec-case (first x)
                  (flat-recursion (rest x))))))
```

Stopping condition

Base case

Recursive case
Flat recursion design pattern

*stop-cond?* is a predicate

*base-case* is a constant, literal value

*rec-case* is a procedure of two parameters

Determine the length of a list

Scheme provides a built-in length function to compute the length of a list – but we can define our own!

```scheme
(define compute-length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (compute-length (rest lst))))))
(tester '((compute-length '(1 2 3 4 5)))
(tester '((compute-length '(a b c d e f g h)))
(tester '((compute-length '()))
```

Fetch *n*th element of a list

In computer science, we often index elements of a list starting with index 0.

Thus, a list containing *n* elements would have its elements indexed as (0, 1, 2, ..., *n*-1)

Fetch *n*th element of a list

Scheme provides a built-in function `list-ref` that fetches the *n*th 0-indexed element of a list – but we can define our own!

```scheme
(define fetch-nth-elt
  (lambda (lst n)
    ;; Base case: n = 0
    (if (= n 0)
      ;; Zeroth element is the "first"
      (first lst)
      ;; Recursive case:
      (fetch-nth-elt (rest lst) (- n 1))))
(tester '((fetch-nth-elt '(0 1 2 3 4 5) 0)
(tester '((fetch-nth-elt '(0 1 2 3 4 5) 3)
Acknowledgments

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger