

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.S184—Zombies Drink Caffeinated 6.001
 IAP 2012

Lecture 2

Lists, HOPs, and symbols

Flying first-class

Procedures are first-class objects in Scheme. They may be passed in as parameters, stored in variables, and returned from functions.

Write Scheme expressions with the following names and behaviors:

1. `divide-by`: Given a number, return a procedure that accepts a number and divides it by this first number
2. `square-and-add`: Given a number, return a procedure that accepts a number, squares it, and adds the first number
3. `compose`: Given two procedures, return a procedure which, given an input, applies the second function then the first. Thus `((compose f g) 5)` is equivalent to `(f (g 5))`

Then make sure you can evaluate this expression:

`((compose (square-and-add 42) (divide-by 2)) 20)`. What do you get?

consider this

1. Draw box-and-pointer for the values of the following expressions. Also give the printed representation.

`(cons 1 2)`

`(cons 1 (cons 3 (cons 5 nil)))`

`(cons (cons (cons 3 2) (cons 1 0)) nil)`

`(cons 0 (list 1 2))`

`(list (cons 1 2) (list 4 5) 3)`

2. Write expressions whose values will print out like the following.

`(1 2 3)`

`(1 2 . 3)`

`((1 2) (3 4) (5 6))`

3. Write expressions using `car` and `cdr` that will return 4 when the name `lst` is bound to the following values:

```
(7 6 5 4 3 2 1)
```

```
((7) (6 5 4) (3 2) 1)
```

```
(7 (6 (5 (4 (3 (2 (1))))))))
```

```
(7 ((6 5 ((4)) 3) 2) 1)
```

Down for the Count

Write a procedure, `list-ref`, with type `List<A>`, `non-negative integer -> A`, which will return the Nth element of a list. Start counting from 0 like any good computer scientist.

```
(define list-ref
  (lambda (L n)
    ...
```

Copy cat

Give a list `L`, write a procedure `copy` which produces a new list with fresh new `cons` cells but contains the same elements. Then, evaluate:

```
(define L1 (list 1 5 (list 8 9) 'foo (quote bar)))
(eq? L1 (copy L1))
(eq? (copy L1) (copy L1))
(equal? L1 (copy L1))
```

Got it backwards

Write a procedure `reverse` which, given a list `L`, returns a new list where the elements appear in the reverse order. Thus:

```
(reverse '(1 2 3 4 5)) => (5 4 3 2 1)
(reverse (list (list 1 2) (list 3 4) 5)) => (5 (3 4) (1 2))
```

Now write a variant, `deep-reverse` which also reverses any sub-lists. For example:

```
(deep-reverse '(1 (2 3) (4 (5 6 (7 8) 9) 10) 11)) => (11 (10 (9 (8 7) 6 5) 4) (3 2) 1)
```

Boy, those nested parentheses are irritating, right? OK, write a procedure `flatten` which, given a list, removes all of the inner parentheses, producing a list with all elements of sublists as a single list, such as:

```
(flatten '(1 (2 3) (4 (5 6 (7 8) 9) 10) 11)) => (1 2 3 4 5 6 7 8 9 10 11)
```

A special snowflake

Create a procedure, `unique`, which given a list returns a new list where each element appears only once:

```
(unique '(1 2 2 3 4 5 4 8)) => (1 2 3 4 5 8)
```

And a variation: Instead, return a new list containing items that only appeared once in the original list.

```
(single-out '(1 2 2 3 4 5 4 8)) => (1 3 5 8)
```

To the max

Suppose you're given a list of numbers. Determine the largest number in the list. You can do this directly, but you can also do this using `fold-right`.

```
(max (list 9 123 2 -5 5.6 11 42)) => 123
```

Taking classes, abstractly

Suppose you and Ben Bitdiddle are working for the registrar, who has asked you to develop a Scheme system to keep track of each student's registration. Ben started working on this, but then got a call from Alyssa P. Hacker who really needs help with some sort of adventure game she's working on. Ben's only got an abstraction for a class so far:

```
(define (make-units C L H)    ;; Class, Lecture, Homework
  (list C L H))
(define get-units-C car)
(define get-units-L cadr)
(define get-units-H caddr)

(define (make-class number units)
  (list number units))
(define get-class-number car)
(define get-class-units cadr)
(define (get-class-total-units class)
  (let ((units (get-class-units class)))
    (+ (get-units-C units)
       (get-units-L units)
       (get-units-H units))))
(define (same-class? c1 c2)
  (= (get-class-number c1) (get-class-number c2)))
```

A schedule abstraction

The next thing that's needed is a way of representing a schedule, a set of classes.

1. Write a constructor that returns an empty schedule.

```
(define (empty-schedule)
```

2. Write a procedure that when given a class and a schedule, returns a new schedule including the new class:

```
(define (add-class class schedule)
```

3. Write a procedure that computes the total number of units in a schedule.

```
(define (total-scheduled-units sched)
```

4. Write a procedure that drops a particular class from a schedule and returns the new schedule

```
(define (drop-class sched classnum)
```

5. Implement the freshman credit limit by taking in a schedule, and removing classes until the total number of units is less than max-credits.

```
(define (credit-limit sched max-credits)
```

A student abstraction

Obviously you'll also need some students. From w20, Ben zephyrs you the following code:

```
(define (make-student number sched-checker)
  (list number (list) sched-checker))
(define get-student-number car)
(define get-student-schedule cadr)
(define get-student-checker caddr)

(define (update-student-schedule student schedule)
  (if ((get-student-checker student) schedule)
      (list (get-student-number student)
            schedule
            (get-student-checker student))
      "invalid schedule"))
```

6. Finish the call to `make-student` to limit the student to taking at least 1 class.

```
(make-student 575904467
```

7. Finish the call to `make-student` to create a first-term freshman (limited to 54 units).

```
(make-student 575904467
```

8. Write a procedure that takes a schedule and returns a list of the names of the classes in the schedule. Use `map`.

```
(define (class-names schedule)
  (map
```

A Higher power

1. Rewrite `drop-class` to use `filter`.
2. Rewrite `total-scheduled-units` to use `map` and `fold-right`.
3. Rewrite `credit-limit` to use `fold-right`.

Bonus

What does the following expression evaluate to?

```
( (lambda (x) (x x)) (lambda (x) (x x)) )
```