# Exam Prep 8: Scheme

Students from past semesters wanted more content and structured time to prepare for exams. Exam Prep sections are a way to solidify your understanding of the week's materials. The problems are typically designed to be a bridge between discussion/lab/homework difficulty and exam difficulty.

**Reminder:** There is nothing to turn in and there is no credit given for attending Exam Prep Sections.

We try to make these problems **exam level** , so you are not expected to be able to solve them coming straight from lecture without additional practice. To get the most out of Exam Prep, we recommend you **try these problems first on your own** before coming to the Exam Prep section, where we will explain how to solve these problems while giving tips and advice for the exam. Do not worry if you struggle with these problems, **it is okay to struggle while learning**.

You can work with anyone you want, including sharing solutions. We just ask you don't spoil the problems for anyone else in the class. Thanks!

You may only put code where there are underscores for the codewriting questions.

You can test your functions on their doctests by clicking the red test tube in the top right corner after clicking Run in 61A Code. Passing the doctests is not necessarily enough to get the problem fully correct. You must fully solve the stated problem.

## Reminder about eval

A very useful special form in scheme is eval, which when given a scheme list, evaluates it as if it were scheme source code.

```
scm> (eval (list 'cons 1 (list 'cons 2 '())))
(1 2)
scm> (eval '(+ 1 2))
3
scm> (define a 'b)
a
scm> (define b 'c)
b
scm> (define c 5)
c
scm> (eval 'a)
b
scm> (eval (eval 'a))
c
scm> (eval (eval (eval 'a)))
5
```

You will find understanding how eval works useful for the problems below.

# Q1: Fixing Scheme with Infix Notation

Adapted From Summer 2018 Final Q8 and Fall 2020 Examprep 8 Q3

**Important:** Scroll down for the function signatures, skeletons, and doctests for all parts.

**Difficulty:** ☆

**Part A:** First, write the helper function `skip`, which skips the first n items in a list, returning the rest. For full credit, your solution must be tail recursive. You may assume that `n` is non-negative.

```
scm> (skip 2 '(1 2 3 4))
(3 4)
scm> (skip 10 '(1 2 3 4))
()
```

**Difficulty:** ☆☆☆

**Part B:** NOne annoying thing about Scheme is that it can only understand arithmetic operations that are written inprefix notation. That is, if I want to evaluate an expression, the arithmetic operator must come first, which is different than in math class.

Let's leverage our skills to define a Scheme procedure `infix` that accepts arithmetic operations with infix notation, which places operators between operands as you are used to. You only need to support the addition and multiplication operators * and +, but you need to support order of operations.

**HINT:** You may find it useful to use `skip`, but it is not required!

```
scm> (infix '(1 + 2))
3
scm> (infix '(1 * 2))
2
scm> (infix '(3 + 2 * 5 + 4))
17
scm> (infix '(1 + 2 * (3 + 4)))
15
```

**Difficulty:** ⭐⭐

**Part C:** Update your infix scheme interpreter such that it works with names as well as numeric literals.

**HINT:** You will need to modify the given code!

```
scm> (define x 3)
scm> (infix '(x + 2))
5
scm> (infix '(1 * x))
3
scm> (infix '((x + x) * (x + x)))
36
```

**Note:** The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (skip n lst)
    (if (or (= n 0) (null? lst))
        lst
        (skip (- n 1) (cdr lst))
    )
)

(expect (skip 2 '(1 2 3 4)) (3 4))

(expect (skip 10 '(1 2 3 4)) ())

(define (infix expr)
    (cond
        ((not (pair? expr)) (eval expr))
        ((null? (cdr expr)) (infix (car expr)))
        (else
            (define left (infix (car expr)))
            (define right (infix (car (skip 2 expr))))
            (define operator (car (skip 1 expr)) )
            (cond
                ((equal? operator '+ )
                    (+ left (infix (skip 2 expr)))
                )
                ((equal? operator '*)
                    (infix (cons (* (eval left) (eval right))
                                 (skip 3 expr) )
                    )
                )
            )
        )
    )
)

(expect (infix '(1 + 2)) 3)

(expect (infix '(1 * 2)) 2)

(expect (infix '(3 + 2 * 5 + 4)) 17)
```

```
39
40  (expect (infix '(1 + 2 * (3 + 4))) 15)
41
42  (expect (infix '(1 + 2 * (3 + 4 * (5 + 6)))) 95)
43
44  (define x 3)
45
46  (expect (infix '(x + 2)) 5)
47
48  (expect (infix '(1 * x)) 3)
49
```

## Q2: Group by Non-Decreasing

**Difficulty:** ☆☆☆

Define a function `nondecreaselist`, which takes in a scheme list of numbers and outputs a list of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

**Hint:** You might want to review the nesting list structure in partition options from examprep 07 (./sol-examprep07#q2)

**Note:** The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
1   (define (nondecreaselist s)
2
3       (if (null? s)
4           nil
5           (let ((rest (nondecreaselist (cdr s)) ))
6               (if (or (null? (cdr s)) (> (car s) (car (cdr s))))
7                   (cons (list(car s)) rest)
8                   (cons (cons (car s) (car rest)) (cdr rest))
9               )
10          )
11      )
12  )
13
14  (expect (nondecreaselist '(1 2 3 1 2 3)) ((1 2 3) (1 2 3)))
```

```
15
16   (expect (nondecreaselist '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))
17           ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1)))
18
```

# Q3: Directions - Fall 2014 Final Q4(c)

**Difficulty:** ☆☆

Implement the Scheme procedure `directions`, which takes a number `n` and a symbol `sym` that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to `n` by repeatedly applying `car` and `cdr` to the nested list. Assume that `n` appears exactly once in the nested list bound to `sym`.

**Hint:** The implementation searches for the number `n` in the nested list `s` that is bound to `sym`. The returned expression is built during the search.

```
scm> (define a '(1 (2 3) ((4))))
scm> (directions 1 'a )
(car a)
scm> (directions 2 'a)
(car (car (cdr a)))
```

```
2     (define (search s exp)
3       ; Search an expression s for n and return an expression based on exp.
4       (cond
5         ((number? s)
6          (if (= s n ) exp nil) )
7         ((null? s)
8          nil)
9         (else
10         (search-list s exp))))
11    (define (search-list s exp)
12      ; Search a nested list s for n and return an expression based on exp.
13      (let ((first
14             (search (car s) (list 'car exp)) )
15            (rest
16            (search (cdr s) (list 'cdr exp)) ))
17        (if (null? first)
```

```
18              rest
19              first)))
20      (search (eval sym) sym))
21
22    (define a '(1 (2 3) ((4))))
23
24    (expect (directions 1 'a) (car a))
25
26    (expect (directions 2 'a) (car (car (cdr a))))
27
28    (expect (directions 4 'a)  (car (car (car (cdr (cdr a))))))
29
30    (define b '((3 4) 5))
31
32    (expect (directions 4 'b) (car (cdr (car b))))
33
```