# Homework 7: Scheme Lists

> Adapted from cs61a of UC Berkeley.

## Readings

You might find the following references useful:

- Scheme Specification
  - In the same folder with this handout.
- Scheme Built-in Procedure Reference
  - In the same folder with this handout.
- Section 3.2
- Section 3.5

## Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-hw07.git

```
git clone https://github.com/JacyCui/sicp-hw07.git
```

`hw07.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip hw07.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the homework. However, I might not give my solution exactly when the homework is posted. You need to finish the task on your own first. If any problems occurs, please make use of the comment section.

## Scheme Editor

### How to launch

In your `hw07` folder you will find a new editor. To run this editor, run `python3 editor`. This should pop up a window in your browser; if it does not, please navigate to localhost:31415 and you should see it.

Make sure to run `python3 ok` in a separate tab or window so that the editor keeps running.

## Features

The `hw07.scm` file should already be open. You can edit this file and then run `Run` to run the code and get an interactive terminal or `Test` to run the `ok` tests.

`Environments` will help you diagram your code, and `Debug` works with environments so you can see where you are in it. We encourage you to try out all these features.

`Reformat` is incredibly useful for determining whether you have parenthesis based bugs in your code. You should be able to see after formatting if your code looks weird where the issue is.

> By default, the interpreter uses Lisp-style formatting, where the parens are all put on the end of the last line
>
> ```
> (define (f x)
>     (if (> x 0)
>         x
>         (- x)))
> ```
>
> However, if you would prefer the close parens to be on their own lines as so
>
> ```
> (define (f x)
>     (if (> x 0)
>         x
>         (- x)
>     )
> )
> ```
>
> you can go to Settings and select the second option.

# Required Questions

## Scheme Lists

### Q1: Filter Lst

Write a procedure `filter-lst`, which takes a predicate `fn` and a list `lst`, and returns a new list containing only elements of the list that satisfy the predicate. The output should contain the elements in the same order that they appeared in the original list.

**Note:** Make sure that you are not just calling the built-in `filter` function in Scheme - we are asking you to re-implement this!

```
(define (filter-lst fn lst)
  'YOUR-CODE-HERE
)

;;; Tests
(define (even? x)
  (= (modulo x 2) 0))
(filter-lst even? '(0 1 1 2 3 5 8))
; expect (0 2 8)
```

Use Ok to unlock and test your code:

```
python3 ok -q filter_lst -u --local
python3 ok -q filter_lst --local
```

## Q2: Interleave

Implement the function interleave, which takes a two lists as arguments. interleave will return a new list that interleaves the elements of the two lists. Refer to the tests for sample input/output.

```
(define (interleave first second)
  'YOUR-CODE-HERE
)

(interleave (list 1 3 5) (list 2 4 6))
; expect (1 2 3 4 5 6)

(interleave (list 1 3 5) nil)
; expect (1 3 5)

(interleave (list 1 3 5) (list 2 4))
; expect (1 2 3 4 5)
```

Use Ok to test your code:

```
python3 ok -q interleave --local
```

## Q3: Accumulate

Fill in the definition for the procedure `accumulate`, which combines the first `n` natural numbers according to the following parameters:

1. `combiner`: a function of two arguments
2. `start`: a number with which to start combining

3. `n`: the number of natural numbers to combine
4. `term`: a function of one argument that computes the *n*th term of a sequence

For example, we can find the product of all the numbers from 1 to 5 by using the multiplication operator as the `combiner`, and starting our product at 1:

```
scm> (define (identity x) x)
scm> (accumulate * 1 5 identity)  ; 1 * 1 * 2 * 3 * 4 * 5
120
```

We can also find the sum of the squares of the same numbers by using the addition operator as the `combiner` and `square` as the `term`:

```
scm> (define (square x) (* x x))
scm> (accumulate + 0 5 square)  ; 0 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
55
scm> (accumulate + 5 5 square)  ; 5 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
60
```

You may assume that the `combiner` will always be commutative: i.e. the order of arguments do not matter.

```
(define (accumulate combiner start n term)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q accumulate --local
```

## Q4: No Repeats

Implement `no-repeats`, which takes a list of numbers `lst` as input and returns a list that has all of the unique elements of `lst` in the order that they first appear, but no repeats. For example, `(no-repeats (list 5 4 5 4 2 2))` evaluates to `(5 4 2)`.

*Hints*: To test if two numbers are equal, use the `=` procedure. To test if two numbers are not equal, use the `not` procedure in combination with `=`. You may find it helpful to use the `filter-lst` procedure with a helper `lambda` function to use as a filter.

```
(define (no-repeats lst)
  'YOUR-CODE-HERE
)
```

Use Ok to unlock and test your code:

```
python3 ok -q no_repeats -u --local
python3 ok -q no_repeats --local
```