Homework 08：Macros & Streams

1. Instructions

> Please download homework materials `hw08.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 3. The starter code for these problems is provided in `hw08.scm`, which is distributed as part of the homework materials in the `code` directory.

Submission: When you are done, submit your code to our Grader server as instructed in lab07 by `python submit.py --stuid <YOUR STUDENT ID> --stuname <YOUR NAME>`. You may submit more than once before the deadline; grader will record the highest score graded from your submissions. Check that you have successfully submitted your code and what your score is on Grader website.

> See lab07 for more instructions on on submitting assignments.

> **WARNING:** Do not modify `submit.py`!

Using Ok: If you have any questions about using Ok, please refer to this guide.

Readings: You might find the following references useful:

- Scheme Specification
- Scheme Built-in Procedure Reference

2. Scheme Editor

2.1 How to launch

In your `hw08` folder you will find a new editor. To run this editor, run `python 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 `python ok` in a separate tab or window so that the editor keeps running.

2.2 Features

The `hw08.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.

3. Required Problems

3.1 Macros

Problem 1: List Comprehensions (200 pts)

Recall that list comprehensions in Python allow us to create lists out of iterables:

```
[<map-expression> for <name> in <iterable> if <conditional-expression>]
```

Use a macro to implement list comprehensions in Scheme that can create lists out of lists. Specifically, we want a `list-of` macro that can be called as follows:

```
(list-of <map-expression> for <name> in <list> if <conditional-expression>)
```

Calling `list-of` will return a new list constructed by doing the following for each element in `<list>`:

- Bind `<name>` to the element.
- If `<conditional-expression>` evaluates to a truthy value, evaluate `<map-expression>` and add it to the result list.

Here are some examples:

```
scm> (list-of (* x x) for x in '(3 4 5) if (odd? x))
(9 25)
scm> (list-of 'hi for x in '(1 2 3 4 5 6) if (= (modulo x 3) 0))
(hi hi)
scm> (list-of (car e) for e in '((10) 11 (12) 13 (14 15)) if (list? e))
(10 12 14)
```

> *Hint*: You may use the builtin `map` and `filter` procedures. Check out the Scheme Built-in for more information.
> You may find it helpful to refer to the `for` loop macro introduced in lecture. The filter expression should be transformed using a lambda in a `similar` way to the map expression in the example.

```
(define-macro (list-of map-expr for var in lst if filter-expr)
  'YOUR-CODE-HERE
)
```

You can test your code from the terminal by running

```
python ok --local -q list-comp
```

> *Optional (not graded)*: Recall also that the `if <conditional>` portion of the Python list comprehension was optional. Modify your macro so that the Scheme list comprehension does not require a conditional expression.
> Refer to the macro form in the Scheme Specification for an explanation of how to do optional macro parameters.

3.2 Streams

Problem 2: Multiples of 3 (100 pts)

Define implicitly an infinite stream `multiples-of-three` that contains the multiples of 3.

You may use the `map-stream` function defined below. `map-stream` takes in a one-argument function `f` and a stream `s` and returns a new stream containing the elements of `s` with `f` applied.

```
(define (map-stream f s)
  (if (null? s)
    nil
    (cons-stream (f (car s)) (map-stream f (cdr-stream s)))))
```

> Do not define any other helper functions.

```
(define (map-stream f s)
    (if (null? s)
      nil
      (cons-stream (f (car s)) (map-stream f (cdr-stream s)))))

(define multiples-of-three
  'YOUR-CODE-HERE
)
```

You can test your code from the terminal by running

```
python ok --local -q mult3
```

Problem 3: Run-Length Encoding (100 pts)

Run-length encoding is a very simple data compression technique, whereby runs of data are compressed and stored as a single value. A *run* is defined to be a contiguous sequence of the same number. For example, in the (finite) sequence

```
1, 1, 1, 1, 1, 6, 6, 6, 6, 2, 5, 5, 5
```

there are four runs: one each of 1, 6, 2, and 5. We can represent the same sequence as a sequence of two-element lists:

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

Notice that the first element of each list is the number in a run, and the second element is the number of times that number appears in the run.

We will extend this idea to streams. Write a function called `rle` that takes in a stream of data, and returns a corresponding stream of two-element lists, which represents the run-length encoded version of the stream. You do not have to consider compressing infinite streams - the stream passed in will eventually terminate with `nil`.

```
scm> (define s (cons-stream 1 (cons-stream 1 (cons-stream 2 nil))))
s
scm> (define encoding (rle s))
encoding
scm> (car encoding)  ; Run of number 1 of length 2
(1 2)
scm> (car (cdr-stream encoding))  ; Run of number 2 of length 1
(2 1)
scm> (define s (list-to-stream '(1 1 2 2 2 3))) ; Makes a stream with the same
elements as the list passed in
scm> (stream-to-list (rle s))
((1 2) (2 3) (3 1))
```

```
(define (rle s)
  'YOUR-CODE-HERE
)
```

> You can define some helper functions in this problem.

You can test your code from the terminal by running

```
python ok --local -q rle
```

After all, remember to submit your answers by

```
python submit.py --stuid [YOUR STUDENT ID] --stuname [YOUR NAME]
```