
I WANT
YOU
To take your vitamins

Q5: (Tutorial) Interpreters Review

Discuss the follow questions with your tutorial group - they will be helpful for your understanding of the Scheme project! If you wish to take notes, we recommend you take notes on a separate document so it won't accidentally get erased.

What are the four parts of an interpreter (Hint: what does REPL stand for)? What does each part do? What parts did you work on implementing in the discussion?

For the Calculator interpreter implemented in discussion, for the following executed code, what would be the input into the "Read" portion of the interpreter?

```
scm> (cons '+ (cons 2 (cons 3 nil)))
(+ 2 3)
```

What would be the output of the "Read" portion for the same code?

How does the evaluate stage work in Calculator? How do we know if an input into `calc_eval` is a call expression?

Q6: (Tutorial) Replicate

Write a function that takes an element `x` and a non-negative integer `n`, and returns a list with `x` repeated `n` times.

Tip: If you aren't sure where to start, try writing the corresponding recursive function for Linked Lists in Python first!

```
(define (replicate x n)
    'YOUR-CODE-HERE









)




;;; Tests
(replicate 5 3)
; expect (5 5 5)
```

Q7: (Tutorial) Run Length Encoding

A **run-length encoding** is a method of compressing a sequence of letters. The list `(a a a b a a a a)` can be compressed to `((a 3) (b 1) (a 4))`, where the compressed version of the sequence keeps track of how many letters appear consecutively.
Write a function that takes a compressed sequence and expands it into the original sequence.

Hint: You may want to use `my-append` and `replicate`.

```
scm> (my-append '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

```
(define (my-append a b)
(if (null? a)
b
(cons (car a) (my-append (cdr a) b))))
```

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

;;; Tests
(uncompress '((a 1) (b 2) (c 3)))
; expect (a b b c c c)
```

Q8: (Tutorial) Map

Write a function that takes a procedure and applies it to every element in a given list.

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









)

;;; Tests
(map (lambda (x) (* x x)) '(1 2 3))
; expect (1 4 9)
```

Q10: (Tutorial) Tree Sum

Using the abstract data type from problem 9, write a function that sums up the entries of a tree, assuming that the entries are all numbers.

Hint: you may want to use the `map` function you defined in problem 8, and also write a helper function for summing up the entries of a list.

```
(define (tree-sum tree)
'YOUR-CODE-HERE









)

'YOUR-CODE-HERE
```