PRACTICE PROBLEMS

warmup (Q5)

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.

This section should be fairly open-ended and you can choose to spend as much or as little time on this as you wish / as your students feel comfortable with

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?

Your Answer:

SOL:

The four parts of an interpreter are read, evaluate, print and loop.

1. Read takes in the user input code and outputs some data structure representing the expression.
2. Evaluate takes in this data structure and outputs a value representing what this expression evaluates to. In this step, call expressions, arithmetic expressions, special forms, etc. will all turn into simple values / primitives.
3. Print takes in however the value returned from the evaluate stage is represented and displays a human-readable form to the user interacting with the interpreter.
4. Loop simply means loop! We go back to the read stage to execute the next user-inputted line of code.

In this discussion, we worked on Read and Evaluate. In the scheme project you will also be implementing the Read and Evaluate stages of the Scheme Interpreter.

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)
```

Read → step 1 → scheme/calculator

Your Answer:

SOL:

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

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

Read outputs data structure representing expression => using pair class!

Your Answer:

SOL:

```
r('+', Pair(2, Pair(3, nil)))
```

loes the evaluate stage work in Calculator? How do we know if an input into `calc_eval` all expression?

nswer:

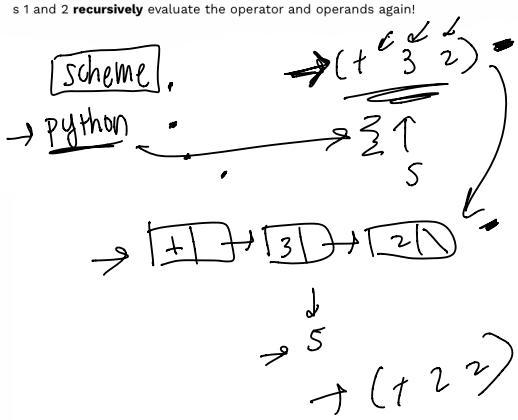
SOL:

e evaluate stage, Pairs are inputted and passed into `calc_eval` to be turned into es. We know the input is a call expression if it is a Pair. If it's an operator (such as `+`, tc), we return the corresponding operator. Else, we know it's a primitive, and so we n itself, since primitives evaluate to themselves.

importantly, the evaluate stage is recursive -- in particular, remember the rules of uation:

Evaluate the operator, which evaluates to a function.
Evaluate the operands from left to right.
Apply the function to the value of the operands.

s 1 and 2 **recursively** evaluate the operator and operands again!



Replicate (Q6)

Q6: (Tutorial) Replicate

input → takes an element `x` and a non-negative integer `n`, and → output 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!

Your Answer

```
1   (define (replicate x n)
2       'YOUR-CODE-HERE
3   )
4
5   ;;; Tests
6   (replicate 5 3)
7   ; expect (5 5 5)
8
```

repeat x=5 3 times in new list →

```
scm> (replicate 5 3)
(5 5 5)
```

Hints
1. think recursively!
2. How do we make new lists in scheme? using cons

Link in chat! ↳please write your solution/ideas there!!

4:26

work time: 5 min

SOLUTION:

```
(define (replicate x n)
    (if (= n 0)        ← base case, if n=0 that means we want to return an empty list (do not need to repeat x any more times)
        nil            ← return empty list (base case)
        (cons x (replicate x (- n 1)))   ← create a new list: add x to the list one time, recursively call replicate to repeat x n-1 more times
    )
)
```

recursive call (reduce n by 1 because we have added x to our list one time "manually")

Run Length Encoding (Q7)

(cons ↓ 1st)

Link(3, →)

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 (input) and expands it into the original sequence. (output)

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

`my-append` is implemented as follows, where `my-append` takes in two lists and concatenates them together.

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

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

Your Answer

```
1   (define (uncompress s)
2       'YOUR-CODE-HERE
3   )
4
5   ;;; Tests
6   (uncompress '((a 1) (b 2) (c 3)))
7   ; expect (a b b c c c)
8
```

(null? s)

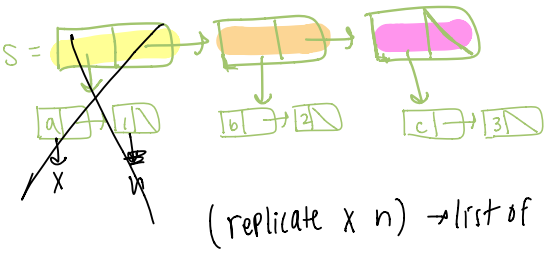
decompress one letter, recurse on rest

Hints:
- start with base case!
- think recursively
  - how can my append & replicate be helpful?
- drawing out input to doctest may be helpful!

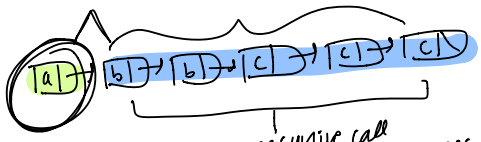
Link in chat!
↳ please write your solution/ideas there!!

work time: 7 min

Remember s = '((a 1) (b 2) (c 3)) ← input to our function



4:40



recursive call to uncompress

(replicate x n) → list of x repeated n times

( ) lnk = Link.empty
lnk.rest

SOLUTION:

```
(define (uncompress s)
  (if (null? s)        ← very common base case! If the list is empty, we have nothing to compress so just return the empty list
      s                ← could also just be nil (here s is nil)
      (my-append (replicate (car (cars)) (car (cdr (car s)))) (uncompress (cdr s)))
  )
)
```

- (my-append ...): essentially concatenating the two parts together into a single list (orange decompresses first letter & navy decompresses rest of letters)
- (car (cars)): gets the letter
- (car (cdr (car s))): gets the # of times we want to repeat that letter
- (replicate ...): returns a list that repeats letter appropriate # of times (first letter is uncompressed)
- (uncompress (cdr s)): recursive call on rest of letters to decompress them as well!

Map (Q8)

Q8: (Tutorial) Map

input

what function needs to do

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

```
1   (define (map fn lst)
2       'YOUR-CODE-HERE
3   )
4
5   ;;; Tests
6   (map (lambda (x) (* x x)) '(1 2 3))
7   ; expect (1 4 9)
8
```

← Apply fn to every element in lst

1 2 3

(1)(1)=1 (2)(2)=4 (3)(3)=9

Link in chat!
↳please write your solution/ideas there!!

4:53

work time: 5 min

SOLUTION:

```
(define (map fn lst)
    (if (null? lst)      → same base case again! If lst is empty then we have nothing to apply function to
        nil              → return empty list (could also be lst)
        (cons (fn (car lst)) (map fn (cdr lst)))
    )
)
```

- cons: create a new list
- (fn (car lst)): "manually" apply function to first element of list
- (map fn (cdr lst)): recursively call map on rest of list

Tree Sum (Q10)

We will assume we already have implementation for tree ADT (Q9).

Q9: Make Tree

Solution

creates new tree

returns label of root of tree

returns branches of tree

```
(define (make-tree label branches) (cons label branches))
(define (label tree)
    (car tree)
)
(define (branches tree)
    (cdr tree)
)
```

tree(_,[])

Q10: (Tutorial) Tree Sum

Using the abstract data type above, 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 above, and also write a helper function for summing up the entries of a list.

Your Answer

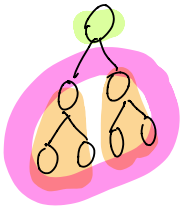
input

```
1  (define (tree-sum tree)
2      'YOUR-CODE-HERE
3  )
4
5  'YOUR-CODE-HERE
6
7
```

Link in chat!
↳please write your solution/ideas there!!

work time: 6 min

(t list)



= [8,9]

for b in branches:
tree-sum

(branches tree) = list of trees

SOLUTION:

tree

```
(define (tree-sum tree)
    ( + (label tree) (sum (map tree-sum (branches tree))))
)
```

need this helper function since (branches tree) is a list so map will find the total sum of each of the branches but we also need to sum up the sums of the branches!

get the root's label

get the sum of all the other labels in the tree ("recursive call")

```
(define (sum lst)
    (if (null? lst)
        0
        (+ (car lst) (sum (cdr lst)))
    )
)
```

helper function (sum of a list)

if the list is empty, have nothing to add (sum=0)

add first value with rest to get total

recursive call for sum of rest of list