FINAL REVIEW PART 2: SCHEME AND REGEX Solutions

COMPUTER SCIENCE MENTORS

April 26 - 29, 2021

1 Examtool Practice Test

If you're looking for a practice test, CSM created one with problems from a combination of previous 61A Finals and our own problem bank. Check it out here!

Exam Password: `o3xquVYr19BzoHiVcUx-rEbIb9Sq8NwXrq4NfW8u-4A`

Here are the PDF solutions. We highly recommend that you attempt all the problems before looking at the solutions.

2 Scheme Lists

1. The **map** function takes in a two-argument function and a list of elements, and applies that function to each element in that list. We want to define our own version of the **map** function EXCEPT instead of applying a function to a list of elements, we want to pass in a single element and apply each function in a list of functions to that element.

   Define a function `reverse-map`, which takes in a list of functions, `operators`, and a single argument, `arg`, and returns a list that results from applying all of the functions in `operators` on `arg`. You may assume that all functions in `operators` will work properly with the single input `arg`.

```
; doctests
scm> (define funcs (list (lambda (x) (- x 10)) (lambda (x) (*
   x 2)) (lambda (x) (integer? x))))
funcs
scm> (reverse-map funcs 2)
(-8 4 #t)
scm> (reverse-map funcs 16)
(6 32 #t)

(define (reverse-map operators args)

    (if (________________________________)

        ________________________________________________ \

        ________________________________________________)
)

(define (reverse-map operators arg)
    (if (null? operators)
        nil
        (cons ((car operators) arg)
            (reverse-map (cdr operators) arg)))
)
```

2. Fill in `backwards-sum` such that it takes in a list of numbers `lst` and returns a new list with each element being the sum of itself and all elements to the right of it in `lst`.

   Sidebar: the word ”sum” being bolded has no significance, it is an auto-formatting issue.

```
; doctests
scm> (backwards-sum '(1 2 3 4))
(10 9 7 4)
scm> (backwards-sum '(2 -1 3 7))
(11 9 10 7)

(define (backwards-sum lst)

)
```

```
(define (backwards-sum lst)
    (cond
        ((null? lst) nil)
        ((null? (cdr lst)) (list (car lst)))
        (else (cons
            (+ (car lst) (car (backwards-sum (cdr lst))))
            (backwards-sum (cdr lst))))
    )
)
```

3. Define `well-formed`, which determines whether `lst` is a well-formed list or not. Assume that `lst` only contains numbers and no nested lists.

```
; Doctests
scm> (well-formed '())
#t
scm> (well-formed '(1 2 3))
#t
; List doesn't end in nil
scm> (well-formed (cons 1 2))
#f

(define (well-formed lst)





)
```

```
; well-formed with nested if statements
(define (well-formed lst)
    (if (null? lst)
        #t
        (if (number? lst)
            #f
            (well-formed (cdr lst)))))

; well-formed with a cond statement
(define (well-formed lst)
    (cond ((null? lst) #t)
        ((number? lst) #f)
        (else (well-formed (cdr lst)))))
```

3 Tail Recursion

4. Implement `slice`, which takes in a a list `lst`, a starting index `i`, and an ending index `j`, and returns a new list containing the elements of `lst` from index `i` to `j - 1`.

```
;Doctests
scm> (slice '(0 1 2 3 4) 1 3)
(1 2)
scm> (slice '(0 1 2 3 4) 3 5)
(3 4)
scm> (slice '(0 1 2 3 4) 3 1)
()

(define (slice lst i j)








)
```

```
(define (slice lst i j)
      (cond ((or (null? lst) (>= i j)) nil)
               ((= i 0) (cons (car lst) (slice (cdr lst) i (-
                  j 1))))
               (else (slice (cdr lst) (- i 1) (- j 1)))))
```

5. Now implement slice with the same specifications, but make you implementation tail recurisve.
   You may wish to use the built-in append function, which takes in two lists and returns a new list containing the elements of the two lists concatenated together.

```
(define (slice lst i j)

)
```

```
(define (slice lst i j)
  (define (slice-tail lst i j lst-so-far)
      (cond ((or (null? lst) (>= i j)) lst-so-far)
            ((= i 0) (slice-tail (cdr lst) i (- j 1) (
               append lst-so-far (list (car lst)))))
            (else (slice-tail (cdr lst) (- i 1) (- j 1) lst
               -so-far))))
  (slice-tail lst i j nil))
```

Alternate Solution:

```
(define (slice lst i j)
  (define (slice-tail lst index lst-so-far)
      (cond ((or (null? lst) (= index j)) lst-so-far)
            ((<= i index) (slice-tail (cdr lst) (+ index 1)
                (append lst-so-far (list (car lst)))))
            (else (slice-tail (cdr lst) (+ index 1) lst-so-
               far))))
  (if (< i j) (slice-tail lst 0 nil) nil))
```

4 Macros

6. Write a macro, **and**-odds, which takes in a list of expressions, exprs, and evaluates to a true value if all of the even-indexed elements of exprs evaluate to true values. If any of the even-indexed elements evaluate to false, **and**-odds should return false.

```
; doctests
scm> (and-odds '((= 10 10)))
#t
scm> (and-odds '((= 1 2)))
#f
scm> (and-odds '(#f #t #t))
#f
scm> (and-odds '((< 5 3) (= 5 5)))
#f
scm> (and-odds '((> 3 2) (< 5 0) (= 5 5)))
#t
scm> (and-odds '((< 1 5) (< 5 2) (< 3 5) (< 5 3) (< 4 5)))
#t
scm> (define a (list 1 #f 3))
a
scm> (and-odds a)
3

(define-macro (and-odds exprs)

    `(if ________________________________________

         ________________________________________

         ________________________________________
    )
)

(define-macro (and-odds exprs)
    `(if (> (length ,exprs) 2)
         (and (car ,exprs) (and-odds (cdr (cdr ,exprs))))
         (eval (car ,exprs))
    )
)
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

7. Define a macro, `eval-and-check` that takes in three expressions and evaluates each expression in order. If the last expression evaluates to a truth-y value, return the symbol ok. Otherwise, return fail.

```
;Doctests
scm> (eval-and-check #f #f #t)
ok
scm> (eval-and-check (+ 2 3) (print 2) (> 2 3))
2
fail
scm> (eval-and-check (define x 1) (print x) (> x 0))
1
ok
```

```
(define-macro (eval-and-check expr1 expr2 expr3)

    ________________________________________________

    ________________________________________________

    ________________________________________________)
```

```
(define-macro (eval-and-check expr1 expr2 expr3)
      `(if (begin ,expr1 ,expr2 ,expr3)
           'ok
           'fail))
```

8. Now expand `eval-and-check` to take in any number of expressions (as long as there is at least one).

```
;Doctests
scm> (eval-and-check #f #f #f #f #t)
ok
scm> (eval-and-check (print 2) (> 2 3))
2
fail
```

```
(define-macro (eval-and-check expr1 . args)

     ________________________________________

     ________________________________________

     ________________________________________)
```

```
(define-macro (eval-and-check expr1 . args)
        `(if ,(cons 'begin (cons expr1 args))
                'ok
                'fail))
```

9. Write a macro, `zero-cond` that takes in a list of condition-value pairs where each pair contains only numbers or arithmetic expressions that evaluate to numbers. It should evaluate each condition, *treating expressions that evaluate to 0 as false-y* and then return the value corresponding to the first truth-y value.

```
;Doctests
scm> (zero-cond
              ((0 'wrong)
               ((- 1 1) 'wrong)
               ((* 1 1) 'correct!)
               (2 'wrong)))
correct!
```

```
(define-macro (zero-cond conditions)
```

```
(define-macro (zero-cond conditions)
     (cons 'cond
              (map (lambda (pair) (cons (not (= 0 (eval (car
```

```
                         pair)))) (cdr pair)))
                                  conditions)))
```

5 Regex Practice

Here's a Regex reference sheet courtesy of Data100 Course Staff. Check it out here!

10. We are given a linear equation of the form $mx + b$, and we want to extract the m and b values. Remember that '.' and '+' are special meta-characters in Regex.

```
import re
def linear_functions(eq_str):
    """
    Given the equation in the form of 'mx + b', returns a
       tuple of m and b values.
    >>> linear_functions("1x+0")
    [('1', '0')]
    >>> linear_functions("100y+44")
    [('100', '44')]
    >>> linear_functions("99.9z+23")
    [('99.9', '23')]
    >>> linear_functions("55t+0.4")
    [('55', '0.4')]
    """
    return re.findall(r"__________", eq_str)

        r'(\d*\.?\d+)\w\+?(\d*\.?\d+)'
```