CS 440 Fall 2022

Midterm Exam

Question Booklet

Instructions:

- This exam is closed-book, closed-notes. Electronic devices of any kind are not permitted.
- You will write all your responses in the provided answer booklet. Please write legibly in the space provided for each exercise and, if necessary, clearly indicate your final answer.
- You may use the question booklet as scratch paper, but we will only score your answer booklet.
- Turn in both the exam question and answer booklets.

Concepts (20 points):

1. Consider the following list constructed using `cons`.

```
(cons (cons 2 3)
      (cons 4
            (cons (cons 5 '())
                  '())))
```

   Which of the following quoted list forms is equivalent to the above?

   a. `'(2 3 4 5)`
   b. `'((2 . 3) 4 (5))`
   c. `'((2 . 3) (4 5))`
   d. `'((2 3) (4 (5)))`

2. Consider the following definitions and subsequent list constructed using `cons`.

```
(define a "hello")
(define lst '(1 2 3))

(cons 'a (cons a (cons 'b lst)))
```

   Which of the following quasiquoted list forms is equivalent to the above?

   a. `` `(a ,a b ,lst) ``
   b. `` `(,a a ,b lst) ``
   c. `` `(a ,a b ,@lst) ``
   d. `` `(a ,@a b ,lst) ``

3. Why might it be a good idea to use tail-recursion, if possible, instead of "regular" recursion?

   a. optimized tail-calls do not require allocating stack frames
   b. tail-recursion will typically reduce the runtime complexity of the implemented algorithm
   c. tail-call optimization eliminates the need for multiple (tree) recursion
   d. tail-recursion is only more efficient when used to implement structural recursion (e.g., to process lists)

4. How do Racket syntax transformer functions (aka a *macro*) differ most significantly from "regular" Racket functions?

   a. macros are carried out at compile-time rather than run-time
   b. macros are unable to manipulate lists using the standard built-in functions
   c. macros rely entirely on pattern matching constructs (e.g., `match`) to work
   d. macros make use of dynamic scoping rather than lexical scoping (the default)

5. Consider the following function definition, which makes use of pattern matching:

```
(define (pat x)
  (match x
    [(list 'x n) (add1 n)]
    [(list _ x y) (* x y)]
    [(list a b) (+ a b)]
    [_ (first x)]))
```

   What does the expression `(pat '(foo 2 3))` evaluate to?

   a. 2
   b. 4
   c. 5
   d. 6

6. Which of the following lists, in the correct order, the typical stages carried out by a programming language interpreter?

   a. parsing, compilation, analysis/optimization, evaluation
   b. parsing, evaluation, desugaring, analysis/optimization
   c. desugaring, parsing, evalaution, analysis/optimization
   d. scanning, parsing, analysis/optimization, evaluation

7. What is the primary purpose of the "environment" structure, as maintained by the `eval` function of our interpreter?

   a. to keep track of all un-evaluated expressions encountered so far
   b. to store variable bindings in the current scope
   c. to hold syntax decorations generated by the parser
   d. to store global function definitions

8. What is one of the potential advantages of using a *lazy* evaluation strategy?

   a. tail-call optimization may be performed on tail-recursive functions
   b. closures may correctly access their lexical environments
   c. expressions used as function arguments may be unevaluated if their results aren't needed
   d. expressions are evaluated just once before their results are passed to functions

9. Assuming *dynamic binding* is in effect, what would the following expression evaluate to?

```
(let ([f (let ([s "hello"])
           (lambda (t)
             (string-append s t)))])
  (let ([s "hola"]
        [t " amigo!"])
    (f " friend!")))
```

   a. `"hello amigo!"`

   b. `"hola amigo!"`

   c. `"hello friend!"`

   d. `"hola friend!"`

10. An implementation of a *closure* essentially couples:

   a. a name with a function definition (lambda)

   b. a function with its lexical environment

   c. an identifier with its binding in the current scope

   d. a function's formal parameters with its arguments

Recursion (16 points):

1. Consider the following recursive function:

```
(define (fn-a n x)
  (if (= n 0)
      '()
      (cons x
            (cons n
                  (fn-a (sub1 n) x)))))
```

   A. What does `(fn-a 5 'a)` evaluate to? Give your answer in quoted list form (i.e., without using `cons`).

   B. In the answer booklet you will find a partial implementation of a tail-recursive version of `fn-a`. Fill in the blank to complete the implementation correctly.

2. Consider the following recursive function:

```
(define (fn-b lst acc)
  (cond [(empty? lst) acc]
        [(not (list? (first lst)))
         (fn-b (rest lst)
               (cons (first lst) acc))]
        [else
         (fn-b (rest lst)
               (fn-b (first lst) acc))]))
```

   A. What does `(fn-b '(1 (a (b) c) 3) '())` evaluate to?

   B. Is `fn-b` tail-recursive? Why or why not?

HOFs (16 points):

Match each of the expressions on the left, which make use of HOFs, with its corresponding result on the right. Some of the results are not used. In the answer booklet, write the letter (one of A-N) designating your chosen result in the blank corresponding to the numbered expression.

```
(define lst1 (range 10)) ; i.e., 0 1 2 ... 9

(define lst2 '("i" "am" "the" "best" "coder"))

;; 1
(map (compose add1 string-length)
     lst2)

;; 2
(map (curry (lambda (x y)
              (+ (* 2 y) x))
            5)
     lst1)

;; 3
(filter (compose even? (curry * 2))
        lst1)

;; 4
(filter (compose (curry < 2)
                 string-length)
        lst2)

;; 5
(foldl (lambda (x r)
          (if (even? x)
              (cons x r)
              r))
       '()
       lst1)

;; 6
(foldl (lambda (n r)
          (append (range n) r))
       '()
       (filter odd?
               (map string-length lst2)))

;; 7
(foldr (lambda (x r)
         (cons (+ x 10) r))
       '()
       lst1)

;; 8
(foldr (lambda (s r)
         (cons (string-length s) r))
       (filter even? lst1)
       lst2)
```

```
;; A
'(0 1 2 3 4 5 6 7 8 9)

;; B
'(1 2 3 4 5 0 2 4 6 8)

;; C
'(9 8 7 6 5 4 3 2 1 0)

;; D
'(1 2 3 4 5)

;; E
'(0 2 4 6 8)

;; F
'(2 3 4 5 6)

;; G
'(8 6 4 2 0)

;; H
'(10 11 12 13 14 15 16 17 18 19)

;; I
'(0 1 2 3 4 0 1 2 0)

;; J
'(0 0 1 2 0 1 2 3 4)

;; K
'(2 4 6 "best" "coder")

;; L
'(5 7 9 11 13 15 17 19 21 23)

;; M
'("the" "best" "coder")

;; N
'("am" "i")
```

Interpreters (16 points):

Consider the following definitions, which implement an interpreter for a toy language.

```
(struct deposit-cmd (slot val))

(struct swap-cmd ())

(struct add-cmd (val))

(define (parse lst)
  (map (lambda (sexp)
         (match sexp
           [(list 'dep i x) (deposit-cmd i x)]
           [(list 'swp)     (swap-cmd)       ]
           [(list 'add x)   (add-cmd x)      ]))
       lst))

(define (eval prog)
  (let eval-env ([prog prog]
                 [reg0 0]
                 [reg1 0])
    (if (empty? prog)
        (list reg0 reg1)
        (match (first prog)
          [(deposit-cmd 0 x) (eval-env (rest prog) x reg1)         ]
          [(deposit-cmd 1 x) (eval-env (rest prog) reg0 x)         ]
          [(add-cmd x)       (eval-env (rest prog) (+ reg0 x) reg1)]
          [(swap-cmd)        (eval-env (rest prog) reg1 reg0)      ])))))
```

For each of the following programs in our toy language, you will determine the result computed by the interpreter above. Given a program `PROG`, assume the interpreter is invoked thusly: `(eval (parse PROG))`.

1. `PROG = '()`
2. `PROG = '((dep 0 5) (dep 1 10))`
3. `PROG = '((dep 1 44) (swp))`
4. `PROG = '((dep 0 5) (swp) (add 10) (swp) (add 44))`