CS2135, A-01

Final Exam

Name:

Problem	Points	Score
1	15	
2	25	
3	30	
4	30	
	Total	

You have 50 minutes to complete the problems on the following pages. There should be sufficient space provided for your answers. You do not need to show templates, but you may receive partial credit if you do. You also do not need to show test cases or examples of data models, but you may develop them if they will help you write the programs.

Your programs may contain only the following Scheme syntax:

define define-struct cond else lambda local begin set!

and the following primitive operations:

empty? cons? cons first rest car cdr list map filter
number? + − ∗ / = < > <= >= *zero?*
symbol? symbol=? equal?
boolean? **and or** *not*

and the functions introduced by **define-struct**.

You may, of course, use whatever constants are necessary.

1. (15 points) Consider the following definition of a class for squares:

```
;; make-square : number → square-object
;; takes the length of a side of the square as input
(define make-square
  (lambda (initlen)
    (local [(define sidelen initlen)]
      (lambda (service)
        (cond [(symbol=? service 'get-sidelen) sidelen]
              [(symbol=? service 'area)
               (lambda () (* sidelen sidelen))]
              [(symbol=? service 'resize)
               (lambda (newlen) (set! sidelen newlen))])))))
```

Add a service *smaller-than?* to the square class that compares the square to another (second) square and returns a boolean (true if the square is smaller than the second square and false otherwise). For full credit,

- add the service without editing the given square class, and
- reuse the existing *get-sidelen* service in the existing class (without copying the existing code)

[If you can not do the problem under these restrictions, you may do it by editing or copying the above code for half credit.]

2. (25 points) Convert the following program into CPS. If the converted version is similar to the original, you may either write your edits directly on the original function text (without recopying it), or cross-out and edit individual lines. You do **not** need to copy the documentation.

   Assume that *printf*, *length*, *filter* and the usual primitives are built-in. You may also assume that *prompt-read-many/k* exists (ie, you do **not** need to provide a definition for prompt-read-many/k).

```
;; display-overdue : symbol list['y or 'n] → void
;; prints a message saying how many 'n are in the project status list
(define display-overdue
  (lambda (name proj-status)
    (printf "~a is behind on ~a projects ~n" name (count-overdue proj-status))))

;; count-overdue : list['y or 'n] → number
;; counts how many times 'n is in the input list
(define count-overdue
  (lambda (anslist)
    (length (filter (lambda (ans) (symbol=? ans 'n)) anslist))))

;; count-missing-projs : symbol symbol → void
;; prompts for info on which projects student should have completed based on class-year
(define count-missing-projs
  (lambda (name class-year)
    (display-overdue
     name (cond [(symbol=? class-year 'senior)
                 (prompt-read-many (list "Suff Done? [y/n]" "IQP Done? [y/n]" "MQP Done? [y/n]"))]
                [(symbol=? class-year 'junior)
                 (prompt-read-many (list "Suff Done? [y/n]" "IQP Done? [y/n]"))]
                [(symbol=? class-year 'sophomore) (prompt-read-many (list "Suff Done? [y/n]"))]
                [else empty]))))
```

3. (30 points) Missing familiar-looking loops, a friend proposed the following for-loop for Scheme:

```
;; for-loop : number function → void
;; takes a number and a function and runs the function the indicated number of times
(define for-loop
  (local [(define index 1)]
    (lambda (high-num f)
      (cond [(> index high-num) (set! index 1)]
            [else (begin
                    (f index)
                    (set! index (add1 index))
                    (for-loop high-num f))]))))
```

(a) (5 points) Your friend wants to use *for-loop* to write a function *print-to-n* that consumes a number n and prints out the numbers from 0 to n. For example, *print-to-n* should behave as follows:

```
> (print-to-n 3)
0 1 2 3
```

Indicate what the following proposed definition of *print-to-n* produces and explain why it does or does not implement the desired behavior.

```
(define (print-to-n n)
  (local ([define index 0])
    (for-loop n (lambda (num) (printf "~a ~n" index)))))
```

(b) (10 points) Is your friend's use of *set*! justified in the *for-loop* code? Why or why not?

(c) (15 points) Write a version of *for-loop* that does not use **set!**. You may write a helper function or change the contract of *for-loop*, but your new *for-loop* function must still take the function to run in the loop as a parameter.

4. (30 points) Let's evolve our current Curly language to provide constructs for monitoring program behavior. Specifically, we want to add constructs for

   - determining how many function calls have been made (*call-count*)
   - resetting the function call count (*reset-count*)

   Call-count will return the number of function calls that have been made since the count was last reset. Reset-count will reset the count and return 0. For example, here is how these expressions should evaluate:

```
> (eval (make-call-count))
0
> (eval (make-plus (make-apply (make-proc 'x (make-var 'x)) 6) 4))
10
> (eval (make-call-count))
1
> (eval (make-plus (make-apply (make-proc 'x (make-var 'x)) 6) 7))
13
> (eval (make-call-count))
2
> (eval (make-plus (make-call-count) 5))
7
> (eval (make-reset-count))
0
> (eval (make-call-count))
0
```

   The revised data definition and new **define-struct**s are as follows:

```
An expr is one of
  - a number,
  - (make-var symbol)
  - (make-proc symbol expr)          (define-struct call-count ())
  - (make-plus expr expr)            (define-struct reset-count ())
  - (make-apply expr expr)
  - (make-call-count)
  - (make-reset-count)
```

   Our previous interpreter appears **on the next page** (with clauses for the new cases). Fill in the cases for *call-count?* and *reset-count?* and make any other edits necessary to implement the behavior shown in the above examples. Your solution may use global variables. **Do not copy the existing code – simply show where any edits should go, or cross-out and rewrite individual lines that must change.**

```
;; eval : expr → value
;; evaluates the expression and returns its answer
(define (eval an-exp)
  (cond [(number? an-exp) an-exp]
        [(var? an-exp) (error 'eval (format "Unbound variable ~a" (var-name an-exp)))]
        [(proc? an-exp) an-exp]
        [(plus? an-exp) (+ (eval (plus-left an-exp))
                           (eval (plus-right an-exp)))]
        [(apply? an-exp)
         (local [(define aproc (eval (apply-func an-exp)))]
           (eval (subst (proc-body aproc)
                        (proc-param aproc)
                        (eval (apply-arg an-exp)))))]
        [(call-count? an-exp) ... ]
        [(reset-count? an-exp) ... ]))
```