CS2135, C02

Final Exam

Name:

Problem	Points	Score
1	20	
2	20	
3	25	
4	35	
	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. (20 points) Consider the following program:

```
;; majority? : symbol list[symbol] → boolean
;; determine whether more than half of the items in the list match the given item
(define (majority? item alos)
  (> (length (filter (lambda (anitem) (symbol=? anitem item)) alos))
     (/ (length alos) 2)))
(and (majority? 'a (list 'a 'b 'a))
     (majority? 'c (list 'z 'z 'z)))
```

   (a) (15 points) Write down all of the closures that get created when running this program (including their environments). [Do not copy the function bodies in the closures – (lambda (anitem) ...) is enough]

   (b) (5 points) What are the contents of the environment when (*symbol=? anitem item*) is evaluated the second time?

2. (20 points) You are developing the code for an on-line video store. Since customers often look for multiple movies at once, the store wants to allow users to add to their orders from multiple pages. Here is an example of the desired interaction:

```
> (define p1 (videostore ...))
> ((p1 'add-title) "Pulp Fiction")
Current order is ("Pulp Fiction")
> ((p1 'add-title) "Star Wars")
Current order is ("Star Wars" "Pulp Fiction")
> (define p2 (p1 'duplicate-page))
> ((p2 'add-title) "Godfather")
Current order is ("Godfather" "Star Wars" "Pulp Fiction")
> ((p1 'add-title) "Peter Pan")
Current order is ("Peter Pan" "Godfather" "Star Wars" "Pulp Fiction")
> (p1 'finish-order)
You've ordered ("Peter Pan" "Godfather" "Star Wars" "Pulp Fiction")
> ((p1 'add-title) "Ben Hur")
Current order is ("Ben Hur")
```

The following code fragment is a start at implementing videostore. Finish the code so that it would produce the interaction shown above. This entails:

- Defining the order variable
- Deciding on the inputs to videostore
- Implementing the duplicate-page service

Do not add global variables. Mark your edits on the code fragment (do not rewrite the code).

```
(define videostore


  (lambda (service)
    (cond [(symbol=? service 'add-title)
           (lambda (title)
             (begin
               (set! order (cons title order))
               (printf "Current order is ~s ~n" order)))]
          [(symbol=? service 'finish-order)
           (begin
             (printf "You've ordered ~s ~n" order)
             (set! order empty))]
          [(symbol=? service 'duplicate-page) ...])))
```

3. (25 points) The following code provides the core of a mastermind program that you want to release on the web. To do this, you would need to (a) convert *prompt-guess/script* to *prompt-guess/web* (b) move all calls to *prompt-guess/script* into script position, replacing calls to *prompt-guess/script* with calls to *prompt-guess/web*.

```
;; prompt-guess/script : → board
;; prompts users for a guess at mastermind
(define (prompt-guess/script)
  ...)
;; mastermind : board board number → number
;; plays mastermind, returning number of tries needed to get right answer
(define (mastermind target guess tries)
  (cond [(guess-correct? guess target) tries]
        [else (begin
                (output "You get ~a white and ~a black pegs"
                        (calc-white target guess)
                        (calc-black target guess))
                (mastermind target (prompt-guess/script) (+ 1 tries)))]))
(mastermind (random-target) (prompt-guess/script) 1)
```

- (10 points) What would be the contract on *prompt-guess/web*?

- (15 points) Rewrite the above code so that all calls to *prompt-guess/script* are in script position. (Rewrite only the definitions that change.)

4. (35 points) Interactive programs (ones that query the user for input to use in a calculation) are such a useful construct that we've decided to add them to Curly.

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

   (**define-struct** *run-iprog* (*prompt proc*))

   An expr is one of
   - a number,
   - a symbol
   - (*make-proc symbol expr*)
   - (*make-add expr expr*)
   - (*make-mult expr expr*)
   - (*make-call expr expr*)
   - (*make-run-iprog string expr*)

   The expr in *make-run-iprog* is expected to evaluate to a proc.

   Here's an example of a run-iprog construct in use:

```
> (interp (make-run-iprog "Enter a num"
                          (make-proc 'num (make-add 'num 3))))
Enter a num: 5 [the user types the 5 at the prompt]
8
```

   (a) (10 points) Our run-iprog definition only requests one user input at a time. Using the existing definition, write an example of a *make-run-iprog* that queries the user for two numbers and adds them.

(b) (20 points) Our previous interpreter appears below (with a clause for the new run-iprog case). Fill in the case for *run-iprog* and make any other edits necessary to implement the behavior shown in the above example. Use *printf* to display the prompt and *read* to get the input. **Do not copy the existing code – simply show where any edits should go, or cross-out and rewrite individual lines that must change.**

```
;; interp : expr → value
;; evaluates the expression and returns its answer
(define (interp an-exp)
  (cond [(number? an-exp) an-exp]

        [(symbol? an-exp) (error 'interp (format "Unbound variable ~a" an-exp)))

        [(proc? an-exp) an-exp]

        [(add? an-exp) (+ (interp (add-left an-exp))

                          (interp (add-right an-exp)))]

        [(call? an-exp)

         (local [(define aproc (interp (call-func an-exp)))]

           (interp (subst (proc-body aproc)

                          (proc-param aproc)

                          (interp (call-arg an-exp)))))]

        [(run-iprog? an-exp)

                                                              ]))
```

(c) (5 points) Assume you wanted to switch to the closure-based interpreter rather than the subst-based interpreter. Would you need to edit your code for the run-iprog case? Why or why not? (**Note: do not rewrite your code – just answer the question based on the code that you wrote.**)