CS2135, A03

Final Exam

Name:

Problem	Points	Score
1	30	
2	35	
3	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 let local define-syntax define-script begin

and the following primitive operations:

empty? cons? cons first rest list map filter
number? $+ - * / = < > <= >=$ *zero?*
symbol? symbol=? equal? eq?
boolean? **and or** *not*
printf

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

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

Unless a problem states otherwise, you are not required to use map and filter in your answers.

---

1. (30 points) A game program needs to keep track of which player should get to move next. Rather than assign a global variable for this task, you want to create a data structure that will keep producing the name of the next player each time you query it (we'll call this the *turns* data structure).

   For example, assume the players are named 'ann, 'bob, and 'chuck. If you had a program *gen-turns* to create the data structure and a program *produce-names* to extract the names in turn, the following interaction would work:

   > (*produce-names* (*gen-turns* 'ann))
   Another? y
   bob
   Another? y
   chuck
   Another? y
   ann
   Another? y
   bob
   Another? n
   done
   >

   (a) (8 points) What kind of data structure should you use for the turns (list, structure, stream, function/lambda, etc) and why? (You do not need to define the data structure for this part.)

(exam continues next page)

(b) (22 points) The following code provides a skeleton for the *gen-turns* and *produce-names* functions. Fill in the blanks so that the resulting code produces the interaction shown at the start of the problem. Define the turns data structure as part of your solution. You may define additional data structures or helper functions as necessary. **Observe the set of constructs you are allowed to use on the exam!**

```
;; gen-turns : symbol → turns
;; creates a turns data structure that uses the given name first
(define (gen-turns curr-name)
  ___________)
```

```
;; produce-names : turns → void
;; produces names in sequence as long as user answers 'y
(define (produce-names t)
  (begin (printf "Another? ")
         (cond [(symbol=? 'y (read))
                (begin (printf "~a~n" ___________)
                       (produce-names ___________))]
               [else (printf "done~n")])))
```

(exam continues next page)

2. (35 points) You want to implement a program for building family trees. The program will query the user for the names of people in the tree, then create the struct for a tree when the user is finished entering names. The data definition for the resulting trees is (similar to the one from the first part of the course):

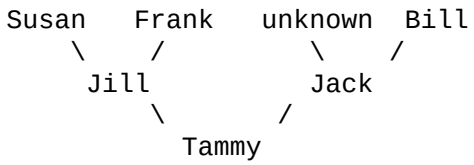
An *ftnode* is one of
- 'unknown,
- (*make-person symbol ftnode ftnode*)

(**define-struct** *person* (*name mother father*))

The original program is written as follows. In order to eventually run this on the web, we have used the **define-script** construct (that causes the defined function to terminate after producing its answer).

```
(define-script (build-tree)
  (begin (printf "Enter name [or unknown]: ")
         (build-tree-help (read))))
(define-script (prompt-parents pname)
  (begin (printf "Who was ~a's mother? [or unknown] " pname)
         (local ((define mname (read)))
           (begin (printf "Who was ~a's father? [or unknown] " pname)
                  (list mname (read))))))
(define-script (build-tree-help pname)
  (cond [(symbol=? pname 'unknown) 'unknown]
        [else (local ((define parents (prompt-parents pname)))
                (make-person pname
                             (build-tree-help (first parents))
                             (build-tree-help (second parents))))]))
```

(a) (15 points) Assume a user calls *build-tree* and then enters the information from the following family tree (so the user would enter *tammy* in response to the first prompt – assume omitted people are unknown).



With the above functions defined as scripts, which names get prompted for and what does the call to *build-tree* return?

(exam continues next page)

(b) (20 points) Convert the program so that all calls to scripts are in script position. You may write simple edits on the original code (but please write a converted function afresh if it would otherwise get messy).

(exam continues next page)

3. (35 points) A software company wants to develop a program to restrict the updates that its employees can make to code, documentation, and status reports. A set of company-defined access rules determine which employees can update which types of files. For example, the company might specify that

   - programmers can read and write code and documentation
   - testers can read code and write documentation
   - managers can read and write reports
   - managers can read documentation
   - ceos can read and write all files

   The program needs two sets of information: the rules (policy) that the company wants to enforce, and the roles that different employees play within the company (i.e., which jobs they perform on which parts of the software system). Each of these can be captured as a language.

   (a) (12 points) The company proposes the following language for capturing information about employees.

```
;; An employee is a (make-employee symbol list[asgmt])
(define-struct employee (name assignments))
;; An asgmt is a (make-asgmt symbol symbol)
(define-struct asgmt (job component))
;; Personnel is a (make-personnel symbol list[employee])
(define-struct personnel (ceo employees))

(make-personnel 'muffy
                (list (make-employee 'laura (list (make-asgmt 'programmer 'interface)
                                                  (make-asgmt 'tester 'database)))
                      (make-employee 'jimbo (list (make-asgmt 'manager 'interface)))
                      (make-employee 'muffy (list (make-asgmt 'programmer 'database)))))
```

   Identify 3 different things you could do to improve this language using techniques we covered in the course (making the same change to two parts of the language would **NOT** count as "different"). For each improvement, state what technique you would use to implement it (i.e.: "introduce a helper function to insert the boolean"). **Do not write any code – a few words per improvement will suffice**.

(exam continues next page)

(b) (23 points) The company proposes the following language for writing down policies (this example corresponds to the sample given at the start of the problem).

```
(define check-policy
  (policy-checker
   (programmer (read write) (code documentation))
   (tester (read) (code))
   (tester (write) (documentation))
   (manager (read write) (reports))
   (manager (read) (documentation))
   (ceo (read write) (code documentation reports))))
```

Each policy becomes a program that can be used to check whether a particular job can perform an access to a certain kind of file. For example, the following interaction uses the policy defined above:

```
> (check-policy 'programmer 'write 'code)
true
> (check-policy 'programmer 'write 'reports)
false
```

Write a macro for *policy-checker* such that this example and interaction works as shown. Your macro should be able to support any policy of this general format (in other words, it should allow different numbers of roles, rules, kinds of files, and kinds of accesses).