CS2135, A-01

Midterm Exam

Name:

Problem	Points	Score
1	10	
2	20	
3	35	
4	35	
	Total	
Extra Credit	10	

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

and the following primitive operations:

empty? cons? cons first rest 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. (10 points) State the contract and purpose for *filter*.

2. (20 points) Our Curly language lacks conditionals; we want to augment the data definition for Curly programs to include such a construct (called *switch*). Switch statements allow a program to execute one of several statements, depending upon the value of a expression. For example, the expression on the left converts some common celcius temperatures to fahrenheit:

```
{switch temp                              {switch {f x}
 case 0 do 32                              case 0 do {g 4}
 case 100 do 212                           case 1 do 2
 case -40 do -40                           case 2 do {f 7} + 4
}                                         }
```

Augment the Curly data definition (**not** the evaluator) to support switch-statements [hint: consider the HTML data definitions that you did in lab]. In your answer, do not copy parts of the existing data definitions that do not change, but be sure to indicate where you would put your changes in the existing data definition (are you adding a new definition, a new line in a specific existing defn, etc).

As a reminder, our current data definitions for Curly programs are as follows:

An expr is one of
- a number,
- (*make-var symbol*)
- (*make-proc symbol expr*)
- (*make-plus expr expr*)
- (*make-mult expr expr*)
- (*make-apply expr expr*)

A defn is a (*make-def symbol expr*)

3. (35 points) Characters in an adventure game are modeled by several pieces of information: their type (wizard, elf, or warlord), an indication of whether they are invisible, a spell they can cast (a function from characters to characters), and how many lives they have left. The following data definition captures characters:

   A character is a
   (*make-character symbol boolean* (*character* -> *character*) *number*)
   (**define-struct** *character type invisible? spell lives*)

   Write the following programs using the characters data definition. Use map and filter where appropriate; correct solutions that miss uses of map and filter will receive partial credit.

   (a) (10 points) *all-elves-invisible?* that consumes a list of characters and returns a boolean indicating whether all of the elves in the list are invisible.

   (b) (10 points) *cast-spell-all* that consumes a list of characters and a spell and returns a list of characters. The program should cast the given spell on every character in the list and return the new list of characters.

(c) (15 points) *make-lives-spell* that consumes two characters and returns a spell. The returned spell should create a character with the max number of lives of the two original characters (leaving other character attributes intact). Example:

```
(define new-spell
  (make-lives-spell (make-character 'wizard #t spell-1 4)
                    (make-character 'warlord #f spell-2 8)))

  (new-spell (make-character 'elf #f spell-3 2))
= (make-character 'elf #f spell-3 8)
```

4. (35 points) Consider the following data definition for binary trees:

A btree (binary tree) is either
- *false*, or
- (*make-bintree value btree btree*)

(**define-struct** *bintree* (*data left right*))

(a) (15 points) Write a program *sign-tree* which consumes a binary tree over numbers and produces a binary tree over symbols. The produced binary tree should have the same structure as the input tree, but where each number is replaced with 'pos, 'neg, or 'zero (depending upon whether the number is positive, negative, or zero). For example

```
      10                    'pos
     /  \                  /    \
    5   -4      ==>    'pos    'neg
   / \  / \            /   \   /   \
  2  -3 6  0        'pos 'neg 'pos 'zero
```

```
(sign-tree (make-bintree 10 (make-bintree 5 (make-bintree 2 false false)
                                            (make-bintree −3 false false))
                            (make-bintree −4 (make-bintree 6 false false)
                                             (make-bintree 0 false false))))
= (make-bintree 'pos
                (make-bintree 'pos (make-bintree 'pos false false)
                                   (make-bintree 'neg false false))
                (make-bintree 'neg (make-bintree 'pos false false)
                                   (make-bintree 'zero false false))))
```

(b) (15 pts) The *sign-tree* program resembles a map, in that it produces an output tree with the same structure as the input tree, but with the data at the nodes transformed. Write a program *btree-map* that consumes a btree and a function and returns a new btree with the same structure as the input tree, but with each node transformed according to the function.

(c) (5 points) Rewrite *sign-tree* in terms of your *tree-map* program.

(d) **Extra Credit Problem:** Do **not** attempt this problem unless you are satisfied with your answers to the four required problems!

Assume you have a subset of Scheme containing only lists: *i.e.*, the only constant you have is *empty* and the only primitives are *cons*, *first*, *rest*, *cons?* and *empty?* (you also have the constructs **define**, **define-struct**, **cond**, and **else**). Consider how to implement basic arithmetic in this subset of Scheme.

i. Propose a representation for natural numbers in this language (*i.e.*, provide a data definition).
ii. Write a program *multiply* that multiplies two numbers given in your representation; the result should be another number in your representation.