CS 115 Fall 2012

Assignment 7

Due at 9:00 AM on Wednesday, November 21

For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch. The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources. Please read the course Web page for more information on assignment policies and how to organize and submit your work.

Be sure to download the interface file from the course Web page and to follow all the instructions listed in the Style Guide (on the Web page and in the printed package of handouts).

For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe, including the definition of constants and helper functions where appropriate.
Contracts: You are expected to use both built-in and user-defined types. See Section 1.6 of the Style and Submission Guide for details.
Language level: Beginning Student with List abbreviation.
Coverage: Module 8.

Useful structure and data definitions:
(define-struct card (value suit))
;; A **card** is a structure (make-card v s), where
;; v is an integer in the range from 1 to 10 and
;; s is a symbol from the set 'hearts, 'diamonds, 'spades, and 'clubs.

(define-struct card-node (value cards left right))
;; A binary search tree **hand-bst** is either
;; empty or,
;; a structure (make-card-node v c l r), where
;; v is an integer, ranging between 1 and 10 showing the value of a playing card
;; c is a list of cards: (listof card) with the same value, v, but from different suits
;; and l, r are of type hand-bst.
;; all card values in l are less than v and
;; all card values in r are greater than v

(define-struct bae (fn arg1 arg2))
;; A binary arithmetic expression (binexp) is either
;; • a number or
;; • a structure (make-bae f a1 a2), where
;;   - f is a symbol in the set '*, '+, '/, '−,
;;   - a1 is a binexp, and
;;   - a2 is a binexp.

Question 1- In this question, you are responsible to choose a winner between two card players based on a card value drawn by you. Each player has several cards in his/her hand and cards in each hand are stored in a binary search tree based on their values. For this purpose, *hand-bst* type is used to show a hand. In addition, the structure *card* is used to present the information about each card.

Write a Scheme function *winning-hand* that consumes two hands, *hand1* and *hand2* of type *hand-bst* and a card value, *drawn-value*, between 1 and 10 and produces a symbol '*handone* or '*handtwo* or '*tie.*

Winning strategy: To find a winner, you should use the fact that each hand is ordered using the binary search tree property. For each hand, you should ignore all the cards with values less than the value of the drawn card. For remaining cards, if a card is red (hearts or diamonds), its value should be doubled as a bonus for that hand. Then you add up the values of cards that are greater than or equal to the value of the drawn card. At the end, the hand with the greater total value is the winning hand. If the sum of the values for both hands is the same, then there is no winner and we have a tie.

Note: You should take advantage of the ordering property of binary search trees in your solution, and only search the necessary parts of tree. Certainly, there are occasions you need to recurse on both subtrees, but if it is possible to recurse only on one subtree, you should avoid recursing on both subtrees, otherwise you will lose marks.

Constant definitions used to define two *hand-bsts*, *myhand* and *friendhand*:

Constants for cards
(define *oneofdiamonds* (*make-card* 1 'diamonds))
(define *oneofspades* (*make-card* 1 'spades))
(define *oneofclubs* (*make-card* 1 'clubs))
(define *twoofhearts* (*make-card* 2 'hearts))
(define *twoofclubs* (*make-card* 2 'clubs))
(define *threeofhearts* (*make-card* 3 'hearts))
(define *threeofdiamonds* (*make-card* 3 'diamonds))
(define *threeofspades* (*make-card* 3 'spades))
(define *fiveofhearts* (*make-card* 5 'hearts))
(define *fiveofspades* (*make-card* 5 'spades))
(define *fiveofclubs* (*make-card* 5 'clubs))
(define *sevenofspades* (*make-card* 7 'spades))

Constants for lists of cards
(define *firstonelist* (*list oneofclubs*))
(define *firsttwolist* (*list twoofclubs*))
(define *firstthreelist* (*list threeofhearts threeofdiamonds threeofspades*))
(define *firstfivelist* (*list fiveofhearts fiveofspades*))
(define *secondonelist* (*list oneofdiamonds oneofspades*))
(define *secondtwolist* (*list twoofhearts*))
(define *secondfivelist* (*list fiveofclubs*))
(define *secondsevenlist* (*list sevenofspades*))

Constants for hand-bsts
```
(define myrightsubtree (make-card-node 5 firstfivelist empty empty))
(define myleftsubtree (make-card-node 1 firstonelist empty
                                    (make-card-node 2 firsttwolist empty empty)))
(define myhand (make-card-node 3  firstthreelist  myleftsubtree myrightsubtree))

(define friendrightsubsubtree (make-card-node 5 secondfivelist empty empty))
(define friendleftsubsubtree (make-card-node 1 secondonelist empty empty))
```

(define *friendleftsubtree* (*make-card-node 2 secondtwolist friendleftsubsubtree friendrightsubsubtree*))

(define *friendhand* (*make-card-node 7 secondsevenlist friendleftsubtree empty*))

Therefore:

- *(winning-hand myhand friendhand 4) => 'handone*
- *(winning-hand myhand friendhand 6) => 'handtwo*

In the first example, the *drawn-card* value is 4, therefore, only cards with value 5 in *myhand* are considered to be added to the total value. In this hand, there are two cards with value 5, one is black so its value is added once and one is red, which value is doubled and added to the total value of the hand. So, the total value would be 5+(5*2)=15. For *friendhand*, there are two cards 7 and 5 in this hand with the value greater than the value of *drawn-card*. So only these two cards should be considered to be added to the total value of the hand. Both of these cards are black cards so their value is counted once, thus, the total value of *friendhand* is 5+7=12 and the winning hand will be *myhand*.

Question 2- Write a Scheme function *multiply-exp* that consumes two numbers *m* and *n* and a *binexp* called *multexp*, and produces a new *binexp*, in which each number, *nbr,* of the original expression is replaced with a new expression *nbr*(m+n),* which is a *binexp* itself. If *m+n* results in zero, then the function should produce the original binary expression.

Constants for binary arithmetic expressions
(define *bae1* (*make-bae* '+ 2 3))
(define *left-bae* (*make-bae* '* 2 3))
(define *right-bae* (*make-bae* '- 7 3))
(define *root-bae* (*make-bae* '/ *left-bae right-bae*))

For example:

- *(multiply-exp -3 2 2) =>*
  (make-bae ' 2 (make-bae '+ -3 2))*
- *(multiply-exp -3 2 root-bae) =>*

```
        (make-bae '/ (make-bae '* (make-bae '* 2 (make-bae '+ -3 2))
                                   (make-bae '* 3 (make-bae '+ -3 2)))
                     (make-bae '- (make-bae '* 7 (make-bae '+ -3 2))
                                   (make-bae '* 3 (make-bae '+ -3 2))))
```

- *(multiply-exp -2 2 bae1) => (make-bae '+ 2 3)*

Question 3- We would like to continue the game in Question 1 by giving the loser of the game one more chance. The loser is allowed to draw a card and add it to his/her hand and play again. Your responsibility is to insert the card in the binary search tree of the losing hand while ensuring its ordering property is not violated. Write a Scheme function *insert-card* that consumes a hand, *ahand,* of type *hand-bst*, a number, *aval*, between 1 and 10 as a card value, and a symbol, *asuit,* as the card suit and produces a new hand of type *hand-bst* that includes the drawn card in it. If a card with the same value as the *aval* value already exists in *ahand,* the card is added to the beginning of the list of cards with that value. If *ahand* is empty, a binary search tree with one node is created (see an example for it below).
Note: This game is done with only one deck of cards, so it is not possible to have two cards of the same suit and value.

For example, if the *friendhand* from Question 1 is the loser hand then:

- *(insert-card friendhand* 9 'clubs*)* => *(make-card-node* 7
```
                                  (list (make-card 7 'spades))
                                  (make-card-node 2
                                    (list (make-card 2 'hearts))
                                     (make-card-node 1
                                       (list (make-card 1 'diamonds) (make-card 1 'spades))
                                       empty
                                       empty)
                                     (make-card-node 5
                                       (list (make-card 5 'clubs))
                                       empty
                                       empty))
                                   (make-card-node 9
                                     (list (make-card 9 'clubs))
                                     empty
                                     empty))
```
- *(insert-card friendhand* 5 'diamonds*)* => *(make-card-node* 7
```
                                  (list (make-card 7 'spades))
                                  (make-card-node 2
                                     (list (make-card 2 'hearts))
                                       (make-card-node 1
                                          (list (make-card 1 'diamonds) (make-card 1 'spades))
                                          empty
                                          empty)
                                       (make-card-node 5
                                         (list (make-card 5 'diamonds) (make-card 5 'clubs))
                                         empty
                                         empty))
                                   empty)
```
- *(insert-card empty* 9 'clubs*)* => *(make-card-node* 9
```
                              (list (make-card 9 'clubs))
                              empty
                              empty)
```

Question 4- Write a Scheme function *modern-species* that consumes a taxon, *t*, and produces a list of modern descendants' names of this taxon, sorted in alphabetical order. If the taxon is a modern taxon, the result contains only the name of that modern taxon. For example, using the constants from the taxon teachpack:

- *(modern-species vertebrate)* => (*list* "Gallus Gallus" "Homo Sapiens" "Pan Troglodytes" "Rattus Norvegicus"*)*.
- *(modern-species human)* => (*list* "Homo Sapiens")

Hint: In writing your function, you may use a sorting helper function such as merge-sort for sorting lists of strings resulting from the left and the right subtrees.

Note: This question uses the taxon teachpack, which you can download from the course webpage under the Resources/DrRacket & Teachpacks link, and add to your a07q4.rkt file. You may use the constants defined in the teachpack in your testing and examples. Do not copy the taxon definitions, i.e. , t-modern and t-ancient into your submitted file, as it causes our tests to fail and you will lose correctness marks.