CS 135 Winter 2015
Troy Vasiga

Assignment: 8

Due:	Tuesday, March 17, 2015 9:00pm
Language level:	Intermediate Student
Allowed recursion:	Pure Structural and Structural Recursion with Accumulators
Files to submit:	`bst-remove.rkt`, `participation.rkt`, `mapfn.rkt`
Warmup exercises:	HtDP 17.3.1, 17.6.4, 19.1.5, 20.1.1, 20.1.2, 24.0.7, 24.0.8
Practise exercises:	HtDP 17.6.5, 17.6.6, 19.1.6, 20.1.3, 21.2.3, 24.0.9

- All helper functions must be **local** definitions.
- You are not required to provide examples or tests for **local** function definitions. You are still encouraged to do informal testing of your helper functions outside of your main function to ensure they are working properly. Functions defined at the "top level" must include the complete design recipe.
- You may define global constants if they are used for more than one part of a question. This includes defining constants for examples and tests. Constants that are only used by one top level function should be included in the **local** definitions.
- In this assignment your *Code Complexity/Quality* grade will be determined both by how clear your approach to solving the problem is, and how effectively you use local constant and function definitions. You should include local definitions to avoid repetition of common subexpressions, to improve readability of expressions and to improve efficiency of code.
- You may use any list function mentioned in the course notes, up to Module 09, unless otherwise explicitly mentioned.
- You may **not** use abstract list functions.
- You may reuse the provided examples, but you should ensure you have an appropriate number of examples and tests.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.

Here are the assignment questions you need to submit.

1. Perform the Assignment 8 questions using the online evaluation "Stepping Problems" tool linked to the course web page and available at

   https://www.student.cs.uwaterloo.ca/~cs135/stepping/

   The instructions are the same as A03 and A04; check there for more information, if necessary.

   Reminder: You should not use DrRacket's Stepper to help you with this question, for a few reasons. First, as mentioned in class, DrRacket's evaluation rules are slightly different from the ones presented in class; you are to use the ones presented in class. Second, in an exam situation, of course, you will not have the Stepper to help you. Third, you can re-enter steps as many times as necessary to get them correct, so you might as well maximize the educational benefit.

2. Recall from lectures, the definition of a binary search tree:

```
(define-struct node (key val left right))
;; A binary search tree (BST) is one of
;; * empty
;; * (make-node Num Str BST BST),
;; where for each node (make-node k v l r), every node in the subtree
;; rooted at l has a key less than k, and every node in the subtree rooted
;; at r has a key greater than k.
```

   For the purpose of presenting examples, we will use the following binary search tree throughout this question:

```
(define t (make-node 5 "" (make-node 3 "" (make-node 2 "" (make-node 1 "" empty empty)
                                                          empty)
                                          (make-node 4 "" empty empty))
                          (make-node 7 "" (make-node 6 "" empty empty) empty)))
```

   On the last assignment, you wrote a function to add a new (key, value) pair to an existing binary search tree. Removing a node from a binary search tree is a somewhat harder problem, which you will explore here. There are several cases to consider:

   - If the node to be removed has no children, then it can simply be removed. For example:

```
(check-expect (bst-remove t 1)
              (make-node 5 "" (make-node 3 "" (make-node 2 "" empty empty)
                                              (make-node 4 "" empty empty))
                              (make-node 7 "" (make-node 6 "" empty empty) empty)))
```

- If the node to be removed has exactly one child, we can replace it with its only child. For example:

```
(check-expect (bst-remove t 7)
    (make-node 5 "" (make-node 3 "" (make-node 2 "" (make-node 1 "" empty empty)
                                                    empty)
                                    (make-node 4 "" empty empty))
                    (make-node 6 "" empty empty)))
```

- If the node (with key k) to be removed has two children, we have to do more work. We must first find the node's *inorder successor*—this is the node in the tree with the smallest key larger than k. We then replace the node with its inorder successor, and instead remove the inorder successor. Since the inorder successor will not have a left child (why?), we will be able to remove it according to one of the first two cases. For example:

```
(check-expect (bst-remove t 5)
    (make-node 6 "" (make-node 3 "" (make-node 2 "" (make-node 1 "" empty empty)
                                                    empty)
                                    (make-node 4 "" empty empty))
                    (make-node 7 "" empty empty)))
```

  Note that the node whose key is 6 is the inorder successor of the node with key 5; hence the contents of the node with key 6 replace those of the node to be removed, and the original node with key 6 is removed. Keep in mind that not only the key, but also the value field of the inorder successor must be copied into the node to be removed.

Write the function *bst-remove* that consumes a BST and a key k, and produces the BST resulting from removing the node with key k from the given BST (if it exists). If there is no node with key k, in the tree, then *bst-remove* should produce the original BST unchanged. If you write any helper functions, you must declare them locally.

Place your solution in the file `bst-remove.rkt`.

3. Redo Assignment 5, Question 2, except that any helper functions must be declared locally. That is, write a function *participation* that consumes a non-empty list of numbers (representing participation grade scores) and produces a number (in the range 0 to 5 inclusive) representing the total participation mark (recall that the participation component has a weighting of 5% in the calculated final grade for CS135). For the participation grade, each number in the list is either 0, 1 or 2. You should take the top 75% of these scores in your computation: use the Racket function *ceiling* to compute the correct number of questions that should be used in the calculation. For example, if there are 5 questions in total, the best 4 should be used, and if there are 7 questions in total, the best 6 should be used. Your answer (the total participation mark) should be left in decimal form (i.e., not rounded or floored or ceilinged). Hints:

   - You are trying to determine a fraction. The denominator of the fraction is the maximum

possible score on the number of questions that should be counted. The numerator of the fraction is a bit more complicated.

- You would like to count as many scores of 2 as possible: if there are enough scores of 2, the mark should be 5 (i.e., perfect).
- If there not enough 2's, then use up all of the 2's, and see how many 1's you have. If you have enough 2's and 1's, then compute the score. If you don't have enough 2's and 1's, then add enough 0's in order to get the correct number of answers.
- Not surprisingly, you will need a few helper functions. **All helper functions should be declared locally.** The function *participation* should be the only globally defined function.

Place your solution in the file `participation.rkt`. You are allowed to use sample solutions, from the course webpage, but you should cite your source of any functions in a comment in your code.

4. In class, we have seen that we are now able to put functions into lists. What can we do with lists of functions? One thing is to apply each function in the list to a common set of inputs. Write a function *mapfn* which consumes a list of functions (each of which takes two numbers as arguments) and a list of two numbers. It should produce the list of the results of applying each function in turn to the given two numbers. For example,

   (*mapfn* (*list* $+$ $-$ $*$ / *list*) '(3 2)) $\Rightarrow$ '(5 1 6 1.5 (3 2))

   Note that the above list being passed to *mapfn* has five elements, each of which is a function that can take two numbers as input. The resulting list is also of length five. You can assume the function will be well-defined for any function and/or pair of numbers.

   Pay close attention to the contract for your function. Hint: you may want to use **local** to give a function a name at one point. However, do not use either global or local helper functions in this question.

   Place your solution in the file `mapfn.rkt`.

This concludes the list of questions for which you need to submit solutions. As always, do not forget to check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

⟨exp⟩ = ⟨var⟩|(**lambda** (⟨var⟩) ⟨exp⟩) | (⟨exp⟩⟨exp⟩)

Although Professor Temple does not trust **define**, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn't look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it's just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want (this should be familiar from the *addgen* example from class, slide 09-39). This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple's definition of zero. It is the function which ignores its argument and returns the identity function.

(**define** *my-zero* (**lambda** (*f*) (**lambda** (*x*) *x*)))

Another way of describing this representation of zero is that it is the function which takes a function f as its argument and returns a function which applies f to its argument zero times. Then "one" would be the function which takes a function f as its argument and returns a function which applies f to its argument once.

(**define** *my-one* (**lambda** (*f*) (**lambda** (*x*) (*f x*))))

Work out the definition of "two". How might Professor Temple define the function *add1*? Show that your definition of *add1* applied to the above representation of zero yields one, and applied to one yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple's representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple's definitions of true and false.

(**define** *my-true* (**lambda** (*x*) (**lambda** (*y*) *x*)))
(**define** *my-false* (**lambda** (*x*) (**lambda** (*y*) *y*)))

Show that the expression ((*c a*) *b*), where *c* is one of the values *my-true* or *my-false* defined above, evaluates to *a* and *b*, respectively. Use this idea to define the functions *my-and*, *my-or*, and *my-not*.

What about *my-cons*, *my-first*, and *my-rest*? We can define the value of *my-cons* to be the function which, when applied to *my-true*, returns the first argument *my-cons* was called with, and when applied to the argument *my-false*, returns the second. Give precise definitions of *my-cons*, *my-first*, and *my-rest*, and verify that they satisfy the algebraic equations that the regular Racket versions do. What should *my-empty* be?

The function *my-sub1* is quite tricky. What we need to do is create the pair $(0,0)$ by using *my-cons*. Then we consider the operation on such a pair of taking the "rest" and making it the "first", and making the "rest" be the old "rest" plus one (which we know how to do). So the tuple $(0,0)$ becomes

$(0,1)$, then $(1,2)$, and so on. If we repeat this operation n times, we get $(n-1,n)$. We can then pick out the "first" of this tuple to be $n-1$. Since our representation of n has something to do with repeating things n times, this gives us a way of defining *my-sub1*. Make this more precise, and then figure out *my-zero?*.

If we don't have **define**, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. You can read more about it at the following URL (PostScript document):

`http://www.ccs.neu.edu/home/matthias/BTLS/tls-sample.ps`

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, the predecessor of Racket, and is the reason that the teaching languages retain the keyword **lambda** for use in defining anonymous functions.