Assignment: 8

Due:	Tuesday, March 15, 2016 9:00pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	Pure structural recursion only
Files to submit:	`bst.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 only use the list functions that have been discussed in the notes up to the end of Module 9, unless explicitly allowed in the question.
- 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. In this question, you will implement important functions that process BSTs. You must use **local** to avoid repeating a recursive function application with the same arguments. Place your solutions in a file called `bst.rkt`, which should contain the following structure definition of a BST node:

   (*define-struct node* (*key val left right*))

   (a) We saw in class how there are many different BSTs that contain the same set of (key,value) pairs. Sometimes it is useful to rearrange a BST so that the resulting BST has a particular desired property (but the same set of (key,value) pairs). In this question, we will develop *root-at-smallest* which will rearrange a BST so that its smallest element is at the root. We say that (*root-at-smallest empty*) $\Rightarrow$ *empty*.

   Note that the smallest element of a BST is at the root when the root's left child is *empty*.

   Here's how you do the required rearrangement:

   - If the BST is *empty*, or if the root's left child is *empty*, then the smallest element is already at the root, so produce the same BST as you were given.
   - Otherwise, the root has a left subtree (the subtree rooted at the left child; see slide 08-04 for terminology). Recursively rearrange that left subtree so that its smallest element is at its root.
   - Now the BST will look like Figure 1(a). y is the key at the root, x is the key at the root's left child (and so is the smallest key in the whole BST), **A** is the right subtree of the node containing x, and **B** is the right subtree of the root. Note that all the keys in **A** are greater than x and less than y. Also, all the keys in **B** are greater than y.
   - Produce the rearranged tree, as shown in Figure 1(b). The smallest key in the BST (x) is now at the root; its left child is of course *empty*. The root's right child contains the key y, and that node's left and right subtrees are **A** and **B**, respectively.

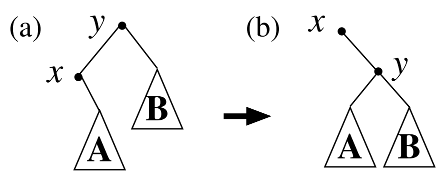

Figure 1: [For part (a)] Moving the smallest element of a BST to the root

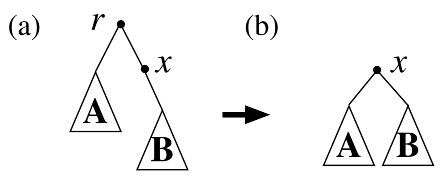

Figure 2: [For part (b)] Removing the root of a BST

Don't forget to handle the BST's values, as well as its keys; if you move a key, you need to move its associated value along with it.

Submit the function *root-at-smallest*. Although it is tricky to think about, the function itself is quite simple. (The sample solution is less than 10 lines long.) Hint: use local definitions to identify in your code the significant parts of Figure 1; i.e. x, y, A, and B.

(b) The reason the above *root-at-smallest* function is useful is that it helps us implement *bst-remove*. *bst-remove* consumes a key (which is a Num) and a BST, and produces the BST with the node containing the given key removed. (Produce the original tree unchanged if the key is not present.) Some rearrangement may have to be done in order to accomplish this; you must do it as follows:

- If the key to be removed is *not* at the root of the BST, recurse as you did with *bst-add* in A07. Note that you will recurse on only *one* of the two children.
- Otherwise, you are trying to remove the node at the root of the tree. If either of its children is *empty*, produce the other child; this will be the BST containing all of the other nodes in the original BST.
- The trickiest case is when both children are non-empty. In this case, use the *root-at-smallest* function from part (a) to rearrange the right subtree. The resulting BST will look like Figure 2(a). *r* is the key to be removed (which is at the root), *x* is the key at the root's right child, **A** is the root's left subtree, and **B** is the right subtree of the node containing *x*.
- Now removing the root is simply a matter of producing the BST shown in Figure 2(b): the node containing *x* is the new root, and its left and right subtrees are **A** and **B** respectively.

Submit the function *bst-remove*. This function is also tricky to think about, but not complicated to write; the sample solution is less than 15 lines long.

3. This question deals with functional abstraction. Place your solution in the file `mapfn.rkt`.

   (a) 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.

   Pay close attention to the contract for your function. Hint: you will probably discover you need 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.

   (b) Write a predicate function *is-in-order?* that consumes a list of (predicate function, binary relational operator) pairs (see contract below) and a list of operands. A pair in the first list consists of a predicate to determine the data type of the list of operands and a binary relational operator that consumes the same data type. The first list is a non-empty list and can contain at most one such pair from any data type. Each element of the second list is of the same data type. The second list may be empty.

   Here is an acceptable contract for `is-in-order?`. (Yes, we are giving you the contract.) Type this contract into your solutions.

   ```
   ;; is-in-order?: (listof (list (Any → Bool) (X X → Boolean)))
   ;;               (listof Any) → (anyof Bool 'error)
   ;; requires: first list is non-empty
   ```

   The predicate *is-in-order?* behaves as follows:

   - if the operand list is *empty* or has one element, produce *true*
     (*is-in-order?* (*list* (*list integer?* <)) *empty*) $\Rightarrow$ *true*
     (*is-in-order?* (*list* (*list integer?* <)) (*list* 1)) $\Rightarrow$ *true*

   - if no binary relational operator can be applied to the operand list with two or more elements, produce 'error
     (*is-in-order?* (*list* (*list integer?* >)(*list string? string>?*)) (*list* 'a 'b 'c)) $\Rightarrow$ 'error

   - if the application of at least one relational operator on all consecutive elements of the list of operands produces true, then *is-in-order?* produces true, otherwise produces *false*
     Note: in this case, at least one of the binary relational operators can be applied to the list of operands
     (*is-in-order?* (*list* (*list integer?* <)) (*list* 1 2 7)) $\Rightarrow$ *true*

(*is-in-order?* (*list* (*list integer?* >)) (*list* 1 2 7)) ⇒ *false*
(*is-in-order?* (*list* (*list integer?* >)) (*list* 2 1 7)) ⇒ *false*
(*is-in-order?* (*list* (*list string? string<?*)) (*list* "1" "2" "7")) ⇒ *true*
(*is-in-order?* (*list* (*list symbol? symbol=?*) (*list integer?* =)) (*list* 1 1 1)) ⇒ *true*
(*is-in-order?* (*list* (*list integer?* >) (*list integer?* <)) (*list* 1 1 1)) ⇒ *false*

You may assume a predicate will either produce *true* for every operand in the list of operands, or it will produce *false* for every operand so you do not need to consider the following case:
(*is-in-order?* (*list* (*list even?* <)) '(2 3 4 5))

Hint: you can create

- a helper function to check if all the predicates will produce *false* on the first operand of the operand list
- a helper function to check if one of the binary relation operators will produce *true* when applied to all consecutive operands

You may require additional helper functions.

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public 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 Scheme 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, a predecessor of Racket, and is the reason that the teaching languages retain the keyword **lambda** for use in defining anonymous functions.