CS 135
Winter 2014
Troy Vasiga

Assignment: 7

Due:	Tuesday, March 18, 2014, 9:00pm
Language level:	Intermediate Student
Allowed recursion:	Structural, Accumulative
Files to submit:	`bst-remove.rkt`, `mapfn.rkt`, `eval-apply.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

Here are the assignment questions you need to submit.

1. Perform the Assignment 7 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 String 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. In this question, you will extend the eval/apply interpreter for general arithmetic expressions (from Slides 8-66 and 8-67) to accommodate defined constants. The data and structure definitions for general arithmetic expressions will now be as follows:

```
;; An arithmetic expression (AE) is one of
;; * a Num
;; * a Symbol (compliant with Scheme rules for identifiers)
;; * (cons Symbol AEList) (where Symbol is either '+ or '*)
;;
;; A list of arithmetic expressions (AEList) is one of
;; * empty
;; * (cons AE AEList)
;;
;; A list of defines and one expression (LODOE) is one of:
;; * an AE
;; * (cons '(define Symbol Num) LODOE)
```

(a) Write the function *lookup-al* which consumes an association list (with Symbols as keys and Nums as values) and a Symbol to look up, and produces the value associated with the consumed symbol (if the consumed symbol is in the association list) or *false* if the symbol is not in the given association list. You may use AL as the type of the first consumed parameter, even though it differs from the AL previously defined in lecture (which had Numbers for keys and Strings for values).

(b) Write the functions *eval1* and *apply1*, which differ from eval/apply as follows: *eval1* consumes an arithmetic expression and an association list (where the key is Symbol and the value is Num), matching variable names to values. For example,

(*eval1* '(+ x (∗ y 2)) '((x 2) (y 3) (z 4))) ⇒ 8

apply1 consumes three parameters: a Symbol ('+ or '∗), a list of expressions (i.e., an *AEList*), and an association list. The function *apply1* produces the number resulting from applying the function specified by the Symbol to the expressions in the list. For example:

(*apply1* '+ '(x (∗ y 2)) '((x 2) (y 3) (z 4))) ⇒ 8

(c) Write the functions *eval2* and *apply2*, which operate like *eval1* and *apply1*, except that *eval2* consumes a LODOE as parameter. For example,

(*eval2* '((define x 2) (define y 3) (define z 4) (+ x (∗ y 2))))

should produce the value 8. *apply2* consumes a symbol ('+ or '∗), a list of expressions (i.e., an *AEList*), and an association list, and computes the number resulting from applying the function specified by the symbol to the expressions in the list. Note that neither of *eval2* nor *apply2* may call *eval1* or *apply1*. For example,

(*apply2* '+ '(x (∗ y 2)) '((x 2) (y 3) (z 4))) ⇒ 8

(**Hint:** Consider writing a local helper function that uses an accumulator variable to store an association list of defines processed so far. As defines (e.g. '(define x 2)) are

removed from the first parameter, the corresponding pairs (e.g. '(x 2)) are added to the second pair.) You may assume the following:

- No variable is defined more than once.
- All variables needed by the expression are defined.
- The defines only map variable names to literal numbers (so, for example, '(define x (+ 1 1)) would not be allowed).

(d) **5% Bonus** Write the functions *eval3* and *apply3* that differ from *eval2* and *apply2* in that the defines are now allowed to use expressions. For example,

(*eval3* '((define x 2) (define y (+ x 1)) (define z (+ x x)) (+ x (* y 2))))

should produce the value 8.

In this subquestion, you may **not** make the assumptions given in the previous subquestion. If a variable is defined more than once, or missing, or used before it is defined, your program should evaluate (*error* "Bad defines.").

All helper functions you use, other than *lookup-al* must be declared locally.

Place your solution in the file `eval-apply.rkt`.

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)) ⇒ '(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 may want to use *local* to give a function a name at one point (or you may not). 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.

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

The material below first explores the implications of the fact that Scheme programs can be viewed as Scheme data, before reaching back seventy years to work which is at the root of both the Scheme language and of computer science itself.

The text introduces structures as a gentle way to talk about aggregated data, but anything that can be done with structures can also be done with lists. Section 14.4 of HtDP introduces a representation of Scheme expressions using structures, so that the expression (+ (* 3 3) (* 4 4)) is represented as

```
(make-add
  (make-mul 3 3)
  (make-mul 4 4))
```

But, as discussed in lecture, we can just represent it as the hierarchical list '(+ (∗ 3 3) (∗ 4 4)). Scheme even provides a built-in function *eval* which will interpret such a list as a Scheme expression and evaluate it. Thus a Scheme program can construct another Scheme program on the fly, and run it. This is a very powerful (and consequently somewhat dangerous) technique.

Sections 14.4 and 17.7 of HtDP give a bit of a hint as to how *eval* might work, but the development is more awkward because nested structures are not as flexible as hierarchical lists. Here we will use the list representation of Scheme expressions instead. In lecture, we saw how to implement *eval* for expression trees, which only contain operators such as $+,-,*,/$, and do not use constants.

Continuing along this line of development, we consider the process of substituting a value for a constant in an expression. For instance, we might substitute the value 3 for x in the expression (+ (∗ x x) (∗ y y)) and get the expression (+ (∗ 3 3) (∗ y y)). Write the function *subst* which consumes a symbol (representing a constant), a number (representing its value), and the list representation of a Scheme expression. It should produce the resulting expression.

Our next step is to handle function definitions. A function definition can also be represented as a hierarchical list, since it is just a Scheme expression. Write the function *interpret-with-one-def* which consumes the list representation of an argument (a Scheme expression) and the list representation of a function definition. It evaluates the argument, substitutes the value for the function parameter in the function's body, and then evaluates the resulting expression using recursion. This last step is necessary because the function being interpreted may itself be recursive.

The next step would be to extend what you have done to the case of multiple function definitions and functions with multiple parameters. You can take this as far as you want; if you follow this path beyond what we've suggested, you'll end up writing a complete interpreter for Scheme (what you've learned of it so far, that is) in Scheme. This is treated at length in Section 4 of the classic textbook "Structure and Interpretation of Computer Programs", which you can read on the Web in its entirety at `http://mitpress.mit.edu/sicp/` . So we'll stop making suggestions in this direction and turn to something completely different, namely one of the greatest ideas of computer science.

Consider the following function definition, which doesn't correspond to any of our design recipes, but is nonetheless syntactically valid:

```
(define (eternity x)
  (eternity x))
```

Think about what happens when we try to evaluate (*eternity* 1) according to the semantics we learned for Scheme. The evaluation never terminates. If an evaluation does eventually stop (as is the case for every other evaluation you will see in this course), we say that it *halts*.

The non-halting evaluation above can easily be detected, as there is no base case in the body of the function *eternity*. Sometimes non-halting evaluations are more subtle. We'd like to be able to write a function *halting?*, which consumes the list representation of the definition of a function with one parameter, and something meant to be an argument for that function. It produces *true* if and only if the evaluation of that function with that argument halts. Of course, we want an application of *halting?* itself to always halt, for any arguments it is provided.

This doesn't look easy, but in fact it is provably impossible. Suppose someone provided us with code for *halting?*. Consider the following function of one argument:

```
(define (diagonal x)
  (cond
    [(halting? x x) (eternity 1)]
    [else           true]))
```

What happens when we evaluate an application of *diagonal* to a list representation of its own definition? Show that if this evaluation halts, then we can show that *halting?* does not work correctly for all arguments. Show that if this evaluation does not halt, we can draw the same conclusion. As a result, there is no way to write correct code for *halting?*.

This is the celebrated *halting problem*, which is often cited as the first function proved (by Alan Turing in 1936) to be mathematically definable but uncomputable. However, while this is the simplest and most influential proof of this type, and a major result in computer science, Turing learned after discovering it that a few months earlier someone else had shown another function to be uncomputable. That someone was Alonzo Church, about whom we'll hear more shortly.

For a real challenge, definitively answer the question posed at the end of Exercise 20.1.3 of the text, with the interpretation that *function=?* consumes two lists representing the code for the two functions. This is the situation Church considered in his proof.