Assignment: 7

Due:	Tuesday, March 8, 2016 9:00 pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure Structural, Accumulative, Generative recursion
Files to submit:	`bst.rkt`,`bstbonus.rkt` (optional), `filedir.rkt`
Warmup exercises:	HtDP 14.2.1, 14.2.2, 17.2.1, 17.3.1, 17.6.4, 17.8.3, 18.1.1, 18.1.2, 18.1.3, 18.1.4, 18.1.5
Practise exercises:	HtDP 14.2.3, 14.2.6, 17.1.2, 17.2.2, 17.6.2, 17.8.4, 17.8.8, 18.1.5

- Unless specifically asked in the question, you are not required to provide a data definition or a template in your solutions for any of the data types described in the questions. However, you may find it helpful to write them yourself and use them as a starting point.
- In your solutions, if you create any data types yourself that are beyond the question descriptions, or data types discussed in the notes, your program file must include a data definition.
- You may use the abbreviation (*list* ...) or quote notation for lists as needed.
- You may use any list functions discussed in the notes, except *reverse*.
- You may use any other list functions discussed in the notes up to and including module 9, unless the question specifies otherwise.
- Your *Code Complexity/Quality* grade will be determined by how clear your approach to solving the problem is and some basic efficiency requirements - i.e. your functions should avoid repeating the same function application multiple times as in the *max-list* example in slide 4 of module 7 and you should not be using *reverse* to fix the ordering of a list that you produce.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.

Caution: Most of the solutions on this assignment can be completed using functions and helper functions that are a reasonable length of code. If you find your solution requires pages and pages of code, you will want to spend more time reasoning about the solution to the question.

Here are the assignment questions you need to submit.

Note: For this assignment, you must use the provided `a7.rkt` file, which includes the structure definitions, data definitions, some sample data, and some functions to display data. To make use of these definitions and functions, you must place the `a7.rkt` file in the same directory (folder) with your assignment files, `bst.rkt` and `bstbonus.rkt` and `filedir.rkt`. At the beginning of each of your files, include the following:

```
(require "a7.rkt")
```

As a test, in your assignment file, try the expressions (*fs-print sample-fs*) and (*bst-print sample-bst*), which use the provided print function to display the sample data.

For your examples and tests, **do not** use our sample data from `a7.rkt`. You may reuse the examples provided in the question specifications below but you should ensure you have an appropriate number of examples and tests. We recommend that you carefully construct a few trees that you can use for testing multiple questions.

The structure definition and data definition for a *BST* are defined in the `a7.rkt` file so you **do not** need to include them in your file. It will cause an error if you try to include a structure definition using **define-struct** in your own file, as they are already defined in `a7.rkt`. If you want to include the structure and data definition in your file for easy reference, be sure to have them commented out in your code.

1. For this question, place your solution in the file `bst.rkt`. We will use the same definition for a binary search tree (*BST*) as in the notes (Module 08, Slides 30 to 47). You can assume there are no duplicate keys. Remember, **do not** include the **define-struct** in your file as code (comment it out, see above note):

   ```
   ;; A binary search tree (BST) is one of:
   ;; * empty
   ;; * a Node

   (define-struct node (key val left right))
   ;; A Node is a (make-node Num Str BST BST)
   ;; requires: key > every key in left BST
   ;; key < every key in right BST
   ```

   (a) Write a function *bst-internal-count* that consumes a *BST* and produces the total number of internal nodes (nodes with 1 or 2 children) in the *BST*.

   (b) Write a predicate *bst-bounded?* which consumes a *BST* and two *Int*s representing the lower and upper bounds for a range, in that order. The function *bst-bounded?* produces *true* if all keys in the *BST* are greater than or equal to the lower bound and less than or equal to the upper bound and *false* otherwise. You may assume that the lower bound is less than or equal to the upper bound. If the consumed *BST* is empty, produce *true*.

(c) Write a function *bst-add*, which consumes a *BST*, a *key*, and a *value* (a *Num* and a *Str*) in that order. If the key is already present in the *BST*, it produces a *BST* identical to the consumed *BST* except that the value previously associated with the key is replaced by the new value. If the key is not in the *BST*, the new tree is produced with the new node in the appropriate place.

(d) Write a function *bst→al* that consumes a *BST* and produces an association list (*AL*) that contains all entries in the tree, sorted by their keys in **ascending** order. Recall that an *AL* is a (*listof* (*list Num String*)). **Note: for full marks, you cannot compute an intermediate, unsorted** *AL* **and then sort it afterwards.**

(e) Write a function *bst-value-list* that consumes a *BST* and produces a (*listof Str*) containing all the values (*Str*s) stored in the *BST* with no duplicates. The strings in the list should be in **decreasing** order of their associated *keys* (even though the keys themselves will not appear in the list); i.e. the string with the smallest key will be the last item in the list. In the case of duplicate values, keep only the value corresponding to the smallest key. **Note: for full marks, you cannot build an intermediate list and then sort it or remove duplicates afterwards.**

(f) **[Bonus 5%]** For this question, place your solution in the file `bstbonus.rkt`. Write a function *bst-nth* that consumes a *BST* and a positive integer *n* and produces the *Str* value associated with the *n*-th smallest key in the *BST*, or returns *false* if *n* is invalid. For example, if $n = 1$, then *bst-nth* should produce the value associated with the smallest key (if it exists). **Note: you are not allowed to use *bst→al* or produce an intermediate list.**

2. For this question, place your solution in the file `filedir.rkt`. We will use the following data definitions for representing a file system (file tree) on a computer. **Do not** include the **define-struct** in your file as code (comment it out, see above note):

```
;; A FileSystem is a:
;; * Dir

;; A FileDir is one of:
;; * File
;; * Dir

;; A FDList is one of:
;; * empty
;; * (cons FileDir FDList)

(define-struct file (name size timestamp))
;; A File is a (make-file Str Nat Nat)

(define-struct dir (name contents))
;; A Dir is a (make-dir Str FDList)
```

A *FileSystem* is a single *Dir*, and a *FileDir* is a mixed data type that is either a *File* or a *Dir*. You can use *FileSystem* and *Dir* interchangeably in their contracts. A *File* structure includes the name of the file, the size of the file (in bytes) and a timestamp representing the number of seconds since Jan 1, 1970. *(Aside: try entering (current-seconds) in a DrRacket interactions window for such a count.)* A *Dir* structure has a name and a *FDList*, which can be empty.

The following constant definitions are used in the question specifications below.

```
(define file1 (make-file "oldfile" 1000 5))
(define file2 (make-file "newfile" 1000 55555555))
(define dir0 (make-dir "emptydir" empty))
(define dir1 (make-dir "onefile" (list file1)))
(define dir2 (make-dir "twofiles" (list file1 file2)))
(define dir2b (make-dir "twofiles" (list file1)))
(define dir3 (make-dir "onesies" (list dir1 dir1 dir1)))
(define fs1 (make-dir "rootdir" (list file1 dir0 dir1 dir2 dir3 file2)))
(define fs2 (make-dir "u" (list file2 dir0 dir1)))
```

(a) Provide templates for *File*, *Dir*, *FileDir* and *FDList*.

(b) Write a function *count-files* that consumes a *FileSystem* and produces the total number of files in the entire file system tree.

For example, (*count-files dir0*) produces 0, (*count-files dir3*) produces 3, and (*count-files fs1*) produces 8.

(c) A directory is empty if it does not contain any sub-directory or file. Write a function *empty-dir-exist?* that consumes a *FileSystem* and produces *true* if an empty directory exists anywhere in the filesystem tree and produces *false* otherwise.

(d) Write a function *oldest-file* that consumes a *FileSystem* and produces the file name (a *Str*) of the oldest file; i.e. the file with the smallest timestamp. For this question, you may assume that all timestamps are unique and there is a file as the first item in the *FDList* of the *FileSystem* - other directories in the *FDList* do not have this guarantee. For example, in the constant *fs2* defined above, the file *file2* is the first item in the *FDList*.

(e) For any file or directory *fd* that is contained in the file system a hierarchical name of *fd* is a string consisting of the appended names of all directories on the path from the initial directory used to define the file system to *fd* that are separated by /. Write a function *list-file-paths* that consumes a *FileSystem* and produces a (*listof Str*) that contains the hierarchical names for all of the files in the entire file system. The function should produce the files' hierarchical names in the same order as generated by the provided *fs-print* function. You may use *string-append* in this question.

For example, (*list-file-paths fs2*) will produce (*list* "u/newfile" "u/onefile/oldfile").

(f) Write a function *backup-fs* that consumes a *FileSystem* and a *timestamp* (Nat) and produces a *FileSystem* containing only the files whose timestamp is strictly less than the given *timestamp*. The backup *FileSystem* should have the same directory structure as

the consumed *FileSystem* except any directory not containing any files is removed. If there are no files to be backed up, the backup *FileSystem* should have an empty *FDList*. For example, (*backup-fs fs1* 6) will produce (*make-dir* "rootdir" (*list file1 dir1 dir2b dir3*)).

This concludes the list of questions for which you need to submit solutions. As always, do not forget to 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.*

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.