CS 135 Fall 2015
Akinyemi, Heinle, Holtby, Istead, Roegiest, Tompkins

Assignment: 7

Due:	Tuesday, November 10th, 2015 9:00pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure Structural and Structural Recursion with an Accumulator
Files to submit:	`staff.rkt`, `advanced.rkt`, `org-chart.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 **not** use the Racket functions *reverse* or *sort* in any of your solutions, unless stated otherwise.
- You may **not** create your own *reverse* or *sort* helper functions, either.
- 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 must avoid repeating the same function application multiple times as in the *list-max* example in slide 4 of module 7
- You may reuse the provided examples, but you must 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.
- 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.

You were hired by Purple Chicken Tech Company (PCTC) as a data scientist to provide them advanced data analytics. The data they provided you is about their current staff.

```
;; A Department-List is a (listof Str)

(define-struct staff-member (id name dept))
;; A Staff-Member is a (make-staff-member Nat Str Str)
;; requires: id is unique
;; (i.e., every staff-member with the same id also has the same name)

(define-struct salary (staff-id base bonus))
;; A Salary is a (make-salary Nat Num Num)
;; requires: base, bonus ≥ 0

;; A Staff-List is a (listof Staff-Member)
;; requires: elements are sorted by increasing id

;; A Salary-List is a (listof Salary)
```

Sample data used in examples:

```
(define staff1 (make-staff-member 1 "John" "Engineering"))
(define staff2 (make-staff-member 2 "Adam" "R&D"))
(define staff3 (make-staff-member 3 "Albert" "Engineering"))
(define staff4 (make-staff-member 4 "Liz" "Finance"))
(define staff5 (make-staff-member 5 "Anne" "Defence"))
(define staff6 (make-staff-member 6 "Suzy" "R&D"))
(define staff7 (make-staff-member 7 "Leslie" "R&D"))
(define staff8 (make-staff-member 8 "Josh" "Engineering"))

(define salary1 (make-salary 1 50000 10))
(define salary2 (make-salary 2 74000 250))
(define salary3 (make-salary 3 85000 10))
(define salary4 (make-salary 4 60000 0))
(define salary5 (make-salary 5 93000 100))
(define salary6 (make-salary 6 68500 0))
(define salary7 (make-salary 7 250000 0))
(define salary8 (make-salary 8 120000 0))

(define staff-list (list staff1 staff2 staff3 staff4 staff5 staff6 staff7 staff8))
(define sal-list (list salary1 salary2 salary3 salary4 salary7 salary8))
(define dept-list (list "Engineering" "Defence" "R&D" "Management"))
```

1. Purple Chicken wants you to start analyzing their staff information by providing simple and basic statistical information about their employees as follows. Place your functions in the file `staff.rkt`. Note that **for this question only**, your marks will be entirely based on correctness and **not** on your documentation or use of the design recipe, although you are strongly encouraged to follow the design recipe regardless.

   (a) Write a function *add-staff* that consumes a *Staff-List* and a *Staff-Member*. The function *add-staff* produces another *Staff-List* that adds the *Staff-Member* to the original *Staff-List*. If a *Staff-Member* with the same *id* is already in the list, the function should not change anything and produce the original *Staff-List*. Both the original *Staff-List* and the newly produced *Staff-List* must be sorted by the *id* of each *Staff-Member*.

   (b) Write a function *update-staff-info* that behaves the same as *add-staff*, except that if a *Staff-Member* with the same *id* is already in the list, the corresponding *Staff-Member* is replaced with the new *Staff-Member*.

   (c) Write a function *all-staff-info* that consumes a *Staff-List* and produces a list of strings, where each string is the concatenation of the *id*, *name*, and *dept* (with a space in-between each field) of each *Staff-Member* in the consumed list.

   Hint: You may need to use the *string-append* function to do the concatenation. For example: (*string-append* `"alpha" "bet"`) should produce `"alphabet"`. You may also need to use the *number->string* function to convert a number to a string. For example: (*number->string* 500) produces `"500"`.

   (*all-staff-info* (*list staff1 staff2 staff3*)) ⇒
   (*list* `"1 John Engineering" "2 Adam R&D" "3 Albert Engineering"`)

   (d) The CEO wants you to provide the total number of *Staff-Member*s in a given *Department-List*. Write a function *count-staff-by-dept* that consumes a *Staff-List* and a *Department-List* and produces a list of the total number of *Staff-Member* in each department in the provided *Department-List*.

   (*count-staff-by-dept*
   (*list staff1 staff2 staff3 staff4 staff5 staff6*) *dept-list*) ⇒ (*list* 2 1 2 0))

   (e) While having the ability to know the total number of staff per department is nice, the CEO also wishes for the ability to determine the average salary for a department. Accordingly, you are required to write a function *avg-salary-by-dept* that consumes a *Staff-List*, a *Salary-List*, and a *Department-List* and produces a list of the average salary for all staff members in each department that is in the *Department-List*. The salary of each staff member (as identified by their *id*) includes both the *base* and the *bonus*. If a staff salary does not exist in the *Salary-List*, you may assume the staff salary is 0. If there are no staff members in a department, the average salary should be 0.

   (*avg-salary-by-dept staff-list sal-list dept-list*) ⇒ (*list* 255020/3 0 324250/3 0)

2. When you have completed the initial basic analysis, your CEO wants you to provide advanced analytics. You will combine the information in the *Staff-List* with the information in the *Salary-List* to provide important and sensitive information about your company as follows...

   In this set of analytics, you are required to provide fast searching functionalities that rely on a binary search tree (BST). This BST is similar to the binary search tree from the lecture notes, but the keys and values have different types.

   (**define-struct** *staff-node* (*staff left right*))
   ;; A Staff-Node is a (make-staff-node Staff-Member Staff-BST Staff-BST)
   ;; requires: ids of all Staff-Member on the left subtree are smaller than the id of Staff-Member
   ;; ids of all Staff-Member on the right subtree are larger than the id of Staff-Member

   ;; A Staff-BST is one of:
   ;;* empty
   ;;* Staff-Node

   Place your functions in the file `advanced.rkt`.

   (a) Define a function *add-staff-bst* that consumes a *Staff-BST* and a *Staff-Member* and produces a new *Staff-BST* with the *Staff-Member* added. You may assume that no *Staff-Member* with the same *id* is already in the *Staff-BST*.

   (b) Define a function *create-staff-bst-from-list* that consumes a list of *Staff-Member* and produces a corresponding *Staff-BST*. You may assume each *Staff-Member* in the list of *Staff-Member* has a unique id. Your *create-staff-bst-from-list* must use pure structural recursion, which means that the *first Staff-Member* in the list will be the last added.

   (c) The forecasting for PCTC's next fiscal quarter is looking particularly poor. As a result, the owners of the company decided to restructure the company and downsize (e.g., by firing some of the staff). They decided to fire all staff members that earn more than a certain threshold amount. For example if they chose an amount of $100,000 as the threshold amount, all staff members that earn greater than $100,000 will be fired.

   Write a function *who-to-fire* that consumes a *Salary-List*, a *Staff-BST*, and a positive *Num* (representing the threshold amount) and produces a (sorted) *Staff-List* of everyone that should be fired. If there is no entry in the Salary List, the person should not be fired. For example:

   (**define** *staff-bst* (*create-staff-bst-from-list staff-list*))
   (*who-to-fire sal-list staff-bst* 100000) ⇒
   (*list* (*make-staff-member* 7 "Leslie" "R&D")
   (*make-staff-member* 8 "Josh" "Engineering"))

   (d) Of course, just knowing which *Staff-Member* to fire is not sufficient. The goal is to remove them from the *Staff-BST*. Write a function *remove-from-bst* that consumes a

Staff-BST and a staff id number and removes the *Staff-Node* corresponding to that id number from the *Staff-BST*. If no corresponding *Staff-Node* exists, the original *Staff-BST* should be produced.

You should use the immediate successor algorithm to remove the *Staff-Node* from the *Staff-BST*. Next is a brief sketch of the algorithm to guide you. Once you find the *Staff-Node* to be removed, you will be in one of three situations: the node is a leaf; the node has one child; the node has two children. The first two cases are straightforward... when the node is a leaf node, you should replace the node with an empty *Staff-BST*. When the node has one child, you should replace the node with its child.

The situation is more complex when the node has two children. In this case, you should replace the node by its immediate successor (the *Staff-Node* with minimum id in its right subtree). However, you will now have a pair of duplicate nodes in the *Staff-BST*, so you should remove the immediate successor from the right subtree. Note that you are guaranteed that the immediate successor node will fall into one of the first two cases[1].

Hint: You may recall from class that the BST node with the minimum key is the leftmost node.

3. After completing the initial restructuring of PCTC, the CEO now wants you to generate a new organization chart for the company. An Org-Chart is a tree where the leaf nodes are staff members with no subordinates, and internal nodes are supervisors who provide supervision for other staff members.

```
(define-struct supervisor (id subordinates))
;; A Supervisor is a (make-supervisor Nat (listof Org-Chart))
;; requires: id values are unique

;; An Org-Chart is one of:
;;* Nat
;;* Supervisor
```

   Place your functions in the file `org-chart.rkt`.

   (a) Write a function *direct-reports* that consumes an *Org-Chart* and an *id*, and produces a list of all staff *id*s that are supervised (directly or indirectly) by the *id*. The order of the list is immaterial. If the *id* does not exist or does not supervise any other staff, the function should produce an empty list. Note that the *id* should not appear in the list.

   (b) In the summer months, when most staff want to go on their vacation, they require a supervisor to approve vacation requests. A vacation request can only be approved by direct or indirect supervisors as indicated in the organization chart.

   Write a function *vacation-approval* that consumes an *Org-Chart* and a staff id and produces a list of the ids of all supervisors who can approve the vacation request. The

[1] It is relatively trivial to prove this.

CEO does not have a supervisor (being at the root of the organization chart), so *vacation-approval* will produce an *empty* list (but do not worry about the CEO as they do not require approval for their own vacations). All other staff members require a supervisor to approve their vacation requests. The order of the produced list is immaterial. If the *id* does not exist, the function should produce an empty list.

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.*

The material below first explores the implications of the fact that Racket programs can be viewed as Racket 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.