CS 135 Fall 2016
Graham, Holtby, Petrick, Sakhnini

Assignment: 7

Due:	Tuesday, November 8th, 2015 9:00pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure Structural and Structural Recursion with an Accumulator
Files to submit:	`inventory.rkt`, `orgchart.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 function *reverse* in any of your solutions, unless stated otherwise.
- You may **not** create your own *reverse*, either.
- You may **not** create your own *sort* function.
- You may use any other list functions discussed in the notes up to and including module 8, 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.
- You do **not** need to include tests and examples for your helper functions, but you must ensure that the tests you do write will fully test any helpers used (i.e. you must not have any black highlighting other than in templates).
- 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.

1. You have been hired by the πr^2 baking company to write an inventory system for keeping track of blueberries, napkins, and other consumables.

   The CEO has just learned about binary search trees and so insists that you use one, with the following data definition:

```
(define-struct invnode (name count left right))
;; An InventoryTree is one of:
;; * empty
;; * (make-invnode Str Nat InventoryTree InventoryTree)
;;    requires: All names in the left subtree are string<? name
;;                 All names in the right subtree are string>? name
```

   (a) Write a function *item-count* that consumes the name of an item (a string) and an InventoryTree and produces the amount of that item in inventory. Note: If the item is not in the tree, then there are none (0).

   (b) Write a function *add-item* that consumes the name of an item (a string) an amount (positive integer) and an InventoryTree, and produces an InventoryTree equal to the consumed tree but with the count of the consumed ingredient increased by the consumed amount.

   Note: If the tree did not previously contain that item, then the tree produced should contain a new node with that item and the consumed amount.

   (c) Write a function *use-item* that consumes the name of an item, an amount (positive integer), and an InventoryTree, and produces a copy of the consumed tree except that the node containing the consumed item has a count that is decreased by the consumed amount. If the tree does not contain enough of that item (or does not contain it at all) then the function produces a tree identical to the consumed tree. Note: If this results in an item with a count of 0, do not remove the node from the tree! A node with a count of 0 indicates that the bakery is out of something, and that is useful information.

   (d) Write a function *out-of* that consumes an InventoryTree and produces a list of items that are in the tree, but have a count of 0. This list must be in sorted order (according to *string<?*).

   For example, if the bakery is out of apples, napkins, and flour, it would produce (*list* `"apples" "flour" "napkins"`)

   Reminder: You may **not** create your own sort function.

   (e) A recipe is a (*listof* (*list Str Int*)) where the strings are ingredient names (and are unique) and the integers are positive values representing the amount of the ingredient required.

   Write a function *can-make?* that consumes a recipe and an InventoryTree, and produces *true* if the InventoryTree has enough ingredients to make the recipe, and *false* otherwise. For example, if the recipe is (*list* (*list* `"apples" 2`') (`"butter" 1`)) then the function will produce *true* if the InventoryTree contains a count of 2 or more for "apples" and 1 or more for "butter". If an ingredient is not in the tree, then the function produces *false*.

(f) **[Bonus 5%]** Write a function *out-of/bonus* that is identical to *out-of* but which does not use any built-in functions that produce lists other than *cons*, and does not use any built-in functions that consume lists other than *empty?*. This restriction includes *first* and *rest*! You may use helpers that consume and/or produce lists, but those helpers must also obey these restrictions.

Note: If your *out-of* function already obeys these restrictions you may use it as a helper function, or copy and paste the code.

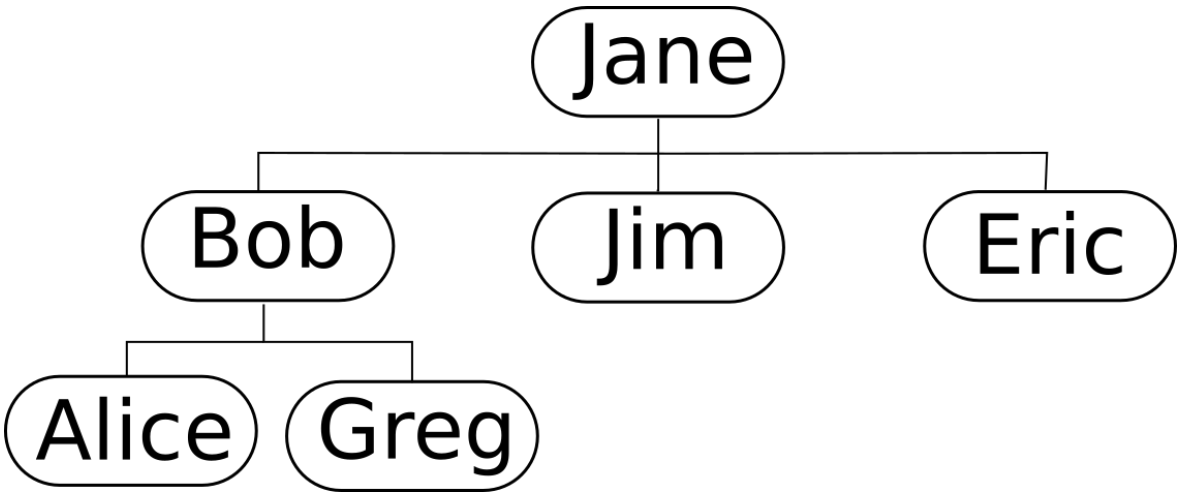
Submit your code for this question in a file named `inventory.rkt`

2. Now that the inventory has been taken care of, you must now write some functions for dealing with the bakery's growing list of employees. An organizational chart or "org chart" can be represented with a general tree. Use the following data definition:

```
(define-struct orgnode (name employees))
;; An OrgChart is a (make-orgnode Str (listof OrgChart))
;; requires: name is unique

(define my-orgchart (make-orgnode "Jane"
                                  (list (make-orgnode "Bob"
                                                      (list (make-orgnode "Alice" empty)
                                                            (make-orgnode "Greg" empty)))
                                        (make-orgnode "Jim" empty)
                                        (make-orgnode "Eric" empty))))
```

my-orgchart looks like:



Every employee **supervises** the employees directly below them in the tree, and this person is called a **supervisor**. In the example tree, Bob supervises Alice and Greg, while Jim doesn't supervise anybody. Put differently, Bob is the supervisor of both Alice and Greg, and Jim isn't anybody's supervisor.

Every employee is **responsible** for all employees below them in the tree (directly or not). In the example tree, Jane is responsible for all other employees, while Greg is responsible for nobody.

(a) Write the templates for a function that consumes an OrgChart and a function that consumes a (*listof OrgChart*).

(b) Write the function *responsible-count* that consumes a string (an employee's name) and an OrgChart, and produces the number of other employees the consumed employee is responsible for. If the employee is not in the chart, the function produces 0.

Reminder: An employee is responsible for all of the employees below them in the OrgChart. An employee is not responsible for themselves.

Example: (*responsible-count* `"Jane"` *my-orgchart*) $\Rightarrow$ 5

Submit your code for this question in a file named `orgchart.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.*

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.