CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 7

Due:	Tuesday, Nov 12th, 2019 9:00 pm
Language level:	Beginning Student with List Abbreviations
Files to submit:	`bstd.rkt`, `itree.rkt`, `winsys.rkt`
Warmup exercises:	HtDP 14.2.1, 14.2.2
Practice exercises:	HtDP 14.2.3, 14.2.4, 14.2.6

- **Make sure you read the OFFICIAL A07 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 06 carry forward.
- This assignment covers material up to and including Module 11, slide 48.
- The **only** list functions you may use are: `cons`, `cons?`, `empty`, `empty?`, `first`, `second`, `rest`, `list` and `append`.
- You may **only** use the functions that have been discussed in the lecture slides, unless explicitly allowed or disallowed in the question.
- You may **not** use `reverse`, abstract list functions or `local` on this assignment.
- Except for Question 3a, templates are not required, but you may find including them helpful.
- Solutions will be marked for both correctness [80%], test cases [10%] and style [10%]. Follow the guidelines in the Style Guide, pages 1-17 (skipping section 3.7.4 for now).
- We will provide some starter code for each question.

Overview

A windowing environment in computer science is the software that manages different parts of the display. In Windows 10 it is called the Desktop Window Manager and in macOS it is called Quartz Compositor.

Both of these environments implement the WIMP (windows, icons, menus, pointer) paradigm. An additional complexity that both of these operating systems support is allowing for multiple programs to run (or appear to run) at the same time, so the windowing system must manage multiple (possibly overlapping) windows.

Our Goal

The task we will be investigating is when a user clicks the mouse on the screen, determining which window and which part of the window they have clicked on. We will simplify this task by assuming every component of a window is a rectangle (which includes squares).

A Window can have many components such as a Menu Bar (a rectangular area) that can be further subdivided into Menu Items (which are smaller rectangles), a maximize button, a minimize button and a close button. We will use symbols to identify each of these components, e.g. `'Close`, `'Maximize`, and `'Minimize`. We will use the symbol `'None` to identify the region outside of a window; i.e. a rectangle is inside a window if and only if it is not associated with the label `'None`.

Our Approach

We will start with some simple tasks (which will be in 1 dimension) and then work our way up to identifying which part of a window a user has clicked on (which will be in 2 dimensions).

1. **An Augmented Binary Search Tree Dictionary** (`BSTD`) [25% Correctness]

   For this question we will be using the following data definition for a binary search tree dictionary which associates Natural numbers to Symbols.

```
(define-struct node (key val left right))
;; A Node is a (make-node Nat Sym BSTD BSTD)
;; requires: key > every key in left BSTD
;;           key < every key in right BSTD

;; A binary search tree dictionary (BSTD) is one of:
;; * empty
;; * Node
```

   (a) A range search in a BSTD considers all the keys within a range. First create a function that produces the number of keys in the given range.

```
;; (range-count dict low high) produces the number of keys that
;; are >= low and < high in dict.
;; range-count: BSTD Nat Nat -> Nat
;; requires: low < high
```

   For example in the BSTD illustrated in Figure 1, we would have the following.

```
(check-expect(range-count root1 20 30) 0)
(check-expect(range-count root1 10 12) 0)
(check-expect(range-count root1 10 13) 1)
(check-expect(range-count root1 12 13) 1)
(check-expect(range-count root1  4  8) 4)
(check-expect(range-count root1  0 15) 9)
```

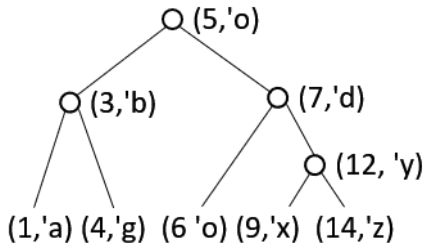

Figure 1

(b) Now create a function that produces a list of all the corresponding values in that range ordered from the value with the lowest key to the value highest key. Hint: carefully consider the order that the recursion is done. You may reorder the sequence of recursive calls in the template as long as the base case is always first.

You **may use** the list function `append` for this question but you **cannot use** `reverse`.

```
;; (range-query dict low high) produces a list of values whose
;; keys in the dict are in the range >= low and < high. The
;; list of values produced are in ascending order by their key.
;; range-query: BSTD Nat Nat -> (listof Sym)
;; requires: low < high
```

For example, in the BSTD illustrated in Figure 1, we would have the following.

```
(check-expect(range-query root1 20 30) '() )
(check-expect(range-query root1 10 12) '() )
(check-expect(range-query root1 10 13) '(y))
(check-expect(range-query root1 12 13) '(y))
(check-expect(range-query root1  4  8) '(g o o d))
(check-expect(range-query root1  0 15) '(a b g o o d x y z))
```

The data definitions, the example in Figure 1, the function purposes and contracts will be provided in a starter file that you can download, called `bstd.rkt`. Add your code to this file and submit it as your solution to Q1.

2. **Interval Trees** (`ITrees`) [10% Correctness]

For this question we are breaking up the natural numbers into intervals. Each interval will have a symbol associated with it. Consuming a Natural number `n`, use a binary search tree to find which interval `n` is in and produce the symbol associated with that interval.

We will call this an `ITree` (for interval tree). The interior nodes will be called `BNode` (for boundary nodes). The leaves are symbols.

```
(define-struct bnode (val left right))
;; A BNode is a (make-bnode Nat ITree ITree)
;; requires: val > every val in left ITree
;;           val < every val in right ITree

;; An ITree (Interval Tree) is one of:
;; * a Sym (a leaf)
;; * a BNode (a boundary node)
```

Given a natural number `n` and a BNode with value `v`

- if `n` $<$ `v`, then the interval occurs in the left subtree;
- if `n` $\geq$ `v`, then the interval occurs in the right subtree.

When a leaf is reached, produce the symbol that is associated with that leaf. For example, for the `ITree` in Figure 2, consuming

- 0, 1 or 2 will produce `'None`
- 3, 4 or 5 will produce `'a`
- 6 or 7 will produce `'b`
- 8 will produce `'c`
- 9, 10, ... will produce `'None`

Create a function called `it-lookup` (which consumes an `ITree` and a `Nat`) that produces the symbol associated with the interval that the `Nat` is found in.

```
;; (it-lookup it n) produces the symbol from the it
;; that is associated with the interval that contains n
;; it-lookup: ITree Nat -> Sym
```

For example, for the ITree in Figure 2 the following should all pass.

```
(check-expect (it-lookup n6 0) 'None)
(check-expect (it-lookup n6 2) 'None)
(check-expect (it-lookup n6 3) 'a)
(check-expect (it-lookup n6 5) 'a)
(check-expect (it-lookup n6 6) 'b)
(check-expect (it-lookup n6 7) 'b)
(check-expect (it-lookup n6 8) 'c)
(check-expect (it-lookup n6 9) 'None)
(check-expect (it-lookup n6 10) 'None)
```

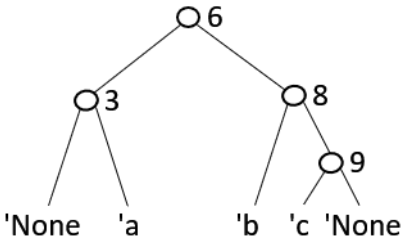

Figure 2

The data definitions, example from Figure 2, function purposes and contracts will be provided in a starter file that you can download, called `itree.rkt`. Add your code to this file and submit it as your solution to Q2.

3. **Finding Rectangles in a Window** (`RTree`) [45% Correctness]

   For this question we will be moving from 1 dimension to 2 dimensions, breaking up the XY plane (pixels on a screen) into different rectangles. Each rectangle will have a symbol

associated with it. Consuming a `posn` (position on the screen) you will use a binary search tree to find which rectangle the `posn` is in and what symbol is associated with that rectangle. We will call this an `RTree` (for rectangle tree).

There will be two types of interior nodes in this tree. One type is called an `XNode` which will divide the window into two parts, one with *x* values *less than* the `XNode` value (left) and one with x values *greater than or equal to* the `XNode` value (right). Similarly, a `YNode` will divide the window into two parts: those with *y* values *less than* the `YNode` value (below) and those with *y* values *greater than or equal to* the `YNode` value (above). Again, the leaves are symbols, which represent different rectangles that make up the window.

Our goal is to be able to represent something similar to the window that DrRacket uses, as in Figure 3.

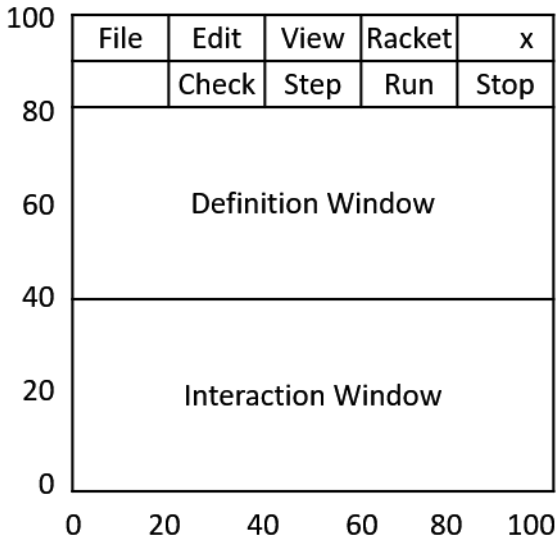

Figure 3

The data definitions for an `RTree` is as follows.

```
;; An RTree (Rectangle Tree) is one of:
;; * Sym (a leaf)
;; * XNode (a boundary node for an x value)
;; * YNode (a boundary node for a y value)

(define-struct xnode (val left right))
;; An XNode is a (make-xnode Nat RTree RTree)
;; requires: val > every xnode-val in left RTree
;;           val < every xnode-val in right RTree

(define-struct ynode (val below above))
;; A YNode is a (make-ynode Nat RTree RTree)
;; requires: val > every ynode-val in below RTree
;;           val < every ynode-val in above RTree
```

(a) Templates

Create the templates for `Rtree`, `XNode` and `YNode`, namely `rtree-template`, `xnode-template` and `ynode-template`.

(b) `rt-lookup`

Create a function, `rt-lookup` (which consumes an `RTree` and a `Posn`) and produces the symbol associated with the rectangle in the `RTree` that contains that `Posn`.

```
;; (rt-lookup rt pos)
;; produces the symbol from rt that is associated with the
;; rectangle that contains pos
;; rt-lookup: RTree Posn -> Sym
;; Examples:
(check-expect (rt-lookup 'Desktop (make-posn 2 2))
              'Desktop)
(check-expect (rt-lookup (make-xnode 5 'left 'right) (make-posn 2 2))
              'left)
(check-expect (rt-lookup (make-xnode 5 'left 'right) (make-posn 5 5))
              'right)
(check-expect (rt-lookup (make-ynode 5 'bottom 'top) (make-posn 2 2))
              'bottom)
(check-expect (rt-lookup (make-ynode 5 'bottom 'top) (make-posn 5 5))
              'top)
```

Besides the fairly eloborate `RTree` in Figure 3, we will provide you with two simpler test cases. One that represents a square with the symbol `'None` produced for any `Posn` outside the square.

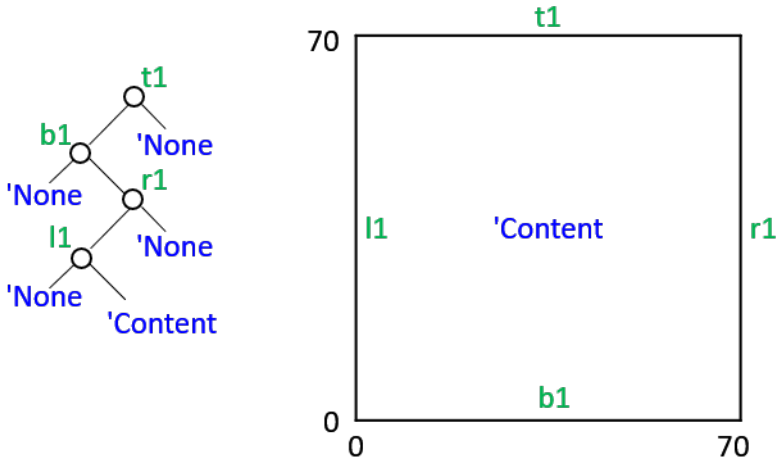

Figure 4

The second one will represent the window shown in Figure 5. This is just two modifications to Figure 4 where two nodes are added: a `YNode` called `y60` to separate `'Content` from from the upper part of the window and an `XNode x60` to separate `'Menu` from `'x`.

In both of these test cases, if nodes start with 't', 'b', 'l' or 'r' they are referring to the top, bottom, left and right boundaries of the window.

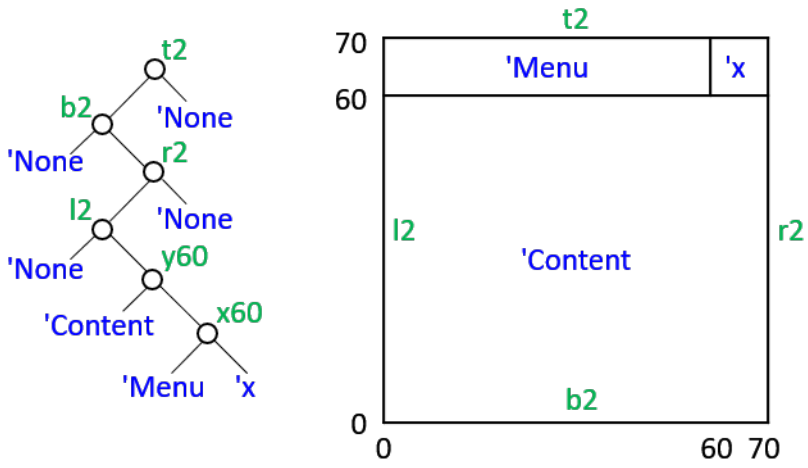

Figure 5

For your testing, you can use these two trees or simple variations of them.

(c) `rt-max-x`

It can be useful to know the maximum value of any `XNode` in the `RTree`. Create a function that performs this task. Because an `RTree` can possibly have no `XNodes` (e.g. a single leaf) `rt-max-x` can also produce `'None`.

```
;; (rt-max-x rt)
;; produces the maximum value of any XNode in the rt
;; or produce 'None if there are no XNodes in the rt
;; rt-max-x: RTree -> (anyof Nat 'None)
;; Examples:
(check-expect (rt-max-x 'a) 'None)
(check-expect (rt-max-x (make-ynode 8 'a 'b)) 'None)
(check-expect (rt-max-x (make-xnode 8 'a 'b)) 8)
```

(d) `Win` and `move-win`

In order to allow for multiple windows, each window will have a name, which will be a natural number called its `wid` (Window ID) and a corresponding `RTree`. Use the following data definition.

```
(define-struct win (wid rtree))
;; A Win (Window) is a (make-win Nat RTree)
;; requires: points outside the window are labelled 'None
```

Create a function `move-win` which consumes a `Win`, an `x` value and a `y` value (which could be negative) and produces a new `Win` with all the values in `XNodes` incremented by `x` and all the values in the `YNodes` by incremented by `y`. Since windows can be moved partially off the screen, we allow for negative numbers. In order to keeps things simple, there is no need to check if the `x` and `y` values are too large or too small.

```
;; (move-win wi x y)
;; moves the Win wi, x pixels along the x-axis
;; and y pixels in the y-axis.
;; move-win: Win Int Int -> Win
```

(e) `WinSys` and `winsys-lookup`

A WinSys (window system) is a list of (possibly overlapping) windows with the first window in the list being the top window on the screen. The data definition is as follows.

```
;; A WinSys is one of:
;; * empty
;; * (cons Win WinSys)
```

Create a function (which consumes a `WinSys` and a `Posn`) that produces the `wid` (window id) of the top window that contains that point. It also produces the symbol that corresponds to the rectangle in that window. Recall that if a point is associated with a rectangle labelled `'None` for a window, then that window does not contain that point. If no window contains that point then produce the value `(list 0 'None)`.

```
;; (winsys-lookup ws pos)
;; produces the wid of the top window in ws that contains pos
;; and the Sym corresponding to the rectange that contains pos
;; winsys-lookup: WinSys Posn -> (list Nat Sym).
```

The data definitions, examples, and function purposes and contracts will be provided in a starter file that you can download, called `winsys.rkt`. Add your code to this file and submit it as your solution to Q3.

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.