Assignment: 7
Due: Tuesday, Mar 19th, 2019 9:00pm
Language level: Intermediate Student
Allowed recursion: Pure Structural and Structural Recursion with an Accumulator, unless otherwise specified
Files to submit: pizza-tree.rkt, drawings.rkt, prim-bonus.png, prim-bonus.rkt, shape-bonus.png, shape-bonus.rkt
Warmup exercises: HtDP 14.2.1, 14.2.2, 15.1.1, 15.1.3, 18.1.1, 18.1.2, 18.1.3, 18.1.4, 18.1.5
Practice exercises: HtDP 14.2.3, 14.2.4, 14.2.6, 15.1.2, 15.1.4, 18.1.5

- 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.
- You may use the following list functions and constants: cons, cons?, empty, empty?, first, second, third, fourth, fifth, sixth, rest, list, append, member? length, string->list, list->string. You may not use reverse unless specified in the question.
- Some correctness marks will be allocated for avoiding exponential blowups, as demonstrated by the max-list example in slide 07-6. (But note that not every repeated function invocation leads to an exponential blowup.) For some functions, we will provide basic tests on relatively large arguments. If your function times out when running these basic tests you may need to see whether you are running into exponential blowup.
- You may not use abstract list functions on this assignment. You may use local.
- It will be very helpful to design examples and tests before writing your function implementations.
- Solutions will be marked for both correctness [80%] and good style [20%]. Follow the guidelines in the Style Guide.

Pro Tip: Sketch out solutions to these problems before attempting to implement them in DrRacket. Otherwise you will very likely end up stuck and frustrated.

1. [5%] Perform the assignment 7 questions using the online evaluation “Stepping Problems” tool linked to the course web page and available at

The instructions are the same as those in assignment 1. If you get stuck, please do not post questions to Piazza asking for the next step, since such questions are considered to be violations of our academic integrity policy. If you are really stuck see an ISA or instructor in person.

The use of `local` will involve lifting definitions to global scope. Read the instructions for each question carefully; often you will be asked not to resubmit simplified constants in your steps. Don’t get stuck because you leave those constants in.

2. [30%]

You are managing the inventory for Racket Pizza. Currently, inventory is kept track of using an association list

```
;; An Inventory List (InvList) is a (listof (list Str Nat))
;; requires: each Str is unique
```

The `Str` represents the name of the item, e.g. "mushroom" and "large pizza box". The `Nat` represents how many are currently in stock.

You have been asked to convert their inventory into a Binary Search Tree, using the following definitions

```
(define-struct invnode (item count left right))
;; An InventoryTree (InvTree) is one of:
;; * empty
;; * (make-invnode Str Nat InvTree InvTree)
;;   requires: item is string>? all items in left InvTree
;;   item is string<? all items in right InvTree
```

Note: the `string<?` and `string>?` functions determine which string comes first in alphabetical order\(^1\). For example, `(string<? "cat" "catastrophe") => true` and `(string<? "alpha" "beta") => true.

Example constants:

```
(define example-ilist '(("mushroom" 50) ("pepperoni" 25) ("tomato sauce" 13) ("broccoli" 0)))
```

```
(define example-itree (make-invnode "pepperoni" 25
 (make-invnode "broccoli" 0 empty
 (make-invnode "mushroom" 50 empty empty))
 (make-invnode "tomato sauce" 13 empty empty))
```

The tree shown can be visualized as

\(^1\)Actually, *lexicographical* order, but close enough
(a) Write a function `invtree-count` that consumes an `InvTree` and a string and produces the current count of that item. If the tree does not contain a matching key, then the count is 0.

Example

```scheme
(check-expect (invtree-count example-itree "pepperoni") 25)
(check-expect (invtree-count example-itree "pineapple") 0)
```

(b) Write a function `invlist-count` that consumes an `InvList` and a string and produces the current count of that item. If the list does not contain a matching key, then the count is 0.

Since the example list and example tree have the same values, the examples are also the same (other than the type of data structure).

(c) Write a function `invtree-add` that consumes an `InvTree`, a string, and a positive integer, and produces an `InvTree` where the count of the named item has been increased by the consumed integer. Remember: If the item (the string) is not in the tree currently, then it’s count is 0. You will need to add a new node, while obeying the BST’s ordering property.

For example, if we added some pineapple to the example tree, there is only one place to add it (without needing to restructure the entire tree).

Try it yourself: `(and (string<? "pepperoni" "pineapple") (string>? "tomato sauce" "pineapple"))`

(d) Write a function `invtree-low?` that consumes an `InvTree` and a natural number, and produces true if the tree contains at least one item with a count less than or equal to the consumed number. For example, if the number was 0, it will produce true if and only if the `InvTree` is out of something.

(e) Many of Racket Pizza’s existing functions consume `InvLists`. Until you can rewrite them, it would be helpful to convert between the association list and BST formats. Write
a function `invtree->invlist` that consumes an `InvTree` and produces the corresponding `InvList`. The produced list must be in sorted order (according to the keys).

**Hint:** `append` is very useful here! Remember, for a given node, everything to its left should be before, and everything to its right should be after (that’s what a BST’s ordering property means).

Example:

```scheme
(check-expect (invtree->invlist example-itree)
 '(('broccoli" 0) ('"mushroom" 50) ('"pepperoni" 25) ('"tomato sauce" 13)))
```

Place your solutions in `pizza-tree.rkt`

3. [45%] This question involves images. In particular, you will create a "drawing compiler" that consumes high-level specifications of images, and converts them to low-level primitives. These primitives will be consumed by a function we provide, which will generate an image on the screen.

To do this question you will need to download `a07drawinglib.rkt` from the Assignments page. (You may need to right-click the file and “Save Target As”, as you did for Assignment 0). Download this file to your working folder (that is, the same folder containing your program). Then at the top of your program, include the following line to use the library:

```scheme
(require "a07drawinglib.rkt")
```

This library provides the following low-level structures and types, which will be the “drawing primitives” we use:

```scheme
;; A Coordinate is a (make-posn Int Int)

;; An ImageColor is a Str
;; requires: the Str is from the racket/draw colour database:
;; https://docs.racket-lang.org/draw/color-database____.html

(define-struct prim-circle (center radius color))
;; A PrimCircle is a (make-prim-circle Coordinate Nat ImageColor)

(define-struct prim-triangle (p1 p2 p3 color))
;; A PrimTriangle is a (make-prim-triangle
;; Coordinate Coordinate Coordinate ImageColor)

;; A PrimElement is (anyof PrimTriangle PrimCircle)

Note the Americanized spellings of “color” and “center” in these definitions.

The library also provides the Image data type. This is a value that cannot be simplified. Dr.Racket will automatically draw it for you! (Don’t worry about the details).

**DO NOT COPY AND PASTE IMAGE VALUES FROM THE INTERACTION WINDOW.**
The top-left corner of an image has coordinate (0, 0). The centers of circles are defined relative to this, and so are the points of triangles. Note that larger $x$ values in a coordinate go right along the image, and larger $y$ values go down.

In the following figure, there is a circle centred at (40, 20) and a triangle with coordinates (30, 70), (80, 40), and (75, 80).

![Diagram of circle and triangle](image)

To render a list of PrimElements into an Image, The a07drawinglib.rkt library provides the following function:

```
;; (render-image canvas-size prim-shapes) produces an Image on an image of size canvas-size containing the list of shapes listed by prim-shapes.
;; render-image: Posn (listof PrimElements) -> Image
;; requires: canvas-size is a (make-posn Nat Nat)
```

The x component of canvas-size represents horizontal (the width of the canvas) and the y component represents vertical (the height of the canvas).

The order of elements in the list of PrimElements is important. PrimElements at the front of the list are rendered on top of elements appearing later. For example, overlap-prims below consists of a small white circle on top, with a dark blue triangle underneath it, and a chocolate-coloured circle on the bottom.

```
(define overlap-prims (list
  (make-prim-circle (make-posn 30 30) 4 "White")
  (make-prim-triangle
    (make-posn 10 40) (make-posn 50 40) (make-posn 30 10) "DarkBlue")
  (make-prim-circle (make-posn 30 30) 15 "Chocolate")))
```

Rendering the above produces the following Image:
(a) Define a constant `prim-picture` which defines a list of `PrimElements` that makes a pretty picture when rendered. The picture should contain between 3 to 5 circles and 3 to 5 triangles. Be creative! The picture should fit in a 300 x 300 pixel canvas.

**Bonus (3%)**: Although your drawing does not need to be fancy or interesting, we’ll award a bonus of 3% for the best drawings, as judged by the ISAs and their subjective, personal tastes. If you would like to show your drawing to the ISAs (and be considered for this bonus), please copy your `prim-picture` constant to a new Racket file called `prim-bonus.rkt`, and also upload an image of it in the file `prim-bonus.png`. To save a rendered image as a PNG file, use the `save-image` function provided by `a07drawinglib.rkt`. For example:

```racket
(save-image (render-image (make-posn 60 60) overlap-prims) "overlap-prims.png")
```

saves the `overlap-prims` constant into a file called "overlap-prims.png". The file will be saved in the same folder as your Racket file.

(b) Write the function `rect`, which constructs a rectangle out of two `PrimTriangles`, one on top of the other. The function consumes a `Coordinate` representing the offset of the rectangle from the the top left corner (0,0), a `Coordinate` representing the width and height of the rectangle (both of which should be `Nats`), and an `ImageColor`. The function produces a list of `PrimTriangles` representing the rectangle.

For example,

```racket
(render-image (make-posn 100 100)
  (rect (make-posn 20 30) (make-posn 23 54) "Red"))
```

indicates that the rectangle’s top left corner is at (20,30), and the rectangle has a width of 23 pixels and a height of 54 pixels. When rendered, the rectangle produces the following image:

Notice that the displayed rectangle has a thin diagonal line. This is okay.

There are many different ways to lay out `PrimTriangles` to create a rectangle. For the ease of our marking scripts, please generate your rectangles using the following guidelines:

- Generate two `PrimTriangles`. 
• Each generated PrimTriangle should have a corner at the top left corner of the rectangle. This should be the first coordinate. Subsequent coordinates should be specified in clockwise order.
• The “top” PrimTriangle should appear first in the list.

For example, the “top” triangle in the above picture has coordinates (20,30), then (43,30), then (43,84) in that order.

Specifying images in terms of low-level triangles and circles gets tedious. For that reason, we can use a nested list structure to specify images at a (moderately) higher level.

Drawings will be made of shapes. We can think of every Shape as being drawn relative to an origin point. This difference between where the shape is drawn and its origin point is called its offset. Specifying an offset allows you to place shapes in different positions of an image. Sometimes the origin point will be (0,0), but this won’t necessarily be true for nested shapes.

We will define four kinds of shapes: circles, triangles, rectangles, and component shapes, which are made up of other shapes grouped together. Here is how offsets work for these different types:

• A circle measures its offset from its origin point to the centre of the circle.
• A rectangle measures its offset from its origin point to the top left corner of the rectangle (that is, the corner closest to (0,0)).
• A triangle measures its offset in a complicated way. First, imagine the smallest rectangle that will enclose that triangle. This is a “bounding rectangle”. The offset of the triangle is specified relative to the top-left corner of that bounding rectangle.
• A component can be thought of as being drawn in an imaginary rectangle. Shifting the imaginary rectangle shifts the component on the image. This imaginary rectangle measures its offset from the origin point to the top left corner of the rectangle.

The images below demonstrate offsets in action:

The green circle has an offset at its centre, which is (50,25). The pink rectangle has an offset at its top left corner, which is (60,55). The turquoise triangle’s bounding box is shown as a dashed rectangle. The offset for the triangle is (10,40).

We can now specify data definitions for higher-level drawings.
A Point is a (list Int Int)
An Offset is a Point

A ShapeID is a Sym
requires: ShapeID is not 'circle, 'triangle, 'rectangle, 'component

A Shape is one of:
- (list 'circle ShapeID radius ImageColor)
- (list 'triangle ShapeID Point Point Point ImageColor)
- (list 'rectangle ShapeID width height ImageColor)
- (list 'component ShapeID Picture)
requires: radius,width,height are Nat
The ShapeID of a component does not appear in its Picture
when recursively expanded.
(i.e. there are no circular definitions)

A Picture is a (listof (list Offset ShapeID))

A ShapeList is a (listof Shape)
requires: every ID in the ShapeList is unique

A BundledDrawing is a (list width height Picture ShapeList)
requires: width, height are Nat
Every ShapeID in the Picture occurs in ShapeList.

Here is an example of a ShapeList:

(define summer-shapes '((circle top-scoop 10 "Pink")
(circle bottom-scoop 10 "LightBlue")
(component ice-cream
  (((0 40) cone)
   (((10 35) bottom-scoop)
     (((10 25) top-scoop))))
  (triangle cone (0 0) (20 0) (10 50) "Burlywood")
(circle moon 35 "Light Gray")
(circle sun 40 "Yellow")))

This defines a number of Shapes, including one component made of multiple shapes. Nothing prevents components from containing other components, but this ShapeList does not have this. Every Shape has a unique identifier, which is a symbol.

Here is an example of a Picture:

(define ice-cream-pic '(((10 50) ice-cream)
  ((70 20) sun)
  ((130 30) ice-cream)))

The idea is that a picture is made up of a number of shapes, with an offset. As with PrimElements, the order of elements in the Picture is important: the ice cream cone with an offset of (10,50) is rendered last (that is, on top of the other shapes) because it appears first in the list.
Finally, here is a BundledDrawing that puts the Picture and ShapeList together with an image size:

```scheme
(define ice-cream-drawing (list 200 150 ice-cream-pic summer-shapes))
```

Note that this list uses `list` notation, not quote notation, because `ice-cream-pic` and `summer-shapes` are constants, not symbols.

To make accessing elements from Shapes and Points easier, we have defined a number of “accessor functions” in `a07drawinglib.rkt` for you, similar to the accessors described in slide 06-46. They are `point-x`, `point-y`, `shape-type`, `shape-id`, `component-picture`, `circle-radius`, `circle-color`, `triangle-p1`, `triangle-p2`, `triangle-p3`, `triangle-color`, `rect-width`, `rect-height`, and `rect-color`. See the library for function documentation.

(c) Define a ShapeList constant `fun-shapes`. This ShapeList should contain at least three `component` Shapes, including at least one `component` that contains at least one other `component`.

Next, define a Picture constant called `fun-pic` which uses elements of `fun-shapes` to construct a fun drawing.

Finally, define a BundledDrawing constant called `fun-drawing` which uses `fun-shapes` and `fun-pic`.

You can use these constants to test the other functions you write.

(d) Write `get-picture-ids`, which consumes a Picture and a ShapeList, and produces a list of ShapeIDs that occur in the Picture, with no duplicates.

For example, `(get-picture-ids ice-cream-pic summer-shapes)` produces `(list 'top-scoop 'bottom-scoop 'cone 'ice-cream 'sun)`. The ShapeIDs must all be present, but they do not need to be in a particular order.

`(list 'ice-cream 'top-scoop 'sun 'cone 'bottom-scoop)` would also be valid.

(e) Write `picture->primitives`, which consumes a Picture and a ShapeList, and produces a list of PrimElements that can be rendered. Note that the list produced is flat, not nested (that is, the function produces `(listof PrimElement)`).

Note that the order of elements in the produced list is important for this function, because this determines the order that elements are drawn. For example,

`(picture->primitives '(((20 30) ice-cream)) summer-shapes)` produces

```
(list
 (make-prim-triangle 
  (make-posn 20 70)
  (make-posn 40 70)
  (make-posn 30 120)
  "Burlywood")
 (make-prim-circle (make-posn 30 65) 10 "LightBlue")
 (make-prim-circle (make-posn 30 55) 10 "Pink")
```
(f) Write `drawing->image`, which consumes a BundledDrawing and produces an Image. Because we cannot compare Images with `check-expect`, you do not need to provide tests or examples for this function. However, you should informally test the function by generating images. For example, `(drawing->image ice-cream-drawing)` produces

![Image of an ice cream](image.png)

(g) **Bonus (2%)**: If you wish, you may also submit a BundledDrawing for the ISAs to appreciate. They will award a small bonus to the nicest pictures. This may be the same constant you defined for `fun-drawing`, or it may be different. To participate in this part of the bonus, define a BundledDrawing with a 300x300 canvas called `shape-drawing` in the file `shape-bonus.rkt`, and use `save-image` to create a corresponding PNG image (`shape-bonus.png`) to submit.

Place your solutions in `drawings.rkt`.

---

**Enhancements: (drawings)** There are many ways to extend the drawing engine we wrote. You can make a copy of your code and then add features to your heart’s content. Here are some ideas.

- Our system supports translation of components. You can also add functionality that scales components, or rotates them.

- You can add a Shape type that allows conditional execution. This combined with image scaling would make it easy to produce fractals.

- Modify `render-image` in `a07drawinglib.rkt` to accept RGBA values for colours. This gives you transparency for free.

- A Picture contains pairs of Offsets and ShapeIDs. You could extend this so that in addition to taking (Offset ShapeID) pairs, a Picture can also be defined using (Offset Shape) pairs.

- You could support additional primitives, such as lines or text (all of which are supported in the underlying Racket image library).
None of these should be handed in, and they should not modify your 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 \(( + \ (\ast\ 3\ 3)\ (\ast\ 4\ 4))\) is represented as

\[
\begin{align*}
& (\text{make-add}) \\
& \quad (\text{make-mul} \ 3 \ 3) \\
& \quad (\text{make-mul} \ 4 \ 4)
\end{align*}
\]

But, as discussed in lecture, we can just represent it as the hierarchical list `\((+\ (\ast\ 3\ 3)\ (\ast\ 4\ 4))\)`. 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 \(( + \ (\ast\ x\ x)\ (\ast\ y\ y))\) and get the expression \(( + \ (\ast\ 3\ 3)\ (\ast\ y\ y))\). 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 `interleaf-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:

\[
\begin{align*}
\text{define} & \quad (\text{eternity} \ x) \\
& \quad (\text{eternity} \ x)
\end{align*}
\]

Think about what happens when we try to evaluate \((\text{eternity} \ 1)\) 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 \text{halts}.

The non-halting evaluation above can easily be detected, as there is no base case in the body of the function \text{eternity}. Sometimes non-halting evaluations are more subtle. We’d like to be able to write a function \text{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 \text{true} if and only if the evaluation of that function with that argument halts. Of course, we want an application of \text{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 \text{halting?}. Consider the following function of one argument:

\[
\begin{align*}
\text{define} & \quad (\text{diagonal} \ x) \\
& \quad (\text{cond} \\
& \quad \quad [\text{halting?} \ x \ x \ (\text{eternity} \ 1)] \\
& \quad \quad \text{[else} \quad \text{true}\text{])})
\end{align*}
\]

What happens when we evaluate an application of \text{diagonal} to a list representation of its own definition? Show that if this evaluation halts, then we can show that \text{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 \text{halting?}.

This is the celebrated \text{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 \text{function=} consumess two lists representing the code for the two functions. This is the situation Church considered in his proof.