Assignment: 4

Due: Thursday, June 11 at 9:00 am
Language level: Beginning Student
Coverage: Modules 1–4

For this and all subsequent assignments, to receive full marks you are required to use the design recipe for every function you write. You may include the examples given in the assignment in your submissions, but they will be ignored by the markers—you must develop a complete suite of examples and tests on your own. For your convenience, an interface file which contains the headers of the required functions is available on the course webpage.

Do not send any code files to course staff; they will not be accepted. Submissions must be made via MarkUs as described on the course webpage. After submission, check your basic test results to ensure your files were properly submitted. Solutions that do not pass the basic tests are unlikely to receive any correctness marks.

Remember, the solutions you submit must be **entirely your own work**.

1. In this question, you will perform step-by-step evaluations of Racket programs, by applying the semantic rules given in class until you either arrive at a final value or cannot continue. You will use an online evaluation tool that we have created for this purpose:

   `www.student.cs.uwaterloo.ca/~cs115/stepping`

   After logging in, try the "Warm-up questions" to get used to the system, and then move on to the "Assignment questions" once you are ready.

   You can re-enter a step as many times as necessary until you get it right, so keep going until you finish every question. There is no file to submit for this question—we will record that you have completed each question once you see the message "Question complete!", and your completion status will be contained in the basic tests for this assignment.

   You should not use DrRacket's stepper for this question, because it is important for you to get practice applying the semantic rules so that you can trace through the execution of programs (both your own and those on exams). The DrRacket stepper also uses slightly different rules.

2. One way of keeping track of a length of time which is useful in some contexts is to separately store the number of seconds, minutes, and hours which have passed. In this question, we'll do this by employing the following structure and data definition:

   (**define-struct** *duration* (*hours minutes seconds*))
   ;; A Duration is a (make-duration Nat Nat Nat)
   ;; requires: minutes, seconds < 60

   (a) Write a Racket function *duration-length* which consumes a *Duration* and produces its length in hours.

   For example, (*duration-length* (*make-duration* 1 30 0)) should produce 1.5.

   (b) Write a Racket function *duration<?* which consumes two *Durations* and produces *true* if the second is strictly longer than the first, and *false* otherwise.

   For example, (*duration<?* (*make-duration* 1 2 3) (*make-duration* 3 2 1)) should produce *true*.

   (c) Write a Racket function *add-duration* which consumes two *Durations* (*dur1* and *dur2*, in that order) and produces a *Duration* whose length is exactly the lengths of *dur1* and *dur2* added together.

   For example, (*add-duration* (*make-duration* 1 2 3) (*make-duration* 3 2 1)) should produce (*make-duration* 4 4 4).

3. One way of storing simple two dimensional shapes in a coordinate system is to store the coordinates of their vertices. For example, a triangle would require storing three points:

   (**define-struct** *triangle* (*vertex1 vertex2 vertex3*))
   ;; A Triangle is a (make-triangle Posn Posn Posn)
   ;; requires: vertex1, vertex2, and vertex3 are distinct vertices of a triangle

   A rectangle whose sides are parallel to the *x* and *y* axes can be stored with just two vertices, so long as they are from opposite corners:

   (**define-struct** *rectangle* (*vertex1 vertex2*))
   ;; A Rectangle is a (make-rectangle Posn Posn)
   ;; requires: vertex1 and vertex2 are distinct non-adjacent vertices of a rectangle
   ;; whose sides are parallel to the x and y axes

   A complication of using this representation is that there are multiple ways of representing the same shape; for example the *Rectangles* defined by

   (**define** *rectangle1* (*make-rectangle* (*make-posn* 0 0) (*make-posn* 1 1)))
   (**define** *rectangle2* (*make-rectangle* (*make-posn* 1 0) (*make-posn* 0 1)))

   represent the same shape. Write a Racket function *shape=?* which consumes two *Triangles* or *Rectangles* (or one of each) and produces *true* if the shape represented by both of the consumed values are the same, and *false* otherwise.

   For example, (*shape=? rectangle1 rectangle2*) should produce *true*.