Due: Wednesday, February 13 at 10:00 AM (No late submissions)

Assignment Guidelines:

- For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including helper functions.
- **For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe. Read the Style Guide carefully to ensure that you are following the proper conventions.** ***In addition, your solution must include the definition of constants and helper functions where appropriate.***
- **Built-in functions and special forms**: Unless otherwise indicated in the question you may use only the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment.
- **Helper functions**: Where applicable, please organize your code into helper functions to keep solutions simple and readable. **This is required for full marks**.
- Download the interface file from the course web page to ensure that all function names are spelled correctly, and each function has the correct number and order of parameters.
- Read each question carefully for restrictions.
- Test data for all questions will always meet the stated assumptions for consumed values.
- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include references to the parameter names of your functions.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
- Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
- You may post general assignment questions using the discussion groups on Waterloo LEARN. Choose Connect → Discussions. Read the guidelines for posting questions. Do NOT post any code as part of your questions.
- Check Markus and your basic test results to ensure that your files were properly submitted. In most cases, solutions that do not pass the basic tests will not receive any correctness marks.
- Read the course web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the Style Guide.
- Your solutions should be placed in files a4qY.rkt, where Y is a value from 1 to 2.

Plagiarism: The following applies to all assignments in CS115.

- Be sure to read the Plagiarism section at https://www.student.cs.uwaterloo.ca/~cs115/#assignments

Due: Wednesday, February 13 at 10:00 AM (No late submissions)

Language level: Beginning Student
Coverage: Module 4

Assignment 4 focuses on structures and functions (and helper functions!) that could be used to build a video game. While the questions are inspired by Nintendo's game *The Legend of Zelda: Breath of the Wild (BotW)*, they can be solved using the techniques in Modules 1 through 4—no gaming experience or knowledge is required.

1. Players are encouraged to explore the vast world of *BotW* to find hidden shrines. Visiting these shrines in random order often takes months of gameplay, even though following the shortest path connecting all of the shrines may take only a few days.

   Finding the shortest path connecting any set of positions is a famous problem in computer science known as the Travelling Salesman Problem (TSP). This assignment question only requires that we find the shortest path connecting four shrine positions—enough to caution the student from any serious engagement with the game while on campus!

   a. We define an `Xy-Path` to be the line segments connecting a sequence of four (x,y) coordinates visited in increasing order, i.e., start at (x1, y1), then travel straight to (x2, y2), and so on.

   ```
   (define-struct xy-path (x1 y1 x2 y2 x3 y3 x4 y4))
   ;; An Xy-Path is a (make-xy-path Num Num Num Num
   ;;                               Num Num Num Num)
   ```

   Write a template for the `Xy-Path` type.

   b. Write a function `xy-path-length` that consumes an `Xy-Path` and produces the distance (a `Num`) travelled between the four (x,y) coordinates along that path. Recall that we provide code for the Euclidean distance between two `Posn`'s on Slide 12 of Module 4.

   Example:
   ```
   (xy-path-length (make-xy-path 0 0 1 0 2 0 3 0)) => 3
   ```

   Testing hint: use `check-expect` to test exact path lengths, but use `check-within` to test inexact path lengths. A tolerance of 0.0001 can be used.

   c. We define a `Posn-Path` the same way as an `Xy-Path`, four coordinates visited in increasing order, except that the structure uses nested `Posn` structures for each of the four coordinates.

   ```
   (define-struct posn-path (p1 p2 p3 p4))
   ;; A Posn-Path is a (make-posn-path Posn Posn Posn Posn)
   ```

   Write a template for the `Posn-Path` type.

d. Write a function `posn-path-length` that consumes a `Posn-Path` and produces the distance travelled between the four (x,y) path coordinates (similar to `xy-path-length`).

Example:

```
(define posn-path0
    (make-posn-path (make-posn 0 0) (make-posn 1 0)
                    (make-posn 3 1.5) (make-posn 4 1.5)))
(posn-path-length posn-path0) => 4.5
```

because the first and last segments have length 1, and the middle segment has length 2.5.

e. Use `posn-path-length` as a helper function to write a function `shortest-path2` that consumes two `Posn-Path`'s and produces the shorter of those two paths. Then use `shortest-path2` as a helper function to write a function `shortest-path4` that consumes four `Posn-Path`'s and produces the shortest of those four paths. If more than one of the consumed paths has the shortest length, then your function may produce any of these paths.

Example:

```
(define posn-path1
    (make-posn-path (make-posn 0 0) (make-posn 1 0)
                    (make-posn 2 0) (make-posn 4 0)))
(define posn-path2
    (make-posn-path (make-posn 0 0) (make-posn 1 0)
                    (make-posn 2 0) (make-posn 5 0)))
(define posn-path3
    (make-posn-path (make-posn 0 0) (make-posn 1 0)
                    (make-posn 2 0) (make-posn 6 0)))
(define posn-path4
    (make-posn-path (make-posn 4 0) (make-posn 2 0)
                    (make-posn 1 0) (make-posn 0 0)))
(shortest-path2 posn-path1 posn-path2) => posn-path1
(shortest-path4 posn-path0 posn-path1 posn-path2 posn-path3)
    => posn-path1
```

Indirect Testing:

Since the shortest path may not be unique, we can't test whether `shortest-path2` and `shortest-path4` produce a specific path. Instead, we need to test whether the produced path has the same length as a known shortest-length path. E.g., both paths are the shortest path:

```
(check-expect
    (posn-path-length (shortest-path2 posn-path1 posn-path4))
    (posn-path-length posn-path1))
(check-expect
    (posn-path-length (shortest-path2 posn-path1 posn-path4))
    (posn-path-length posn-path4))
```

f. Write a function `shortest-path` that consumes four `Posn`'s and produces the `Posn-Path` with shortest length that connects those `Posn`'s. Consider writing a helper function like `shortest-path24` that checks all 24 different `Posn-Path`'s that connect the four `Posn`'s.

Hint: The function `shortest-path4` used `shortest-path2` as a helper function. We could continue this approach and use `shortest-path4` to write a function `shortest-path8`, which could help write `shortest-path16`, and so on.

Example/Testing:
As before, the shortest path is not unique (the reverse path has the same length). Your function may produce any of the shortest paths, so indirect testing should be used, as in part e.

```
(define posn0 (make-posn 0 0))
(define posn1 (make-posn 1 0))
(define posn2 (make-posn 2 0))
(define posn3 (make-posn 3 0))
(check-expect
  (posn-path-length (shortest-path posn3 posn0 posn2 posn1))
  (posn-path-length (make-posn-path posn0 posn1 posn2 posn3)))
```

2. The `Combatant` structure represents the sword-wielding characters from *BotW*. The `Weapon` structure represents the `Combatant`'s sword.

```
(define-struct weapon (damage-per-strike health))
;; A Weapon is a (make-weapon Nat Nat)
;; Requires: damage-per-strike > 0

(define-struct combatant (sword health))
;; A Combatant is a (make-combatant Weapon Nat)
```

Both the `Weapon` and `Combatant` structures have a `health` field that represent their ability to continue in battle:

- Every time the opponent's sword damages the combatant, the `Combatant`'s `health` field decreases by the value of the `Weapon` structure's `damage-per-strike` field, unless `health` reaches the minimum value of zero, when the combatant becomes non-viable and incapable of continuing in battle.

- The `Weapon` structure's `health` field decreases by one every time the sword strikes the opponent, until reaching the minimum value of zero (broken sword, incapable of battle).

a. If the combatant knows the total amount of damage that a sword can inflict before breaking, then they can avoid battling opponents that are too powerful (i.e., the sword breaks before the opponent becomes non-viable). Knowing the total damage also allows the combatant to choose which sword to wield.

Write a function `total-damage` that consumes a `Weapon` and produces the total amount of damage that it inflicts before breaking, i.e., the product of the sword's `damage-per-strike` field and `health` field.

Write another function `engage-opponent?` that consumes a sword (`Weapon`) and an opponent (`Combatant`) and produces `true` if the sword could make the opponent non-viable before breaking. The function produces `false` otherwise.

Examples:

```
(define sword (make-weapon 5 4))
(define opponent (make-combatant (make-weapon 1 1) 10)
(total-damage sword) => 20
(engage-opponent? sword opponent) => true
```

b. We wish to write a function `strike-opponent` to simulate the simplest of battles between two combatants: an attacking combatant strikes an opponent combatant exactly once. Then this function must consume two `Combatant`'s, the `attacker` and the `opponent`. The preamble specifies that the strike damages the `health` field of the opponent and the `health` field of the attacker's sword, so the function must produce both the updated opponent and the updated attacker. The `strike-opponent` function produces the two updated objects by producing a `Battle-Outcome` which contains both the updated opponent and the updated attacker:

```
(define-struct battle-outcome (attacker opponent))
;; A Battle-Outcome is a (make-battle-outcome Combatant
;;                                            Combatant)
```

Write a function `strike-opponent` that simulates the battle and updates the two `Combatant`'s `health` fields as described in the preamble.

Note the highlighted `health` field values in the example. Also note that no strike occurs and the combatants remain unchanged if the consumed attacker is non-viable or has a broken sword, or if the consumed opponent is non-viable.

Examples:

```
(define attacker-before (make-combatant (make-weapon 2 5) 1))
(define attacker-after (make-combatant (make-weapon 2 4) 1))
(define nonviable-attacker (make-combatant (make-weapon 2 5) 0))
(define opponent-before (make-combatant (make-weapon 1 1) 4))
(define opponent-after (make-combatant (make-weapon 1 1) 2))

(strike-opponent attacker-before opponent-before) =>
    (make-battle-outcome attacker-after opponent-after)
(strike-opponent nonviable-attacker opponent-before) =>
    (make-battle-outcome nonviable-attacker opponent-before)
```