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.**

• 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. A list of functions described in each module of the lecture slides can be found at [https://www.student.cs.uwaterloo.ca/~cs115/built_in](https://www.student.cs.uwaterloo.ca/~cs115/built_in)

• 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 3 (use `a4q4.rkt` if submitting a solution for the bonus question).

**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#Plagiarism](https://www.student.cs.uwaterloo.ca/~cs115/assignments#Plagiarism)
Assignment 4 focuses on using structures and functions that could be used to design a video game. While the assignment questions are based on themes from Nintendo’s hit game The Legend of Zelda: Breath of the Wild (BotW), they are readily solvable by applying the ideas and techniques found in Modules 1 through 4. No gaming experience or knowledge required.

1. Players are encouraged to explore the vast world of BotW to find hidden shrines. Visiting these shrines in random order takes months of actual gameplay, while the shortest path connecting all of the shrines takes 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 length of the shortest path connecting three shrine positions—enough to caution the student from any serious engagement with the game while on campus!

a. Write a function path-length that consumes three Posn’s (that constitute an ordered path) and produces the distance travelled between the positions along that path. Recall that we provide code for the Euclidean distance between two Posn’s on Slide 12 of Module 4.

Example:

(check-expect (path-length (make-posn 0 0) (make-posn 1 0) (make-posn 2 0)) 2)

Testing hint: consider two kinds of tests, axis-aligned and diagonal. The path of the axis-aligned Posn’s should follow the x or y axes, which produces a Nat path length that can be tested using check-expect. Diagonal paths should be tested using check-within.

b. Using the functions path-length and min, develop a function shortest-path-length that consumes the Posn locations of three shrines and produces the length of the shortest path that visits all three shrines.

Example:

(define shrine-a (make-posn 2 0))
(define shrine-b (make-posn 0 0))
(define shrine-c (make-posn 1 0))
(check-expect (shortest-path-length shrine-a shrine-b shrine-c) 2)
2. The **Weapon** and **Hero** structures represent the playable character from *BotW*, Link, who wields a sword to battle enemies.

```scheme
(define-struct health (maximum current))
;; A Health is a (make-health Nat Nat)
;; Requires:
;;   current <= maximum

(define-struct weapon (strike-damage durability))
;; A Weapon is a (make-weapon Nat Health)
;; Requires:
;;   strike-damage > 0

;; The Hero’s rupee-total field is the total balance of rupees
;; (i.e., the in-game currency) possessed by the Hero
(define-struct hero (sword life rupee-total))
;; A Hero is a (make-hero Weapon Health Nat)

Both of the **Weapon** and **Hero** structures have a nested **Health** structure:

- **The Hero structure’s life field** represents the hero’s ability to continue in battle. Every time the opponent’s sword damages the hero, the life field’s current field decreases by the value of the Weapon structure’s strike-damage field. When current reaches 0, the hero becomes non-viable, incapable of continuing in battle. See sample code below.

  ```scheme
  ;; use appropriate field values to the complete the constructor
  (define link (make-hero …))

  ;; produce Link’s current health
  ;; i.e., the life field’s current field
  (health-current (hero-life link))
  ```

- **The Hero structure’s sword breaks after a fixed number of strikes determined by the durability field’s maximum field.** The current field decreases by 1 every time the sword strikes the opponent. The sample code below illustrates how to produce this field.

  ```scheme
  ;; produce Link’s sword’s maximum health
  ;; i.e., the durability field’s maximum field
  (health-maximum (weapon-durability (hero-sword link)))
  ```

a. If Link knows the total amount of damage that a sword can inflict before breaking, then the hero 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 Link to compare two swords and choose which one to wield.
Design 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 strike-damage field and the sword durability field's current field.

Example:

(define damaged-broadsword (make-weapon 5 (make-health 10 5)))
(check-expect (total-damage damaged-broadsword) 25)

b. Link may replace the wielded sword with another sword, but the other sword must be repaired before use.

i. Recall that in Racket, we don't repair or modify field values—we construct a copy of the data except with the desired field value. Then the sword repair process creates a copy of the sword except with the desired durability field that reflects a repaired sword—the current field is equal to the value of the maximum field.

Develop a function repair-sword that consumes a Weapon and produces another Weapon with the same fields as the consumed Weapon, except the durability field reflects a repaired sword.

Example:

(define fresh-broadsword (make-weapon 5 (make-health 10 10)))
(check-expect (repair-sword damaged-broadsword) fresh-broadsword)

ii. Develop a function repair-and-wield-sword that consumes a Hero and the encountered Weapon and produces another Hero with the same fields as the consumed Hero, except wielding the repaired Weapon.

Example:

(define damaged-claymore (make-weapon 52 (make-health 100 50)))
(define fresh-claymore (make-weapon 52 (make-health 100 50)))
(define life (make-health 100 100))
(define rupees 0)
(define link (make-hero fresh-broadsword life rupees))

(check-expect (repair-and-wield-sword link damaged-claymore) (make-hero fresh-claymore life rupees))

3. While adventuring, the hero Link finds treasure chests that contain either a sword, a rupee, or nothing (i.e., the chest's treasure was already removed). The Treasure and Chest types represent these chests and their treasures.
A Treasure is one of:
* a Nat or
* a Weapon

(define-struct chest (removed? contents))

A Chest is a (make-chest Bool Treasure)

The chest's removed? field is true if the contents have been removed from the chest, false otherwise. If the Treasure is a Nat, it represents the value of a rupee, the in-game currency.

We wish to write a function that opens a chest and adds the contents to Link's inventory.

When the chest contents have already been removed (i.e., the chest is empty), Link adds nothing to the inventory. When the chest contains a rupee, Link always takes it from the chest and adds it to the total rupees. Otherwise when the chest contains a sword, Link only acquires that sword when its total damage is greater than the sword already wielded. Assume that a sword in a treasure chest needs no repair because it is already new.

Develop a function open-chest that consumes a Hero and a Chest and produces another Hero with the same fields as the consumed Hero, except augmented by the Treasure as specified.

Examples:

(define broadsword-link (make-hero fresh-broadsword life rupees))
(define claymore-link (make-hero fresh-claymore life rupees))
(define gold-rupee 300)
(define rupee-chest (make-chest false gold-rupee))
(define claymore-chest (make-chest false fresh-claymore))
(define broadsword-chest (make-chest false fresh-broadsword))
(define empty-chest (make-chest true gold-rupee))

(check-expect (open-chest broadsword-link rupee-chest)
  (make-hero fresh-broadsword life gold-rupee))
(check-expect (open-chest broadsword-link empty-chest)
  broadsword-link)
(check-expect (open-chest broadsword-link claymore-chest)
  claymore-link)
(check-expect (open-chest claymore-link broadsword-chest)
  claymore-link)
4. **BONUS PROBLEM.** No hints will be given for this problem. Only clarification should be expected.

When the hero Link encounters an enemy moblin, they engage in battle, taking turns swinging swords, damaging their health and swords accordingly. The structure definitions and preamble in Question 2 specify how to calculate the damage to the nested `Health` structure fields.

The battle ends immediately after Link or the moblin’s health become nonviable, or, when both of their swords are broken. Note that the battle continues if only one of the swords are broken. For example, if Link’s sword breaks before making the moblin non-viable, then the moblin will continue to battle Link until Link becomes non-viable, or the moblin’s sword breaks, whichever comes first.

```
(define-struct enemy (sword life))
;; An Enemy is a (make-enemy Weapon Health)
```

Develop a function `battle` that consumes a `Hero` and an `Enemy` and produces another `Hero` with the same fields as the consumed `Hero`, except with a health and sword damaged in the battle. Assume that Link always strikes first.

**Examples:**

```
(define moblin-life (make-health 100 100))
(define broadsword-moblin
  (make-enemy fresh-broadsword moblin-life))
(define claymore-moblin (make-enemy fresh-claymore moblin-life))

;; Link defeats the Moblin before either sword breaks
(check-expect (battle claymore-link broadsword-moblin)
  (make-hero
   (make-weapon 52 (make-health 100 98))
   (make-health 100 95)
   rupees))

;; Link’s health becomes nonviable before his sword breaks
(check-expect (battle broadsword-link claymore-moblin)
  (make-hero
   (make-weapon 5 (make-health 10 8))
   (make-health 100 0)
   rupees))

;; The battle ends because both Link and Moblin broadswords broke
;; before Link and Moblin health could become nonviable
(check-expect (battle broadsword-link broadsword-moblin)
  (make-hero
   (make-weapon 5 (make-health 10 0))
   (make-health 100 50)
   rupees))
```