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

Assignment: 6

Due:	Tuesday, October 29 2019, 9:00pm
Language level:	Beginning Student with List Abbreviations
Files to submit:	`not-catan.rkt`, `hr.rkt`, `mobster.rkt`
Warmup exercises:	HtDP 13.0.3, 13.0.4, 13.0.7, 13.0.8, 14.2.1, 14.2.2, 17.2.1, 17.3.1, 17.6.4, and 17.8.3
Practice exercises:	HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6, 14.2.3, 14.2.4, 14.2.6, 17.1.2, 17.2.2, 17.6.2, 17.8.4, and 17.8.8

- **Make sure you read the OFFICIAL A06 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 05 carry forward.
- This assignment covers material up to the end of Module 10, slide 21.
- Of the built-in list functions, you may use **only** `cons`, `first`, `second`, `third`, `rest`, `empty?`, `cons?`, `list`, `member?`, and `length`.
- You may **only** use the functions that have been discussed in the lecture slides, unless explicitly allowed or disallowed in the question. In particular, you may **not** use any of the following Racket functions: `reverse`, `make-string`, `replicate`, or `list-ref`. You may define your own versions of these functions as long as they are written using simple or accumulative recursion.
- Solutions will be marked for both correctness **[80%]** and style **[20%]**. Follow the guidelines in the Style Guide, pages 1-17 (skipping sections 3.7.2 and 3.7.4 for now).

Here are the assignment questions you need to submit.

1. **[20% Correctness]** In the board game Builders of Qumran™, players require five types of resources to progress in the game: Blocks, Wood, Sheep, Wheat, and Rocks. These resources can be stored in an `Inventory` structure. Players can then use their resources to build infrastructure, such as, Streets, Villages, and Towns. The cost for each of these infrastructure items is stored in a `Cost` structure.

```
(define-struct inventory (blocks wood sheep wheat rocks))
;; An Inventory is a (make-inventory Nat Nat Nat Nat Nat)
;; requires: wheat >= sheep
;;           wheat >= 4

(define-struct cost (blocks wood sheep wheat rocks))
;; A Cost is a (make-cost Nat Nat Nat Nat Nat)
```

   (a) Write a template function `inventory-template` that consumes an `Inventory` and produces `Any`. Note that for this template function, you only need to provide a contract and a body.

   (b) Starting with your template, implement the predicate `valid-inventory?`. This predicate consumes `Any` as an argument and produces `true` if the argument is a valid `Inventory`.

```
(check-expect (valid-inventory? (make-inventory 3 2 4 5 1)) true)
(check-expect (valid-inventory? (make-inventory 3 2 2 3 1)) false)
```

   (c) Starting with your template, implement the predicate `affordable?`. This predicate consumes an `Inventory` and a `Cost`. The predicate produces `true` if the cost is covered by the inventory and if the inventory remains a valid `Inventory` after subtracting the cost, and `false` otherwise.

```
(check-expect
  (affordable? (make-inventory 3 2 4 5 1) (make-cost 0 2 0 1 0)) true)
(check-expect
  (affordable? (make-inventory 3 2 5 5 1) (make-cost 0 2 0 1 0)) false)
```

   (d) Starting with your template, implement the function `thief`. This function consumes an `Inventory` as argument and produces another `Inventory`. The thief attempts to steal one unit of each resource. If this is not possible, the thief leaves this one resource unchanged. Calling `(valid-inventory? (thief ...))` on any `Inventory` will always produce `true`.

```
(check-expect
  (thief (make-inventory 3 2 4 5 0)) (make-inventory 2 1 3 4 0))
```

   Place your solutions to this problem in `not-catan.rkt` .

2. **[25% Correctness]** Human Resources is asking you to help them solve some problems related to their personnel records. All personnel records are stored in an employee database.

```
(define-struct pr (first-name last-name))
;; A personnel record (PR) is a (make-pr Str Str).
```

For the purpose of this assignment, the names in a `PR` are unique identifiers, i.e., if two `PR`s are equal, they both refer to the same employee.

```
;; An employee database (ED) is a (listof PR).

(define ed-kitchener (list (make-pr "Gord" "Downie")
                           (make-pr "Rob" "Baker")
                           (make-pr "Gord" "Sinclair")
                           (make-pr "Paul" "Langlois")
                           (make-pr "Johnny" "Fay")))

(define ed-waterloo (list (make-pr "Byron" "Stroud")
                          (make-pr "Jed" "Simon")
                          (make-pr "Gord" "Downie")
                          (make-pr "Devin" "Townsend")
                          (make-pr "Rob" "Baker")))
```

For the purpose of this assignment, an `ED` is a set, i.e., there cannot be two equal `PR`s in one `ED`.

Warning: you are **not allowed** to use the predicate `equal?` anywhere in Question 2!

(a) Racket provides us with numerous functions that allow for comparing values of the same type (e.g., `>=` for `Num`, `symbol=?` for `Sym`, and `string<?` for `Str`). Now that we have defined our own structures, it is useful to write our own (in-) equality predicates:

i. `pr=?` consumes two `PR`s and produces `true` if they are identical (i.e., both first name and last name are equal).
Two examples:

```
(check-expect (pr=? (make-pr "Gord" "Down") (make-pr "Gord" "Down")) true)
(check-expect (pr=? (make-pr "Gord" "Down") (make-pr "Dord" "Gown")) false)
```

ii. `pr<?` consumes two `PR`s and a `Sym`. The `Sym` can either be `'first` or `'last` and determines if the initial comparison should be based on the first or the last name. The predicate produces `true` if the first `PR` precedes the second one lexicographically. If the initial comparison compares two equal names, the predicate will consider the other name. If both `PR`s have identical first and last names, the predicate will produce `false`.
A few examples:

```
(check-expect
  (pr<? (make-pr "Gord" "Down") (make-pr "Gard" "Duwn") 'first) false)
(check-expect
  (pr<? (make-pr "Gord" "Down") (make-pr "Gard" "Duwn") 'last) true)
```

```
(check-expect
  (pr<? (make-pr "Gord" "Down") (make-pr "Gord" "Dawn") 'last) false)
(check-expect
  (pr<? (make-pr "Gord" "Dawn") (make-pr "Gyrd" "Dawn") 'first) true)
(check-expect
  (pr<? (make-pr "Gord" "Down") (make-pr "Gord" "Dawn") 'first) false)
(check-expect
  (pr<? (make-pr "Gord" "Down") (make-pr "Gord" "Down") 'first) false)
```

iii. `pr>?` consumes two `PR`s and a `Sym`. The `Sym` can either be `'first` or `'last` and determines if the initial comparison should be based on the first or the last name. The predicate produces `true` if the first `PR` is lexicographically larger than the second one. If the initial comparison compares two equal names, the predicate will consider the other name. If both `PR`s have identical first and last names, the predicate will produce `false`. Examples work similar to `pr<?`.

Hint: while you can certainly test for (in-) equality using `PR` directly, you might also consider converting `PR` to another type of data and then using some of Racket's built-in functions on the result: doing so might simplify your solution significantly.

(b) Write a function `sort-ed` that sorts an employee database lexicographically. `sort-ed` consumes an `ED` and a `Sym`. The `Sym` can either be `'last`, which would sort the employee database by last name, or `'first`, which would sort it by first name. If the initial comparison compares two equal names, the predicate will consider the other name. The function produces an `ED`.

For example:

```
(check-expect (sort-ed ed-kitchener 'last)
  (list (make-pr "Rob" "Baker")
        (make-pr "Gord" "Downie")
        (make-pr "Johnny" "Fay")
        (make-pr "Paul" "Langlois")
        (make-pr "Gord" "Sinclair")))
```

Another example:

```
(check-expect (sort-ed ed-waterloo 'first)
  (list (make-pr "Byron" "Stroud")
        (make-pr "Devin" "Townsend")
        (make-pr "Gord" "Downie")
        (make-pr "Jed" "Simon")
        (make-pr "Rob" "Baker")))
```

(c) To streamline operations, Human Resources wants to merge employee databases. Write a function `merge-ed` that merges two `ED`s into a single one. While both `ED`s can be of any order, the resulting `ED` must be sorted by last name. Note that employees can work in multiple locations (e.g., "Gord Downie"). Make sure that the merged `ED` does not contain any duplicates!

For example,

```
(check-expect (merge-ed ed-kitchener ed-waterloo)
  (list (make-pr "Rob" "Baker")
        (make-pr "Gord" "Downie")
        (make-pr "Johnny" "Fay")
        (make-pr "Paul" "Langlois")
        (make-pr "Jed" "Simon")
        (make-pr "Gord" "Sinclair")
        (make-pr "Byron" "Stroud")
        (make-pr "Devin" "Townsend")))
```

Place your solutions to this problem in `hr.rkt` .

3. **[35% Correctness]** Congratulations! Your education at UW has paid off and you became a successful ... mobster? Oh, well, at least you can use your programming skills to run your criminal empire as efficiently as possible.

```
(define-struct goon (street-name abilities))
;; A Goon is a (make-goon Str Abilities).
;; An Abilities is a (list Nat Nat Nat), where the elements represent
;;   Loyalty, Wealth, and Influence.

(define goon-btt (make-goon "Bullet Tooth Tony" (list 8 2 5)))
(define goon-ca (make-goon "Cousin Avi" (list 3 9 8)))
(define goon-btb (make-goon "Boris, the Blade" (list 4 6 7)))
(define goon-fff (make-goon "Franky Four Fingers" (list 5 7 3)))

(define applicant-gg (make-goon "Gorgeous George" (list 3 5 4)))
(define applicant-s (make-goon "Sol" (list 5 7 10)))

;; A Gang is a (listof Goon)

(define my-gang (list goon-btt goon-ca goon-btb goon-fff))

;; A Job is a (list Nat Nat Nat), where the elements represent
;;   required Loyalty, required Wealth, and required Influence.

(define job-mule (list 5 0 1))
(define job-financer (list 3 8 5))
(define job-bribe (list 5 6 10))

;; A Job-list is a (listof Job)

(define my-jobs (list job-mule job-financer job-bribe))
```

   (a) Your first order of business is evaluating your gang-members according to their abilities. Write a function `eval-goon` that consumes a `Goon` and a `Job`. The function produces `false` if one or more of the goon's abilities do not meet the requirements for the job. If all requirements are met, the function produces a `Nat` that expresses how qualified

the goon is. This number is calculated by adding up all differences between the goon's abilities and the job's requirements.

For example, "Cousin Avi" would not be a suitable mule, because she does not meet the Loyalty-requirement:

```
(check-expect (eval-goon goon-ca job-mule) false).
```

She would, however, be quite suitable as a financer:

```
(check-expect (eval-goon goon-ca job-financer) 4).
```

Last, "Franky Four Fingers" would be an excellent mule:

```
(check-expect (eval-goon goon-fff job-mule) 9).
```

Hint: to make your function more flexible, implement it in a way that it can process abilities with an arbitrary number of elements! While doing so is not required, it is a good opportunity to practise recursion within recursion. Be careful when choosing the type of recursion for your solution: the best type might not be the obvious one.

(b) Now that you can assess all of your gang-member individually, it is high time to find the right goon for the job!

Write a function `pick-goon` that consumes a `Gang` and a `Job`. The function produces the `Goon` that is most qualified for the job. In case multiple goons are equally qualified, the function produces the first qualified one from your gang. If no goon is qualified, the function produces `false`.

For example, both "Bullet Tooth Tony" and "Franky Four Fingers" could become mules. Both of them are equally qualified (score of 9). You pick, however, "Bullet Tooth Tony" because he is the first in your gang:

```
(check-expect (pick-goon my-gang job-mule) goon-btt).
```

However, none one of your gang-members are qualified enough to bribe others:

```
(check-expect (pick-goon my-gang job-bribe) false).
```

Also, remember that you might just be starting with your gang:

```
(check-expect (pick-goon empty job-financer) false).
```

(c) Going through your list of jobs, you realize that some of them are too demanding for your current gang-members. Maybe you need to recruit more goons into your gang. In order to post your open positions, however, you have to create a list of all jobs that your current gang cannot complete.

Write a function `find-difficult-jobs` that consumes a `Gang` and a `Job-list`. The function produces a `Job-list` that contains all jobs that are too difficult for your current gang-members to complete.

For example,

```
(check-expect (find-difficult-jobs my-gang my-jobs) (list job-bribe)).
```

(d) Hiring day has come! You have been advertising for months, and now applicants are lining up at your secret head-quarters. To deal with the flood of applications, you have to write a program that helps you in your decision-making process.

Write a predicate `hire?` that consumes a `Gang`, a `Job-list`, and a `Goon` (this parameter represents the applicant). The predicate produces `true` if the applicant is qualified to perform a job that none of the current gang-members can, and `false` if every job is already covered by at least one current gang-member.

For example, "Gorgeous George" is not qualified enough to fill your ranks:

```
(check-expect (hire? my-gang my-jobs applicant-gg) false).
```

"Sol" on the other hand will be a great addition to your growing gang, as he will excel in bribing others:

```
(check-expect (hire? my-gang my-jobs applicant-s) true).
```

Place your solutions to this problem in `mobster.rkt` .

This concludes the list of questions for which you need to submit solutions. As always, check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

The IEEE-754 standard, most recently updated in 2008, outlines how computers store floating-point numbers. A floating-point number is a number of the form

$$0.d_1 d_2 \ldots d_t \times \beta^e$$

where β is the base, e is the integer *exponent*, and the *mantissa* is specified by the t digits $d_i \in \{0, \ldots, \beta - 1\}$. For example, the number 3.14 can be written

$$0.314 \times 10^1$$

. In this example, the base is 10, the exponent is 1, and the digits are 3, 1 and 4. A computer uses the binary number system; the binary equivalent to the above example is approximately

$$0.11001001_2 \times 2^{10_2}$$

where 10_2 is the binary integer representing the decimal number 2. Thus, computers represent such numbers by storing the binary digits of the *mantissa* ("11001001" in the example), and the *exponent* ("10" in the example). Putting those together, a ten-bit floating-point representation for 3.14 would be "1100100110".

However, computers use more binary digits for each number, typically 32 bits, or 64 bits. The IEEE-754 standard outlines how these bits are used to specify the mantissa and exponent. The specification includes special bit-patterns that represent Inf, $-$Inf, and NaN.