Assignment Guidelines:

- Solutions to these questions are expected to follow the requirements of the Style Guide (https://www.student.cs.uwaterloo.ca/~cs115/coursenotes1/styleguide.pdf). This includes all relevant design recipe elements, proper use of constants, and proper use of helper functions.
- Submission details:
  - Solutions to these questions must be placed in files `a06q1.rkt`, `a06q2.rkt`, `a06q3.rkt`, respectively, and must be completed in Racket.
  - All solutions must be submitted through MarkUs. Solutions will not be accepted through email.
  - Verify your basic test results using MarkUs to ensure that your files were submitted properly and are readable on MarkUs. *Note, however, that passing the basic tests does not guarantee that you will pass all our correctness tests.*
- 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.
- Restrictions:
  - You may only use the built-in functions and special forms introduced in the lecture slides up to and including the module covered by this assignment. A list of these functions can be found on the Assignments web page: https://www.student.cs.uwaterloo.ca/~cs115/#allowed
  - Read each question carefully to see if any additional restrictions apply.
  - Test data for correctness tests will always meet the stated assumptions for consumed values.
- 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.

Plagiarism: The following applies to all assignments in CS115.

All work in CS 115 is to be done individually. The penalty for plagiarism on assignments (first offense) is a mark of 0 on the affected question and 5 marks off the final grade, consistent with School of Computer Science policy. In addition, a letter detailing the offense is sent to the Associate Dean of Undergraduate Studies, meaning that subsequent offenses will carry more severe penalties, up to suspension or expulsion.

To avoid inadvertently incurring this penalty, you should discuss assignment issues with other students only in a very broad and high-level fashion. Do not take notes during such discussions, and avoid looking at anyone else’s code, on screen or on paper. If you find yourself stuck, contact the ISA or instructor for help, instead of getting the solution from someone else. Do not consult other books, library materials, Internet sources, or solutions (yours or other people’s) from other courses or other terms.

Read more course policies at: https://www.student.cs.uwaterloo.ca/~cs115/#policies

Language level: Beginning Student with List Abbreviations
Coverage: Module 5
**Question 1: Approximating π**

The Leibniz formula (sometimes called the Madhava-Leibniz formula) gives the following infinite series for calculating π:

\[
π = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots \right)
\]

You can calculate an approximation of π by adding up some number of terms of this series. It’s not a very good approximation (it converges to π very slowly), but it’s simple to implement.

Complete the Racket function `approximate-pi` that consumes a single natural number (n), adds up the first \(n + 1\) terms of the series above, and multiplies the sum by 4 to obtain an approximation of π. Use structural recursion on \(n\) to complete the function.

Do not use Racket’s built-in `range` function or the `countup-by` function in your course notes.

For example:

* `(approximate-pi 0) => 4`
* `(approximate-pi 1) => 8/3`
* `(approximate-pi 3) => 304/105`
* `(approximate-pi 5) => 10312/3465`
* `(approximate-pi 10) => 47028692/14549535`

It’s difficult to work out lots of test cases for this function by hand. **For this question, you are permitted to use exactly the examples shown here for your examples and tests. You don’t need to add any of your own.** If you’re feeling bold, you can include an additional test case that uses `check-within` to show that the function really does approximate π to within, say, 0.01. Since the formula converges to π very slowly, you should be careful to only test on values of \(n\) that are sufficiently large enough to converge to π.

Submit your solution in the file `a06q1.rkt`.

✥ ✥ ✥
Question 2: IATA Airport-Country Codes

An IATA airport code is a three-letter geocode designating many airports and metropolitan areas around the world.

;; An Airport is a (list Str (list Str Str))
;; where:
;; * the first item is the IATA code which is also the key.
;;   It is always a three-letter upper case string.
;; * the second item is the associated value of the IATA code.
;;   It is a list of length two, where the first element is the
;;   metropolitan area served by the airport and the second
;;   element is the country of the airport.

Use structural recursion to complete the Racket function same-country that consumes a list of Airport (airports) and a non-empty string (iata-code) that represents an IATA code. The function produces a list of all IATA codes in the same country as iata-code, sorted in ascending alphabetical order. You may assume that the given IATA code is always in the list of Airport.

For example, if you created the following constant:

```
(define alist (list (list "YYZ" (list "Toronto" "Canada"))
                    (list "YWG" (list "Winnipeg" "Canada"))
                    (list "BGO" (list "Bergen" "Norway"))
                    (list "TRF" (list "Sandefjord" "Norway"))
                    (list "YUL" (list "Montreal" "Canada"))
                    (list "LHR" (list "London" "England"))
                    (list "YVR" (list "Vancouver" "Canada")))))
```

Then

```
*(same-country alist "YUL") => (list "YUL" "YVR" "YWG" "YYZ")
```

You may use modified versions of both the lookup-al as well as my-sort functions in your course notes. Do not use Racket’s built-in sort function.

Submit your solution in the file a06q2.rkt.
Question 3: Dictionary Operations (Add and Remove)

For this question, you will be working with a dictionary that keeps track of data values based on their type. You will be using the following data definitions:

--; A DataTypeDictionary is an association list, where
--;   * the keys are one of "Num", "Str", "Char", or "Bool", and
--;   * the associated values are lists, where the types of
--;     the elements match the type indicated by the key.
--; There are no duplicate values in these lists.

--; An Action is a (list (anyof "add" "remove")
--;                   (anyof Num Str Char Bool))
--; where
--;   * the first element of the list represents an instruction
--;   * the second element of the list is a value either
--;     being added or removed from a dictionary.

Use structural recursion to complete the Racket function `update` that consumes a `DataTypeDictionary (data)` and an `Action (act)`, and produces an updated `DataTypeDictionary`. The updates should adhere to the following specifications:

- If the value of `act` is "remove" and the key is in the dictionary, then the matching value in the associated list should be removed from the associated list in the dictionary. If the matching value happens to be the last element in the associated list, then the key should remain in the dictionary, but the associated list becomes empty. If the key does not exist in the dictionary, or there is no matching value in the associated list, the updated dictionary would not change.

- If the value of `act` is "add" and the key is in the dictionary, then the new value should be added to the associated list in the dictionary if it does not already exist. If the value already exists in the associated list, the updated dictionary would not change.

- If the value of `act` is "add" and the key does not exist in the dictionary, then a new association list entry for the key should be added to the dictionary.

- The keys in the dictionary may appear in any order and the elements in the associated lists may also appear in any order.
For example, if you created the following constant:

```racket
(define start (list (list "Num" (list 5 1.3 -1))
  (list "Bool" empty)
  (list "Char" (list #\p))
  (list "Str"
    (list "cs" "CS" "Computer Science")))
```

Then

* `(update start (list "add" true)) =>
  
  (list (list "Num" (list 5 1.3 -1))
    (list "Bool" (list true))
    (list "Char" (list #\p))
    (list "Str" (list "cs" "CS" "Computer Science")))

* `(update empty (list "add" 1)) => (list (list "Num" (list 1)))

* `(update start (list "remove" "CS")) =>
  
  (list (list "Num" (list 5 1.3 -1))
    (list "Bool" empty)
    (list "Char" (list #\p))
    (list "Str" (list "cs" "Computer Science")))

* `(update start (list "remove" #\p)) =>
  
  (list (list "Num" (list 5 1.3 -1))
    (list "Bool" empty)
    (list "Char" empty)
    (list "Str" (list "cs" "Computer Science")))

Note that the order of the produced lists may be different with your implementation.

You may find the built-in functions `member?` and `remove` helpful.

Submit your solution in the file `a06q3.rkt`.

✥ ✥ ✥