You may talk about this assignment with others, but do not write down anything while you talk. After that, do the assignment alone. What you hand in must be your work only. See Mechanics→Homework on the class web page for details on the homework policy.

WHAT TO DO. Follow these and subsequent instructions accurately. There will be penalties for improperly formatted answers.

This assignment comes with auxiliary file template.rkt. Rename this file as usual before starting to work with it. For each question, write your code in your .rkt file where indicated, uncomment and run the test call(s), paste the resulting outputs where indicated, and put a semicolon and a space at the beginning of each output line so it turns into a comment.

ON GRADES. The following remarks should not be controversial, but apparently they are, so here we go. Code or other answers that just get the job done—in this and any other assignment or test—gets you partial credit, and better answers or code get higher grades.

For anything, simpler and more elegant is better. In addition, for code—and unless otherwise stated—recursive is better than iterative, and functional is better than imperative.¹

PREMISE ON RACKET PAIRS AND LISTS. So far we have made no distinction between a pair and a list of two elements, in either math or code. After all, a list is an ordered sequence of items and a pair is an ordered sequence of two items, so lists subsume pairs conceptually. So we could define a point on the plane to be a list of its two coordinates, and write code like the following:

```
(define (point x y) (list x y))
(define (x point) (first point))
(define (y point) (second point))
```

(define p (point 3 2))
(x p)
(y p)

and the last two calls return 3 and 2 respectively.

Mathematically, we will continue to think of a pair as a list of two items. Even as a Racket programmer, if all the code you see is your own, you could have a successful career without ever using the concept of a Racket pair. However, if you go to a party of Racketeers you would soon have awkward conversations. This is because Racket makes a distinction between a pair and a list. A Racket pair is a structure that joins two arbitrary values, and is built with the procedure `cons`:

```
(cons 3 2) which prints as '(3 . 2)
```

The dot in the last expression emphasizes that this is not a list but a pair. A list, on the other hand, is defined recursively in Racket as being either the empty list `null` (also known as `()' ) or a pair whose first item is the first item of the list and whose second item is a list. So

```
(list 3) is equivalent to (cons 3 null) and prints as '(3)
```

and

```
(list 3 2) is equivalent to (cons 3 (cons 2 null)) and prints as '(3 2)
```

(without a dot). Try these.

So conceptually, the Racket pair is the fundamental data associator, and a list is a repeated application of the pair constructor `cons`:

```
(list a_1 a_2 ... a_n) is shorthand for (cons a_1 (cons a_2 ... (cons a_n null) ... ))
```

The Racket designers could have chosen to leave the pair hidden in the language implementation without losing expressiveness. Instead, they chose to reveal this (space-efficient) construct for the programmer to use, and to provide various forms to work with pairs. We will take advantage of these forms. In particular, `car` and `cdr` (acronyms for “content of address register” and “content of data register”) access the two elements of a pair:

```
(car (cons 3 2)) is 3 and (cdr (cons 3 2)) is 2
```

1. This problem gives you some practice with Racket list manipulation forms that will be useful in subsequent problems. You have already seen some of these forms. They are sometimes more general than in the descriptions below, but it is good to start simple.

In the exercises below, you are asked to write functions. This means “write simple Racket functions that use the given forms whenever possible and allowed.” Each function definition should use `define` once and use no `let` or `let*`.

¹Imperative code has exclamation marks somewhere.
(a) The function map takes (i) a function of one argument and (ii) a list and returns the list obtained by applying the function to every element of the given list. Its simplest signature is as follows:

\[(\text{map } \text{function } \text{list}) \rightarrow \text{list}\]

Use map to write a function\(^2\) \((\text{square } \text{lst})\) that produces a new list with the squares of the numbers in the input list.

(b) Write a recursive function \((\text{my-map } \text{function } \text{lst})\) that does the same as map but does not use map. Then write a function my-square that uses my-map to do the same as square.

(c) The function apply takes (i) a function that can take any number of arguments and (ii) a list and returns a single value obtained by applying the function to the values in the list. Its simplest signature is as follows:

\[(\text{apply } \text{function } \text{list}) \rightarrow \text{value}\]

Use apply and map to write a function \((\text{sum-of-cubes } \text{lst})\) that returns the sum of the cubes of the numbers in the given list.

(d) The function filter takes (i) a predicate\(^3\) and (ii) a list and returns a list of all the elements that are in the input list and satisfy the predicate. Its signature is as follows:

\[(\text{filter } \text{predicate } \text{list}) \rightarrow \text{list}\]

Use filter to write a function \((\text{even } \text{lst})\) that produces a new list that contains exactly the numbers in the input list that are even. You may use the Racket predicate even?

(e) Use apply, map, and filter to write a function \((\text{sum-of-perfect-roots } \text{lst})\) that returns the sum of the square roots of the perfect squares in the given list. The number \(x\) is a perfect square if \((\text{exact? } (\text{sqrt } x))\) is true. Do not use your square function.

(f) The predicate and can take any number of arguments:

\[(\text{and}) \rightarrow \#t \quad \text{and} \quad (\text{and } \#t \#f \#t) \rightarrow \#f\]

To take the logical and of a list of Booleans we are tempted to write something like this:

\[(\text{apply and } '(\#t \#f \#t))\]

However, and is not a function, but rather a special form that behaves a little like cond: It scans its arguments and returns \#f as soon as it finds a false one, without evaluating further arguments. If it gets past the last argument (or if there are no arguments), it returns \#t. Because of this behavior, \((\text{apply and } ...)\) does not work. Instead, Racket provides a function andmap with signature

\[(\text{andmap } \text{predicate } \text{list}) \rightarrow \text{Boolean}\]

that returns \#t if and only if all elements in the list satisfy the predicate. For instance,

\[(\text{andmap identity } '(\#t \#f \#t)) \rightarrow \#f\]

Use andmap to write a predicate \((\text{all-even? } \text{lst})\) that returns \#t if and only if either the given list is empty or all its elements are even.

(g) In its simplest form, the function foldl takes (i) a procedure with two arguments, (ii) a value, and (iii) a list, and returns a value:

\[(\text{foldl } \text{procedure } \text{value} \text{list}) \rightarrow \text{value}\]

This function applies the procedure to the first item in the list and the input value to produce a result \(r_1\). It then applies the procedure to the second item in the list and \(r_1\) to produce a second result \(r_2\). It continues applying the procedure to item \(k\) and result \(r_{k-1}\) until the end of the list. When it is done, it returns the last result. As an example, you could define your own function reverse as follows:

\[(\text{define (reverse lst)} \ (\text{foldl cons }'(\emptyset) \text{lst}))\]

Make sure you understand this, perhaps by tracing execution of \((\text{reverse }'(1 \ 2 \ 3))\) by hand.

Use foldl to write a function \((\text{my-other-map } \text{function } \text{lst})\) that does the same as map (as described above). Then write a function \((\text{my-other-square } \text{lst})\) that does the same as \((\text{square } \text{lst})\) but uses my-other-map instead of map.

---

\(^2\)Here and elsewhere we use the identifier \(\text{lst}\) for a list, to avoid clashing with the Racket \(\text{list}\) function.

\(^3\)A predicate is a function into the set of Booleans.
2. Racket provides several facilities to work with sets. In this assignment we will not use these facilities, because our goal is to understand sets themselves. Instead, we implement a set as a list of the items in the set.

**IMPLEMENTATION REMARKS.**

- Racket already provides a predicate `empty?` to check that a list is empty, so we can use that for sets as well.
- Sets do not necessarily contain numbers, so we cannot use the predicate `=` to check for equality of two items. Use the predicate `equal?` instead (good for numbers as well in this context).
- A list is an ordered sequence, but order does not matter in sets. Making sure that it does not is one of the challenges in this problem.
- Duplicates do not matter for sets, either, but **your representation of sets should contain no duplicates** for storage efficiency. To this end, use the Racket function `remove-duplicates` to remove duplicates from lists as appropriate. This function uses `equal?` by default. For instance,
  
  
  ```racket
  (remove-duplicates '(row row row your boat)) evaluates to '(row your boat)
  ```

- To define a function that takes an arbitrary number (possibly zero) of arguments, you write something like this:

  ```racket
  (define (function . items) <body>)
  ```

  The dot is surrounded by spaces. The identifier `items` that follows the dot is bound to a list that collects any number of optional arguments passed to the function. Similarly, a function that takes, say, two mandatory arguments and zero or more optional ones would be defined as follows:

  ```racket
  (define (function arg1 arg2 . items) <body>)
  ```

  (of course, you can replace `function, arg1, arg2, items` by any other identifiers).

- The template `.rkt` file makes the following functions for you:

  - A function `empty-set` that takes no arguments and returns an empty set.
  - A function `make-set . items` that takes an arbitrary number (possibly zero) of arguments and makes a set that contains the given arguments.

  Use these functions to make sets, rather than crafting those sets directly as lists.

(a) Define a Racket function `(insert-items set . items)` that takes a set and an arbitrary number (possibly zero) of additional arguments and adds them to the given set.

  **HINT.** Use `append` and `remove-duplicates`.

(b) Define a Racket function `(remove-items set . items)` that takes a set and an arbitrary number (possibly zero) of additional arguments and removes them from the given set. Nothing bad should happen if an item is not in the set to begin with.

  **HINT.** Look up the function `remove*` in the Racket reference manual. This is almost what you need, except that you want to be able to say

  ```racket
  (remove-items set 'row 'your) rather than (remove-items '(row your) set)
  ```

  The function `remove*` uses `equal?` to check for equality.

(c) Define a Racket predicate `(contains-item? set item)` that returns `#t` if item is in set (according to `equal?`) and returns `#f` otherwise.

  **HINT.** You may find the functions `member` and `list?` useful (look them up in the reference manual).

(d) Define a Racket function `(cardinality set)` that returns the cardinality of the given set.

  **HINT.** It is OK to use the Racket function `length`.

(e) Define a Racket function `(union set-1 set-2)` that returns the union of the two given sets.

  **HINT.** If your definition of `insert-items` is simple, then it differs from that of `union` by a single dot.
(f) Define a Racket function \((\text{intersection set-1 set-2})\) that returns the intersection of the two given sets.

**HINT.** Use \text{filter} and \text{contains-item}?

(g) Define a Racket function \((\text{difference set-1 set-2})\) that returns the set difference \(S_1 \setminus S_2\) of the two given sets.

**HINT.** In their simplest implementations, the function \text{intersection} and \text{difference} differ by a single call to \text{not}.

(h) Define a Racket function \((\text{cartesian-product set-1 set-2})\) that returns the cartesian product of the two given sets. Each element of the product should be represented by a Racket pair. For instance,

\[(\text{cartesian-product (make-set 'a 'b) (make-set 'c 'd 'e))}\]

should yield

\(((a . c) (a . d) (a . e) (b . c) (b . d) (b . e))\)

(pairs in the set can be in any order, but items within a pair cannot).

**HINT.** There are several ways to do this. One uses \text{map} (twice), \text{apply}, and \text{append}. A straight recursion works as well.

(i) Define a Racket predicate \((\text{equal-sets? set-1 set-2})\) that returns \#t if an only if the two sets are equal.

(j) Define a Racket predicate \((\text{subset? set-1 set-2})\) that returns \#t if an only if the first set is a (possibly improper) subset of the second.

3. A \textit{mapping}\(^4\) between domain \(D\) and codomain \(C\) is a nonempty subset of \(D \times C\). In Racket, we can represent a mapping between finite sets using a

\[(\text{struct mapping (domain codomain pairs))}\]

where \text{domain} and \text{codomain} are \textit{sets} and \text{pairs} is a \textit{subset} of \((\text{cartesian-product} \text{ domain codomain})\). This \textit{struct} is already defined at the top of the \texttt{.rkt} file.

Sets and operations on them are implemented\(^5\) as in problem 1. Functions, surjections, injections, and bijections are defined both in FDM (page 51) and in the February 5 and 10 class notes. In the notes, tables that define mappings sometimes contain question marks. Do not use question marks here. For instance, a table with a question mark in its first column and \('r\) in the second for a mapping \(m\) simply means that the item \(r\) is in \((\text{mapping-codomain} m)\) but in no pair of \((\text{mapping-pairs} m)\).

(a) Define a Racket function \((\text{mapping-source m})\) that returns a set with all the elements that occur in some \textit{car} of \((\text{mapping-pairs} m)\).

(b) Define a Racket function \((\text{mapping-range m})\) that returns a set with all the elements that occur in some \textit{cdr} of \((\text{mapping-pairs} m)\).

(c) Define a Racket predicate\(^6\) \((\text{proper-mapping? m})\) that returns \#t if and only if \(m\) is a mapping. Each pair in a proper mapping must contain an element from the domain of \(m\) in its \textit{car} and an element from the codomain in its \textit{cdr}.

(d) Define a Racket predicate \((\text{function? m})\) that returns \#t if an only if \(m\) is a function from its domain to its codomain.

**HINT.** One strategy here is to require \(m\) to be a proper mapping and in addition to check that each element \(x\) in \((\text{mapping-domain} m)\) occurs in exactly one pair in \((\text{mapping-pairs} m)\). For this check, you could \textit{filter} the pairs in \((\text{mapping-pairs} m)\) that have their \textit{car} equal to \(x\) and verify that the resulting list has length 1. To combine results for all \(x\) in \((\text{mapping-domain} m)\) you could use the \texttt{andmap} function. However, there are many (functional, non-iterative) ways to implement this function.

(e) Define a Racket predicate \((\text{surjection? m})\) that returns \#t if an only if \(m\) is a surjection.

**HINT.** Two sets that are easy to compute need to be equal.

(f) Define a Racket predicate \((\text{injection? m})\) that returns \#t if an only if \(m\) is an injection.

**HINT.** Two sets that are easy to compute need to have the same cardinality.

(g) Define a Racket predicate \((\text{bijection? m})\) that returns \#t if an only if \(m\) is a bijection.

**What to Hand In.** At the start of class on the due date, hand in a paper printout of your \texttt{.rkt} file, stapled (not clipped) in the upper-left corner. Make sure that your name shows up on the first page. One student, one staple! We do not provide staplers in class.

Check that all functions and test outputs are visible on your printout. If any line is clipped, insert newline characters (and semicolons, for comments) as appropriate and print again.

By beginning of class on the due date, also submit your (properly named) \texttt{.rkt} file on Sakai (\textit{not} in the Sakai dropbox).

\(^4\) The word “mapping” is used instead of “map” to avoid clashing with the Racket function \texttt{map}. Other than this, “mapping” and “map” are synonyms here.

\(^5\) If you were unable to implement the necessary functions in problem 1 correctly (an only then), you may use the Racket facilities for sets in problem 2. Search for \texttt{sets} in the Racket reference manual.

\(^6\) We cannot use the identifier \texttt{mapping?} for this predicate, because the declaration of the \textit{struct} \texttt{mapping} implicitly generates a predicate \texttt{mapping?}. The latter merely checks that an object is a \textit{struct} of that type, not that it is a “proper” mapping in the sense defined here.