Assignment: 6

Due: Thursday, July 2 at 9:00 am
Language level: Beginning Student with List Abbreviations
Coverage: Modules 1–7

For this and all subsequent assignments, to receive full marks you are required to use the design recipe for every function you write. You may include the examples given in the assignment in your submissions, but they will be ignored by the markers—you must develop a complete suite of examples and tests on your own. For your convenience, an interface file which contains the headers of the required functions is available on the course webpage.

Do not send any code files to course staff; they will not be accepted. Submissions must be made via MarkUs as described on the course webpage. After submission, check your basic test results to ensure your files were properly submitted. Solutions that do not pass the basic tests are unlikely to receive any correctness marks.

Remember, the solutions you submit must be **entirely your own work**.

1. Write a Racket function *divisors* which consumes a nonzero natural number *n* and a symbol *order* which is either 'countup or 'countdown and produces a list containing the natural numbers which divide *n*. If *order* is 'countup the list of numbers should be sorted in increasing order and if *order* is 'countdown the list of numbers should be sorted in decreasing order.

   For example, (*divisors* 12 'countup) should produce (*list* 1 2 3 4 6 12) and (*divisors* 12 'countdown) should produce (*list* 12 6 4 3 2 1). Your solution should use both of the countup and countdown templates discussed in class (don't only use one of the two and use *reverse* to produce the other ordering).

2. Recall from class that an association (*As*) is a (*list Num Str*), where the first element is known as the *key* and the second element is known as the *value*, and an association list (*AL*) is a (*listof As*) whose keys are all distinct.

   One of the common things done with an association list is to add a new association to the list. Since the keys of an association list must be distinct, the key to add must not appear in the association list it is being added to.

   It is sometimes useful to drop this restriction, but in this case some method must be used to keep the keys distinct when there is a key clash, i.e., when the key of the association being added already appears in the association list. In this question, we will demonstrate four possible methods for handling key clashes.

Write a Racket function *update-al* which consumes a symbol *method*, an association *assoc*, and an association list *alst* (in that order) and produces a new association list containing *assoc* along with all of the associations in *alst*, unless there is a clash. If a clash occurs, the method of handling the clash should be determined by the value of *method*:

- If *method* is 'fail the function should produce *false*.
- If *method* is 'new, *assoc* should take precedence and be included in the produced association list.
- If *method* is 'old, the association in *alst* with the clash should take precedence and be included in the produced association list.
- If *method* is 'compare the values associated to the keys which clash should be compared and the association which has the larger value when ordered lexicographically (compare the values with *string>=?*) should take precedence and be included in the produced association list.

For example, if *al* was defined to be the association list (*list* (*list* 1 "abc")):

- (*update-al* 'fail (*list* 1 "def") *al*) should produce *false*.
- (*update-al* 'new (*list* 1 "def") *al*) should produce (*list* (*list* 1 "def")).
- (*update-al* 'old (*list* 1 "def") *al*) should produce (*list* (*list* 1 "abc")).
- (*update-al* 'compare (*list* 1 "def") *al*) should produce (*list* (*list* 1 "def")).

3. In this question, you will write functions which consume two arbitrary lists. For convenience, you may assume that the lists only contain numbers (so that the items in the lists may be compared with = instead of *equal?*).

   (a) Write a Racket function *sublist?* which consumes two lists, *lst1* and *lst2*, and determines if the first list is a sublist of the second list or not, producing *true* in the former case and *false* in the latter case. The list *lst1* is a sublist of *lst2* if and only if all the elements of *lst1* appear somewhere in *lst2* in the exact same order, with no additional elements between them.

   For example, (*sublist?* (*list* 1 2) (*list* 0 1 2 3 4)) and (*sublist?* (*list* 2 3 4) (*list* 0 1 2 3 4)) should produce *true*, but (*sublist?* (*list* 1 2 4) (*list* 0 1 2 3 4)) should produce *false*.

   (b) Write a Racket function *subseq?* which does the same thing as *sublist?* except determines if *lst1* is a subsequence of *lst2*. The definition of a subsequence is the same as a sublist except that the elements of *lst1* may appear in *lst2* with additional elements between them. In other words, *lst1* is a subsequence of *lst2* if and only if *lst1* can be obtained by removing elements from *lst2*.

   For example, (*subseq?* (*list* 1 2) (*list* 0 1 2 3 4)) and (*subseq?* (*list* 1 2 4) (*list* 0 1 2 3 4)) should produce *true*, but (*subseq?* (*list* 2 1 4) (*list* 0 1 2 3 4)) should produce *false*.