CSSE 304 Assignment 5

Objectives: You should learn

- To be very confident and competent with functional-style recursive procedures dealing with lists and with lists of lists.

The general instructions are identical to those for Assignment 4. Read them there
Assume that all arguments have the correct format. Restriction on Mutation continues.

Some practice problems. Some of these may eventually become assigned problems.
All from EoPL: Exercises 1.15, 1.17, 1.18, 1.20, 1.22, 1.24, 1.26, 1.28, 1.32, 1.35

Problem 1 uses the definitions and representations of *intervals* that are described in assignment 1.

#1 (20 points) `minimize-interval-list` Write a Scheme procedure `(minimize-interval-list ls)` that takes a nonempty list (not necessarily a set) of intervals and returns a set of intervals that has smallest cardinality among all sets of intervals whose unions are the same as the union of the list of intervals `ls`. In other words, combine as many intervals as possible. For example:

minimize-interval-list: *IntervalList* → *IntervalList*

```
(minimize-interval-list '((1 3) (2 3)))              → ((1 3))
(minimize-interval-list '((1 2) (3 4)))              → ((1 2) (3 4))
(minimize-interval-list '((1 3) (8 10) (2 4) (9 11)))  → ((1 4) (8 11))
(minimize-interval-list '((2 5) (1 7) (6 10) (10 11))) → ((1 11))
(minimize-interval-list '((1 2) (4 7) (1 2)))        → ((1 2) (4 7))
```

Hint. One possible approach: Sort the list first, sorting by the first number in each interval. What does that give us? Notice that if the first interval in the list can be merged with any other interval in the list, it can be merged with the second interval. So once the list is sorted, a straightforward recursive algorithm will produce the merged list in O(N) time. Of course, the sort itself is not O(N).
How to do the sorting? You can use Scheme's list-sort, which takes as arguments
(a) the predicate to use for comparing intervals and
(b) the list to be sorted.

#2 (5 points) Write the procedure `(exists? pred ls)` that returns `#t` if `pred` applied to any element of `ls` returns `#t`, `#f` otherwise. It should short-circuit; i.e., if it finds an element of `ls` for which `pred` returns `#t`, it should immediately return without looking at the rest of the list. You may assume that `pred` actually is a predicate, and that each element of `ls` is in the domain of `pred`. There is a built-in procedure in *Chez* Scheme (actually two of them) that does this exact thing. Of course you should write this yourself instead of using one of those.

exists?: *predicate* × *relation* → *Boolean*

Examples:

```
(exists? number? '(a b 3 c d)) → #t
(exists? number? '(a b c d e)) → #f
```

#3 (10 points) `vector-index` is like `list-index`, but its second argument is a vector, not a list. Here I mean a Scheme vector type, not the kind of vectors that we used in the first two assignments. You may not use Chez Scheme's `vector->list` procedure in your code. [**Something to think about:** When writing `list-index`, using `list-ref` as a helper procedure would be a bad idea. Can you see why? For this procedure, using `vector-ref` as a helper is really the only choice. And this is not bad. Why do I say that?]

#4 (10 points) Write the procedure (`product set1 set2`) that returns a list of 2-lists (lists of length 2) that represents the Cartesian product of the two sets. The 2-lists may appear in any order, but each 2-list must be in the correct order.

product: *set* × *set* → *set of 2-lists*

Examples:

```
      (product '(a b c) '(x y))) ➔ ((a x) (a y) (b x) (b y) (c x) (c y))
```

#5 (10 points) `replace` Write a recursive Scheme procedure `(replace old new ls)` which takes two numbers, one to be replaced and one new value, as well as a simple list of numbers. It returns a copy of `ls` with all occurrences of `old` replaced by `new`.

Examples:

```
   (replace 5 7 '(1 5 2 5 7))  ➔ (1 7 2 7 7)
   (replace 5 7 '())           ➔ ()
```

#6 (15 points) Write a recursive Scheme procedure `(remove-last element ls)` which takes a symbol and a simple list of symbols. It returns a list that contains everything in the list `ls` but the last occurrence of `element`. If `element` is not in `ls`, the new list contains all of the elements of the original list. In every case, the original list is unchanged.

Examples:

```
   (remove-last 'b '(a b c b d))  ➔ (a b c d)
   (remove-last 'b '(a c d))     ➔ (a c d)
   (remove-last 'b '(a c b d))   ➔ (a c d)
```

Piazza questions and answers from previous terms

Hint for HW5 #1, minimize-interval-list

Hint: Sort the list first, sorting by the first number in each interval. What does that give us? If the first interval in the list can be merged with any other interval in the list, it can be merged with the second interval. So once the list is sorted, a straightforward recursive algorithm will produce the merged list in O(N) time. Of course the sort itself is not O(N).

How to do the sorting? Use Scheme's list-sort, which takes as arguments the predicate to use for comparing intervals and the list to be sorted.

Sorting a list

A student writes:

```
I'm currently working on the problem 1 in A5. I've already solved this question, but
i think maybe there's a better way to do this. My question is "is there any way to
sort list of list by using list-sort, apply and map? "

Example of sort list of list:

((1 3) (4 5) (2 6)) ==> ((1 3) (2 6) (4 5))
```

Yes, there's a clever way to do this, just using list-sort. Take a look at the list-sort documentation linked to from the Summary of Forms. The summary is a useful and cool resource to let you know what all is in there. Notice that you provide your own predicates to sort by.

```
> (list-sort (lambda (n m) (<= (modulo n 10) (modulo m 10))) '(13 25 16 9 37 41 88))

(41 13 25 16 37 88 9)
```

Write the predicate that does what you want, and it sorts how you request.

#1 Are we allowed to use remove and list-tail?

I'm working on Problem #1 and I was wondering if I could use (remove v list) and (list-tail list v) in my procedures?

the instructors' answer,
Yes.

Remove last occurrence

I answered the questions in-line
Claude

The problem asks us to write a recursive Scheme procedure.
Do we need to recurse directly in the remove-last? **No**
Are we allowed to use (reverse list) and use another recursive procedure? **Yes**

Assignment 5 Problem 2 - exists?

In the previous assignment, we cannot use filter to implement filter-in. They take the exact arguments and give the same result.
Here for exists?, Scheme also has a built-in procedure that takes the same arguments and gives the same result. Can we use it?
(I know it's probably more helpful to practice implementing it... Just asking.)

the instructors' answer:
If I assign someth8ing that is already built-in to Scheme, I expect you to write it yourself, not use the built-in.

Can we use ! procedures?

the instructors' answer:
Those cause mutation, you shouldn't use mutation unless a problem statement explicitly says you can.