Homework 1

Due Wednesday, Jan 25, 2023 at 8pm ET

The learning objectives of this homework are to:

- Review recursion and gain more practice using it
- Gain experience writing code in Scheme

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/homework/hw1/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

The Python distribution code for this assignment, as well as the examples below, uses doctests to document examples and to provide a minimal set of test cases. You can run the tests from the command line as follows:

```
$ python3 -m doctest -v hw1_python.py
```

For the Scheme code in this assignment, you must write it in R5RS-compliant Scheme. The officially supported interpreter for this course is Racket[1]. Make sure you choose to run R5RS Scheme. If you use the DrRacket interface, select *Language -> Choose Language -> Other Languages -> R5RS* from the menu. You may need to click on *Run* before the interface will show that R5RS is chosen.

You may also use the `plt-r5rs` command-line interpreter included in the Racket distribution, which is also available on CAEN after running the following command:

```
module load racket
```

To run with `plt-r5rs` on your own machine, you may need to add the `bin` directory under your Racket installation to your path[2] so that the `plt-r5rs` executable can be located.

You can run the provided test cases with:

```
$ plt-r5rs hw1_scheme_tests.scm
```

The autograder for this assignment will also use `plt-r5rs`.

The following restrictions apply to the Scheme questions on this assignment:

- You must write your code in standard R5RS Scheme.
- You may only use `define` at global scope.
- You may **not** use any procedures with side effects.
- You may **not** use `do`, `for-each`, or `syntax-rules`.

[1] On MacOS, you can install Racket with Homebrew (`brew install --cask racket`).

[2] Instructions: Windows; MacOS (e.g. `export PATH="/Applications/Racket v7.4/bin:$PATH"` for a Racket 7.4 installation on MacOS; this is unnecessary if you installed Racket with Homebrew)

Exercises

1. *Recursion.* Write a recursive function in Python that divides an input sequence into a tuple of smaller sequences that each contain 4 or 5 elements from the original sequence. For example, an input sequence of 14 elements should be divided into sequences of 4, 5, and 5 elements. Use as few 5-element sequences as necessary in the result, and all 5-element sequences should be at the end. Finally, preserve the relative ordering of elements from the original sequence, and subsequences should be of the same type as the input sequence.

   Hint: You may assume that the input sequence has length at least 12. Think carefully about how many base cases you need, and what they should be. Use slicing to form subsequences, which preserves the type.

```
def group(seq):
    """Divide a sequence of >= 12 elements into groups of 4 or 5.

    Groups of 5 will be at the end. Returns a tuple of sequences,
    each corresponding to a group, with type matching that of the
    input sequence.

    >>> group(range(14))
    (range(0, 4), range(4, 9), range(9, 14))
    >>> group(tuple(range(17)))
    ((0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13, 14, 15, 16))
    """
    # add your solution below
```

2. *Scheme and recursion.* Write a recursive function `interleave` that takes two lists and returns a new list with their elements interleaved. In other words, the resulting list should have the first element of the first list, the first of the second, the second element of the first list, the second of the second, and so on. If the two lists are not the same size, then the leftover elements of the longer list should still appear at the end.

```
> (interleave '(1 3) '(2 4 6 8))
(1 2 3 4 6 8)
> (interleave '(2 4 6 8) '(1 3))
(2 1 4 3 6 8)
> (interleave '(1 3) '(1 3))
(1 1 3 3)
```

3. *List manipulation.* Write a function `add-to-all` that takes an item and a list of lists and returns a new list of lists, where each nested list is the result of prepending the first argument to the corresponding nested list in the second input argument.

```
> (add-to-all 1 '(() (2) (3 4)))
((1) (1 2) (1 3 4))
> (add-to-all '(foo bar) '((baz) ()))
(((foo bar) baz) ((foo bar)))
> (add-to-all 'qux '(((foo bar) baz) ((foo bar))))
((qux (foo bar) baz) (qux (foo bar)))
```

4. *Merge sort.* In this question, we will implement merge sort on lists of integers.

   a) Implement a `merge` function that given two sorted lists, returns a new sorted list that is the merge of the given lists.

```
> (merge '(1 3 8) '(2 4 5))
(1 2 3 4 5 8)
```

   b) Now implement a `split` function that given a list, returns a pair of lists consisting of the first and second half of the original list. The two halves should either be the exact same size, or the first half should be exactly one item larger than the second.

```
> (split '(4 8 5 3 1 2))
((4 8 5) 3 1 2)
> (split '(4 8 5))
((4 8) 5)
```

You may find the built-in `length` procedure helpful. You may also find it helpful to write a helper function.

c) Finally implement a `mergesort` function that given a list, returns its sorted counterpart.

```
> (mergesort '(4 8 5 3 1 2 6 9 7))
(1 2 3 4 5 6 7 8 9)
```

Use `split` and `merge` in your solution.

Submission

Place your solutions to question 1 in the provided `hw1_python.py` file, and the solution to the remaining questions in `hw1_scheme.scm`. Submit `hw1_python.py` and `hw1_scheme.scm` to the autograder before the deadline. Be sure to register your partnership on the autograder if you are working with a partner.