# Homework 2

*Due Fri Sep 29 at 8pm*

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 hw2_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. Make sure you choose to run R5RS Scheme in Racket. 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
```

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

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 at least 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))
    """
    pass # replace with your solution
```

2. *Higher-order functions.* Define a function `make_accumulator` in Python that returns an accumulator function, which takes one numerical argument and returns the sum of all arguments ever passed to the accumulator. Do **not** define any classes for this problem.

```
def make_accumulator():
    """Return an accumulator function that takes a single numeric
    argument and accumulates that argument into total, then
    returns total.

    >>> acc = make_accumulator()
    >>> acc(15)
    15
    >>> acc(10)
    25
    >>> acc2 = make_accumulator()
    >>> acc2(7)
```

```
    7
    >>> acc3 = acc2
    >>> acc3(6)
    13
    >>> acc2(5)
    18
    >>> acc(4)
    29
    """
    pass # replace with your solution
```

3. *Scope-based resource management.* Read the documentation on the `with` statement in Python. Then fill in the `Timer` class so that it acts as a context manager that times the code between entry and exit of a `with` statement. The `Timer` constructor should take in as a parameter the function or callable to use to read the current time. You should **not** call `time.time()` or any other built-in timing routine directly from the `Timer` class. Instead, call the function or callable that was passed to the constructor.

```
class Timer:
    """A timer class that can be used as a context manager with
    the 'with' statement. The constructor must be passed a
    function or callable to be used to determine the current
    time. Initializes the total time to 0. For each entry and
    exit pair, adds the time between the two calls to the total
    time, using the timer function or callable to read the
    current time.

    >>> class Counter:
    ...     def __init__(self):
    ...         self.count = 0
    ...     def __call__(self):
    ...         self.count += 1
    ...         return self.count - 1
    >>> t = Timer(Counter())
    >>> t.total()
    0
    >>> with t:
    ...     t.total()
    0
    >>> t.total()
    1
    >>> with t:
    ...     t.total()
    1
    >>> t.total()
    2
    >>> t2 = Timer(Counter())
    >>> with t2:
    ...     t2.total()
    0
    >>> t2.total()
    1
    >>> t.total()
    2
    """
    def __init__(self, time_fn):
        pass # replace with your solution
    def total(self):
        pass # replace with your solution
    # add any other members you need
```

4. *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)
```

5. *Context-free grammars.* Consider the following CFG, with start symbol $E$:

$$E \to T \mid T - E$$
$$T \to I \mid I + T$$
$$I \to a \mid b$$

What are the derivation trees produced for each of the following fragments? Note: ASCII art such as the following is acceptable:

```
    E
    |
    T
  / | \
I   +  T
|      |
a      I
       |
       b
```

a) $a+b+a$

b) $a+b-a$

c) $a-b+a$

d) $a-b-a$

# Submission

Place your solutions to questions 1-3 in the provided `hw2_python.py` file, and the solution to question 4 in `hw2_scheme.scm`. Write your answers to question 5 in a PDF file named `hw2.pdf`. Make sure to list any other students with whom you discussed the homework, as per course policy in the syllabus. Submit `hw2_python.py` and `hw2_scheme.scm` to the autograder before the deadline. Submit `hw2.pdf` to GradeScope before the deadline. **Pushing your work to GitHub does not submit it to the autograder!**