Homework 10: **hw10.zip (hw10.zip)**

Due by 11:59pm on Thursday, 4/25

Instructions

Download hw10.zip (hw10.zip).

Note that this homework will require you to code in both Scheme and Python. Your solutions to questions 1-3 should be placed in `hw10.scm`, and your solutions to question 4 should be placed in `hw10.py`.

> Our course uses a custom version of Scheme (which you will build for Project 4) included in the starter ZIP archive. To start the interpreter, type `python3 scheme`. To run a Scheme program interactively, type `python3 scheme -i <file.scm>`. To exit the Scheme interpreter, type `(exit)`.

Submission: When you are done, submit with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be scored. Check that you have successfully submitted your code on okpy.org (https://okpy.org/). See Lab 0 (/lab/lab00#submitting-the-assignment) for more instructions on submitting assignments.

Using Ok: If you have any questions about using Ok, please refer to this guide. (/articles/using-ok.html)

Readings: You might find the following references useful:

- Scheme Specification (/articles/scheme-spec.html)
- Scheme Primitives Reference (/articles/scheme-primitives.html)
- Tail Recursion Lecture Slides (https://drive.google.com/drive/u/0/folders/1d0WywqGhDYnU8uAjIg__kfosAmUlMHNT)
- Streams Lecture Slides (https://drive.google.com/drive/u/0/folders/1pfty5hA-fj1VD88yQF7FO4eAuQ-DhCkZ)
- Section 4.2 (http://composingprograms.com/pages/42-implicit-sequences.html)

Grading: Homework is graded based on effort, not correctness. However, there is no partial credit; you must show substantial effort on every problem to receive any points.

Q1: Accumulate

Fill in the definition for the procedure `accumulate`, which combines the first `n` natural numbers according to the following parameters:

1. `combiner`: a function of two arguments
2. `start`: a number with which to start combining

3. `n` : the number of natural numbers to combine
4. `term` : a function of one argument that computes the *n*th term of a sequence

For example, we can find the product of all the numbers from 1 to 5 by using the multiplication operator as the `combiner` , and starting our product at 1:

```
scm> (define (identity x) x)
scm> (accumulate * 1 5 identity)  ; 1 * 1 * 2 * 3 * 4 * 5
120
```

We can also find the sum of the squares of the same numbers by using the addition operator as the `combiner` and `square` as the `term` :

```
scm> (define (square x) (* x x))
scm> (accumulate + 0 5 square)  ; 0 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
55
scm> (accumulate + 5 5 square)  ; 5 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
60
```

You may assume that the `combiner` will always be commutative: i.e. the order of arguments do not matter.

```
(define (accumulate combiner start n term)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q accumulate
```

Q2: Tail Recursive Accumulate

Update your implementation of `accumulate` to be tail recursive. It should still pass all the tests for "regular" `accumulate` !

You may assume that the input `combiner` and `term` procedures are properly tail recursive.

If your implementation for accumulate in the previous question is already tail recursive, you may simply copy over that solution (replacing `accumulate` with `accumulate-tail` as appropriate).

> If you're running into an recursion depth exceeded error and you're using the staff interpreter, it's very likely your solution is not properly tail recursive.
>
> We test that your solution is tail recursive by calling `accumulate-tail` with a very large input. If your solution is not tail recursive and does not use a constant number of frames, it will not be able to successfully run.

```
(define (accumulate-tail combiner start n term)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q accumulate-tail
```

Q3: Run-Length Encoding

Run-length encoding is a very simple data compression technique, whereby runs of data are compressed and stored as a single value. A *run* is defined to be a contiguous sequence of the same number. For example, in the (finite) sequence

```
1, 1, 1, 1, 1, 6, 6, 6, 6, 2, 5, 5, 5
```

there are four runs: one each of 1, 6, 2, and 5. We can represent the same sequence as a sequence of two-element lists:

```
(1 5), (6 4), (2 1), (5 3)
```

Notice that the first element of each list is the number in a run, and the second element is the number of times that number appears in the run.

We will extend this idea to streams. Write a function called `rle` that takes in a stream of data, and returns a corresponding stream of two-element lists, which represents the run-length encoded version of the stream. You do not have to consider compressing infinite streams - the stream passed in will eventually terminate with `nil`.

```
scm> (define s (cons-stream 1 (cons-stream 1 (cons-stream 2 nil))))
s
scm> (define encoding (rle s))
encoding
scm> (car encoding)  ; Run of number 1 of length 2
(1 2)
scm> (car (cdr-stream encoding))  ; Run of number 2 of length 1
(2 1)
scm> (define s (list-to-stream '(1 1 2 2 2 3))) ; Makes a stream with the same elements as
scm> (stream-to-list (rle s))
((1 2) (2 3) (3 1))
```

```
(define (rle s)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q rle
```

Q4: Generate Paths

Define a generator function `generate_paths` which takes in a Tree `t`, a value `x`, and returns a generator object which yields each path from the root of `t` to a node that has label `x`.

Each path should be represented as a list of the labels along that path in the tree. You may yield the paths in any order.

```
def generate_paths(t, x):
    """Yields all possible paths from the root of t to a node with the label x
    as a list.

    >>> t1 = Tree(1, [Tree(2, [Tree(3), Tree(4, [Tree(6)]), Tree(5)]), Tree(5)])
    >>> print(t1)
    1
      2
        3
        4
          6
        5
      5
    >>> next(generate_paths(t1, 6))
    [1, 2, 4, 6]
    >>> path_to_5 = generate_paths(t1, 5)
    >>> sorted(list(path_to_5))
    [[1, 2, 5], [1, 5]]

    >>> t2 = Tree(0, [Tree(2, [t1])])
    >>> print(t2)
    0
      2
        1
          2
            3
            4
              6
            5
          5
    >>> path_to_2 = generate_paths(t2, 2)
    >>> sorted(list(path_to_2))
    [[0, 2], [0, 2, 1, 2]]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q generate_paths
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)