Homework 3

Due Friday Oct 20 at 8pm

1. *Decorators and memoization.* Memoization is an optimization in recursive algorithms that have repeated computations, where the arguments and result of a function call are stored when a function is first called with that set of arguments. Then if the function is called again with the same arguments, the stored value is looked up and returned rather than being recomputed.

   For this problem, implement a `memoize` decorator in Python that takes in a function and returns a version that performs memoization. The decorator should work on functions that take in any number of non-keyword arguments. You may **not** use the `functools` module in your solution.

   Hint: We recommend using a dictionary to store previously computed values, and variable argument lists to handle functions with any number of parameters.

```
def memoize(fn):
    """Return a function that computes the same result as fn, but
    if a given set of arguments has already been seen before,
    returns the previously computed result instead of repeating
    the computation. Assumes fn is a pure function (i.e. has no
    side affects), and that all arguments to fn are hashable.

    >>> @memoize
    ... def sum_to_n(n):
    ...     return 1 if n == 1 else n + sum_to_n(n - 1)
    >>> try:
    ...     sum_to_n(300)
    ...     sum_to_n(600)
    ...     sum_to_n(900)
    ...     sum_to_n(1200)
    ... except RecursionError:
    ...     print('recursion limit exceeded')
    45150
    180300
    405450
    720600
    >>> @memoize
    ... def sum_k_to_n(k, n):
    ...     return k if n == k else n + sum_k_to_n(k, n - 1)
    >>> try:
    ...     sum_k_to_n(2, 300)
    ...     sum_k_to_n(2, 600)
    ...     sum_k_to_n(2, 900)
    ...     sum_k_to_n(2, 1200)
    ... except RecursionError:
    ...     print('recursion limit exceeded')
    45149
    180299
    405449
    720599
    """
    pass # replace with your solution
```

2. *Chain.* Recall that the `compose()` higher-order function takes in two single-parameter functions as arguments and returns the composition of the two functions. Thus, `compose(f, g)(x)` computes `f(g(x))`.

The following is a definition of `compose()` in Python:

```
def compose(f, g):
    return lambda x: f(g(x))
```

Composition can be generalized to function chains, so that `chain(f, g, h)(x)` computes `f(g(h(x)))`, `chain(f, g, h, k)(x)` computes `f(g(h(k(x))))`, and so on. Implement the variadic `chain()` function in Python.

```
def chain(*funcs):
    """Returns a function that is the compositional chain of
    funcs. If funcs is empty, returns the identity function.

    >>> chain()(3)
    3
    >>> chain(lambda x: 3 * x)(3)
    9
    >>> chain(lambda x: x + 1, lambda x: 3 * x)(3)
    10
    >>> chain(lambda x: x // 2, lambda x: x + 1, lambda x: 3 * x)(3)
    5
    """
    pass # replace with your solution
```

Hint: Your solution should use recursion.

3. *Generators.*

   a) Implement a `scale()` generator that, given an iterable of numbers, scales them by a constant to produce a new iterable. For example, given a generator `naturals()` for the natural numbers, `scale(naturals(), 2)` produces an iterable of the even natural numbers.

```
def scale(items, factor):
    """Produces a new iterable that contains the elements from
    items scaled by factor. Consumes the elements from items.

    >>> def naturals():
    ...     num = 0
    ...     while True:
    ...         yield num
    ...         num += 1
    >>> values = scale(naturals(), 3)
    >>> [next(values) for i in range(5)]
    [0, 3, 6, 9, 12]
    """
    pass # replace with your solution
```

   b) Implement a `merge()` generator that, given two infinite iterables whose elements are in monotonically increasing order, produces a new iterable that contains the items from both input iterables, in increasing order and without duplicates.

```
def merge(items1, items2):
    """Produces a new iterable that contains the elements in
    increasing order from items1 and items2, without
    duplicates. Requires each of items1 and items2 to be
    infinite iterables in monotonically increasing order.
    Consumes the elements from items1 and items2.

    >>> def naturals():
    ...     num = 0
    ...     while True:
    ...         yield num
    ...         num += 1
    >>> values = merge(naturals(), naturals())
```

```
    >>> [next(values) for i in range(5)]
    [0, 1, 2, 3, 4]
    >>> values2 = merge(scale(naturals(), 2), scale(naturals(), 3))
    >>> [next(values2) for i in range(10)]
    [0, 2, 3, 4, 6, 8, 9, 10, 12, 14]
    """
    pass # replace with your solution
```

c) A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, we can build an iterable of such numbers using a generator. Let us call the required iterable of numbers `s` and notice the following facts about it.

- `s` begins with 1.
- The elements of `scale(s, 2)` are also elements of `s`.
- The same is true for `scale(s, 3)` and `scale(s, 5)`.
- These are all of the elements of `s`.

All that is left is to combine the elements from these sources, which can be done with the `merge()` generator above. Fill in the `make_s()` generator that produces the required iterable `s`.

```
def make_s():
    """Produces an iterable over all positive integers that
    only have 2, 3, or 5 as factors.

    >>> values = make_s()
    >>> [next(values) for i in range(18)]
    [1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30]
    """
    pass
```

4. *Lambda calculus.* For this question, you may use the symbol λ or the capital letter L to signify a λ.

a) Evaluate the λ-calculus term below until it is in normal form. Show each α-reduction and β-reduction step, as in the following:

$$\begin{aligned} & (\lambda x.\, x)\ (\lambda x.\, x) \\ \rightarrow\ & (\lambda x.\, x)\ (\lambda y.\, y) && (\alpha\text{-reduction}) \\ \rightarrow\ & \lambda y.\, y && (\beta\text{-reduction}) \end{aligned}$$

In plain text:

```
(L x. x) (L x. x)
-> (L x. x) (L y. y)              (alpha-reduction)
-> L y. y                         (beta-reduction)
```

Term to evaluate:

$$(\lambda x.\, \lambda y.\, x\ \lambda x.\, y\ x)\ (\lambda w.\, w)\ (\lambda x.\, x\ x)\ a$$

Hint: Evaluating this term requires a total of five β-reductions and one α-reduction.

b) The following function maps a pair containing numbers (m,n) to a pair containing $(m+1,m)$:

$$pairincr\ =\ \lambda p.\ pair\ (incr\ (first\ p))\ (first\ p)$$

Applying it to *pair m n* produces:

$$\begin{aligned} pairincr\ (pair\ m\ n)\ &=\ (\lambda p.\ pair\ (incr\ (first\ p))\ (first\ p))\ (pair\ m\ n) \\ &\rightarrow\ pair\ (incr\ (first\ (pair\ m\ n)))\ (first\ (pair\ m\ n))) \\ &\rightarrow\ pair\ (incr\ m)\ (first\ (pair\ m\ n)) \\ &\rightarrow\ pair\ (incr\ m)\ m \end{aligned}$$

Using *pairincr*, define a *decr* function that decrements a Church numeral:

$$decr\ =\ \lambda n.\ \text{[fill in your solution]}$$

Hint: What is the result when *pairincr* is applied to the pair $(0,0)$? What about when it is applied twice?

Submission

Place your solutions to questions 1-3 in the provided `hw3.py` file. Write your answers to question 4 in a PDF file named `hw3.pdf`. Make sure to list any other students with whom you discussed the homework, as per course policy in the syllabus. Submit `hw3.py` to the autograder before the deadline. Submit `hw3.pdf` to GradeScope before the deadline. **Pushing your work to GitHub does not submit it to the autograder!**