Homework 2 Solutions **hw02.zip (hw02.zip)**

Solution Files

You can find solutions for all questions in hw02.py (hw02.py).

The `construct_check` module is used in this assignment, which defines a function `check`. For example, a call such as

```
check("foo.py", "func1", ["While", "For", "Recursion"])
```

checks that the function `func1` in file `foo.py` does *not* contain any `while` or `for` constructs, and is not an overtly recursive function (i.e., one in which a function contains a call to itself by name.)

Required questions

Several doctests refer to these functions:

```
from operator import add, mul, sub

square = lambda x: x * x

identity = lambda x: x

triple = lambda x: 3 * x

increment = lambda x: x + 1
```

Q1: Make Adder with a Lambda

Implement the `make_adder` function, which takes in a number `n` and returns a function that takes in an another number `k` and returns `n + k`. Your solution must consist of a single `return` statement.

```
def make_adder(n):
    """Return a function that takes an argument K and returns N + K.

    >>> add_three = make_adder(3)
    >>> add_three(1) + add_three(2)
    9
    >>> make_adder(1)(2)
    3
    """
    return lambda k: n + k
```

Use Ok to test your code:

```
python3 ok -q make_adder
```

We can solve this with a nested `def` statement as follows:

```
def make_adder(n):
    def inner(k):
        return n + k
    return inner
```

Since the solution must be a single return statement, we simply rewrite the inner function as a `lambda` expression.

Q2: Product

The `summation(n, term)` function from the higher-order functions lecture adds up `term(1) + ... + term(n)`. Write a similar function called `product` that returns `term(1) * ... * term(n)`. **Do not use recursion.**

```
def product(n, term):
    """Return the product of the first n terms in a sequence.
    n    -- a positive integer
    term -- a function that takes one argument

    >>> product(3, identity)  # 1 * 2 * 3
    6
    >>> product(5, identity)  # 1 * 2 * 3 * 4 * 5
    120
    >>> product(3, square)    # 1^2 * 2^2 * 3^2
    36
    >>> product(5, square)    # 1^2 * 2^2 * 3^2 * 4^2 * 5^2
    14400
    >>> product(3, increment) # (1+1) * (2+1) * (3+1)
    24
    >>> product(3, triple)    # 1*3 * 2*3 * 3*3
    162
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'product', ['Recursion'])
    True
    """
    total, k = 1, 1
    while k <= n:
        total, k = term(k) * total, k + 1
    return total
```

Now, define the factorial (http://en.wikipedia.org/wiki/Factorial) function in terms of `product` in one line.

```
def factorial(n):
    """Return n factorial for n >= 0 by calling product.

    >>> factorial(4)  # 4 * 3 * 2 * 1
    24
    >>> factorial(6)  # 6 * 5 * 4 * 3 * 2 * 1
    720
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'factorial', ['Recursion', 'For', 'While'])
    True
    """
    return product(n, lambda k: k)
```

Use Ok to test your code:

```
python3 ok -q product
python3 ok -q factorial
```

The `product` function has similar structure to `summation`, but starts accumulation with the value `total=1`. Factorial is a `product` with the identity function as `term`.

Q3: Accumulate

Let's take a look at how `summation` and `product` are instances of a more general function called `accumulate`:

```
def accumulate(combiner, base, n, term):
    """Return the result of combining the first n terms in a sequence and base.
    The terms to be combined are term(1), term(2), ..., term(n).  combiner is a
    two-argument commutative, associative function.

    >>> accumulate(add, 0, 5, identity)  # 0 + 1 + 2 + 3 + 4 + 5
    15
    >>> accumulate(add, 11, 5, identity) # 11 + 1 + 2 + 3 + 4 + 5
    26
    >>> accumulate(add, 11, 0, identity) # 11
    11
    >>> accumulate(add, 11, 3, square)   # 11 + 1^2 + 2^2 + 3^2
    25
    >>> accumulate(mul, 2, 3, square)    # 2 * 1^2 * 2^2 * 3^2
    72
    >>> accumulate(lambda x, y: x + y + 1, 2, 3, square)
    19      #(((2 + 1^2 + 1) + 2^2 + 1) + 3^2 + 1)
    """
    total, k = base, 1
    while k <= n:
        total, k = combiner(total, term(k)), k + 1
    return total

# Recursive solution
def accumulate2(combiner, base, n, term):
    if n == 0:
        return base
    return combiner(term(n), accumulate2(combiner, base, n-1, term))

# Alternative recursive solution using base to keep track of total
def accumulate3(combiner, base, n, term):
    if n == 0:
        return base
    return accumulate3(combiner, combiner(base, term(n)), n-1, term)
```

`accumulate` has the following parameters:

- `term` and `n`: the same parameters as in `summation` and `product`
- `combiner`: a two-argument function that specifies how the current term is combined with the previously accumulated terms.
- `base`: value at which to start the accumulation.

For example, the result of `accumulate(add, 11, 3, square)` is

```
11 + square(1) + square(2) + square(3) = 25
```

> Note: You may assume that `combiner` is associative and commutative. That is, `combiner(a, combiner(b, c)) == combiner(combiner(a, b), c)` and `combiner(a, b) == combiner(b, a)` for all `a`, `b`, and `c`. However, you may not assume `combiner` is chosen from a fixed function set and hard-code the solution.

After implementing `accumulate`, show how `summation` and `product` can both be defined as simple calls to `accumulate`:

```
def summation_using_accumulate(n, term):
    """Returns the sum of term(1) + ... + term(n). The implementation
    uses accumulate.

    >>> summation_using_accumulate(5, square)
    55
    >>> summation_using_accumulate(5, triple)
    45
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'summation_using_accumulate',
    ...       ['Recursion', 'For', 'While'])
    True
    """
    return accumulate(add, 0, n, term)

def product_using_accumulate(n, term):
    """An implementation of product using accumulate.

    >>> product_using_accumulate(4, square)
    576
    >>> product_using_accumulate(6, triple)
    524880
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'product_using_accumulate',
    ...       ['Recursion', 'For', 'While'])
    True
    """
    return accumulate(mul, 1, n, term)
```

Use Ok to test your code:

```
python3 ok -q accumulate
python3 ok -q summation_using_accumulate
python3 ok -q product_using_accumulate
```

Both an iterative and recursive solution were allowed. Note that they are quite similar to the solution for summation! The main differences are:

- Abstracted away the method of combination (either `+` or `*`)
- Added in a starting base value, since product behaves poorly if we start with 0

Extra questions

> Extra questions are not worth extra credit and are entirely optional. They are designed to challenge you to think creatively! Feel free to skip them.

Q4: Make Repeater

Implement a function `make_repeater` so that `make_repeater(f, n)(x)` returns `f(f(...f(x)...))`, where `f` is applied `n` times. That is, `make_repeater(f, n)` returns another function that can then be applied to another argument. For example, `make_repeater(square, 3)(42)` evaluates to `square(square(square(42)))`. See if you can figure out a reasonable function to return for that case. You may use either loops or recursion in your implementation.

```
def make_repeater(f, n):
    """Return the function that computes the nth application of f.

    >>> add_three = make_repeater(increment, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 4)(5) # square(square(square(square(5))))
    152587890625
    >>> make_repeater(square, 0)(5) # Yes, it makes sense to apply the function zero time:
    5
    """
    g = identity
    while n > 0:
        g = compose1(f, g)
        n = n - 1
    return g

# Alternative solutions
def make_repeater2(f, n):
    def h(x):
        k = 0
        while k < n:
            x, k = f(x), k + 1
        return x
    return h

def make_repeater3(f, n):
    if n == 0:
        return lambda x: x
    return lambda x: f(make_repeater3(f, n - 1)(x))

def make_repeater4(f, n):
    if n == 0:
        return lambda x: x
    return compose1(f, make_repeater4(f, n - 1))

def make_repeater5(f, n):
    return accumulate(compose1, lambda x: x, n, lambda k: f)
```

For an extra challenge, try defining `make_repeater` using `compose1` and your `accumulate` function in a single one-line return statement.

```
def compose1(f, g):
    """Return a function h, such that h(x) = f(g(x))."""
    def h(x):
        return f(g(x))
    return h
```

Use Ok to test your code:

```
python3 ok -q make_repeater
```

There are many correct ways to implement `make_repeater`. The first solution above creates a new function in every iteration of the `while` statement (via `compose1`). The second solution shows that it is also possible to implement `make_repeater` by creating only a single new function. That function make_repeaterly applies `f`.

`make_repeater` can also be implemented compactly using `accumulate`, the third solution.

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)