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 the 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.) Note that this restriction does not apply to all problems in this assignment. If this restriction applies, you will see a call to `check` somewhere in the problem's doctests.

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: Product

The `summation(n, f)` function from the higher-order functions lecture adds up `f(1) + ... + f(n)`. Write a similar function called `product` that returns `f(1) * ... * f(n)`.

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

    >>> 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
    """
    total, k = 1, 1
    while k <= n:
        total, k = f(k) * total, k + 1
    return total
```

Use Ok to test your code:

```
python3 ok -q product
```

The `product` function has similar structure to `summation`, but starts accumulation with the value `total=1`.

Q2: 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, f):
    """Return the result of combining the first n terms in a sequence and base.
    The terms to be combined are f(1), f(2), ..., f(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
    >>> accumulate(lambda x, y: 2 * (x + y), 2, 3, square)
    58
    >>> accumulate(lambda x, y: (x + y) % 17, 19, 20, square)
    16
    """
    total, k = base, 1
    while k <= n:
        total, k = combiner(total, f(k)), k + 1
    return total

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

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

`accumulate` has the following parameters:

- `f` 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, f):
    """Returns the sum of f(1) + ... + f(n). The implementation
    uses accumulate.

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

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

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

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

Q3: Make Repeater

Implement the function `make_repeater` so that `make_repeater(h, n)(x)` returns `h(h(...h(x)...))`, where `h` is applied `n` times. That is, `make_repeater(h, 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)))`.

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

    >>> 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
    5
    """
    g = identity
    while n > 0:
        g = compose1(h, g)
        n = n - 1
    return g

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

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

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

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

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

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

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 `h`.

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

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Just for fun Question

> *This question is out of scope for 61a. Do it if you want an extra challenge or some practice with HOF and abstraction!*

Q4: Church numerals

The logician Alonzo Church invented a system of representing non-negative integers entirely using functions. The purpose was to show that functions are sufficient to describe all of number theory: if we have functions, we do not need to assume that numbers exist, but instead we can invent them.

Your goal in this problem is to rediscover this representation known as *Church numerals*. Here are the definitions of `zero`, as well as a function that returns one more than its argument:

```
def zero(f):
    return lambda x: x

def successor(n):
    return lambda f: lambda x: f(n(f)(x))
```

First, define functions `one` and `two` such that they have the same behavior as `successor(zero)` and `successsor(successor(zero))` respectively, but *do not call `successor` in your implementation*.

Next, implement a function `church_to_int` that converts a church numeral argument to a regular Python integer.

Finally, implement functions `add_church`, `mul_church`, and `pow_church` that perform addition, multiplication, and exponentiation on church numerals.

```
def one(f):
    """Church numeral 1: same as successor(zero)"""
    return lambda x: f(x)

def two(f):
    """Church numeral 2: same as successor(successor(zero))"""
    return lambda x: f(f(x))

three = successor(two)

def church_to_int(n):
    """Convert the Church numeral n to a Python integer.

    >>> church_to_int(zero)
    0
    >>> church_to_int(one)
    1
    >>> church_to_int(two)
    2
    >>> church_to_int(three)
    3
    """
    return n(lambda x: x + 1)(0)

def add_church(m, n):
    """Return the Church numeral for m + n, for Church numerals m and n.

    >>> church_to_int(add_church(two, three))
    5
    """
    return lambda f: lambda x: m(f)(n(f)(x))

def mul_church(m, n):
    """Return the Church numeral for m * n, for Church numerals m and n.

    >>> four = successor(three)
    >>> church_to_int(mul_church(two, three))
    6
    >>> church_to_int(mul_church(three, four))
    12
    """
    return lambda f: m(n(f))

def pow_church(m, n):
    """Return the Church numeral m ** n, for Church numerals m and n.
```

```
    >>> church_to_int(pow_church(two, three))
    8
    >>> church_to_int(pow_church(three, two))
    9
    """
    return n(m)
```

Use Ok to test your code:

```
python3 ok -q church_to_int
python3 ok -q add_church
python3 ok -q mul_church
python3 ok -q pow_church
```

Church numerals are a way to represent non-negative integers via repeated function application. The definitions of `zero`, `one`, and `two` show that each numeral is a function that takes a function and repeats it a number of times on some argument `x`.

The `church_to_int` function reveals how a Church numeral can be mapped to our normal notion of non-negative integers using the increment function.

Addition of Church numerals is function composition of the functions of `x`, while multiplication is composition of the functions of `f`.

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)