Homework 2 Solutions hw02.zip (hw02.zip)

Solution Files

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

Required questions

Several doctests refer to these functions:

```
from operator import add, mul

square = lambda x: x * x

identity = lambda x: x

triple = lambda x: 3 * x

increment = lambda x: x + 1
```

Higher Order Functions

Q1: 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)`.

```
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 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 = term(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`, which we would like to implement:

```
def accumulate(merger, base, n, term):
    """Return the result of merging the first n terms in a sequence and base.
    The terms to be merged are term(1), term(2), ..., term(n). merger is a
    two-argument commutative 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
    >>> # 2 + (1^2 + 1) + (2^2 + 1) + (3^2 + 1)
    >>> accumulate(lambda x, y: x + y + 1, 2, 3, square)
    19
    >>> # ((2 * 1^2 * 2) * 2^2 * 2) * 3^2 * 2
    >>> accumulate(lambda x, y: 2 * x * y, 2, 3, square)
    576
    >>> accumulate(lambda x, y: (x + y) % 17, 19, 20, square)
    16
    """
    total, k = base, 1
    while k <= n:
        total, k = merger(total, term(k)), k + 1
    return total

# Alternative solution
def accumulate_reverse(merger, base, n, term):
    total, k = base, n
    while k >= 1:
        total, k = merger(total, term(k)), k - 1
    return total

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

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

```
        return base
    return accumulate3(merger, merger(base, term(n)), n-1, term)
```

`accumulate` has the following parameters:

- `term` and `n`: the same parameters as in `summation` and `product`
- `merger`: a two-argument function that specifies how the current term is merged 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 `merger` is commutative. That is, `merger(a, b) == merger(b, a)` for all `a`, `b`, and `c`. However, you may not assume `merger` 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 function calls to `accumulate`.

Important: You should have a single line of code (which should be a `return` statement) in each of your implementations for `summation_using_accumulate` and `product_using_accumulate`, which the syntax check will check for.

```
def summation_using_accumulate(n, term):
    """Returns the sum: term(1) + ... + term(n), using accumulate.

    >>> summation_using_accumulate(5, square)
    55
    >>> summation_using_accumulate(5, triple)
    45
    """
    return accumulate(add, 0, n, term)

def product_using_accumulate(n, term):
    """Returns the product: term(1) * ... * term(n), using accumulate.

    >>> product_using_accumulate(4, square)
    576
    >>> product_using_accumulate(6, triple)
    524880
    """
    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
```

The syntax check will run automatically when you submit the assignment, but you can also run the check directly by running the following command.

Use Ok to test your code:

```
python3 ok -q accumulate_syntax_check
```

We want to abstract the logic of `product` and `summation` into `accumulate`. The differences between `product` and `summation` are:

- How to merge terms. For `product`, we merge via `*` (`mul`). For `summation`, we merge via `+` (add).
- The starting base value. For `product`, we want to start off with 1 since starting with 0 means that our result (via multiplying with the base) will always be 0. For `summation`, we want to start off with 0 so we're not adding any extra value what we are trying to sum.

We can then define the merge method as `merger` and the starting base value as `base` inside our `accumulate`, with the remaining logic of going through the loop being similar to `product` and `summation`.

Once we've defined `accumulate`, we can now implement `product_using_accumulate` and `summation_using_accumulate` by passing in the appropriate parameters to `accumulate`.

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Just for fun Question

> This question is out of scope for 61A. You can try it if you want an extra challenge, but it's just a puzzle that is not required or recommended at all. Almost all students will skip it, and that's fine.
>
> If you're interested in learning more about this, feel free to attend the Extra Topics (https://cs61a.org/extra/) lectures.

Q3: 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`.