Homework 2: Higher Order Functions

hw02.zip (hw02.zip)

Due by 11:59pm on Thursday, September 8

Instructions

This homework was originally released with four questions but has been reduced to two.

Download hw02.zip (hw02.zip). Inside the archive, you will find a file called hw02.py (hw02.py), along with a copy of the `ok` autograder.

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)

Readings: You might find the following references useful:

- Section 1.6 (https://composingprograms.com/pages/16-higher-order-functions.html)

Grading: Homework is graded based on correctness. Each incorrect problem will decrease the total score by one point. There is a homework recovery policy as stated in the syllabus. **This homework is out of 2 points.**

Required questions

Getting Started Videos

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
```

Getting Started Videos

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
```


Q1: Product

Write a 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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q product
```

Q2: Accumulate

Let's take a look at how `product` is an instance of a more general function called `accumulate`, which we would like to implement:

```
def accumulate(merger, start, n, term):
    """Return the result of merging the first n terms in a sequence and start.
    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
    """
    "*** YOUR CODE HERE ***"
```

`accumulate` has the following parameters:

- `term` and `n`: the same parameters as in `product`
- `merger`: a two-argument function that specifies how the current term is merged with the previously accumulated terms.
- `start`: 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` and `b`. 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(0) + ... + term(n), using accumulate.

    >>> summation_using_accumulate(5, square)
    55
    >>> summation_using_accumulate(5, triple)
    45
    >>> # You aren't expected to understand the code of this test.
    >>> # Check that the bodies of the functions are just return statements.
    >>> # If this errors, make sure you have removed the "***YOUR CODE HERE***".
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(summation_using_accumul
    ['Expr', 'Return']
    """
    "*** YOUR CODE HERE ***"

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
    >>> # You aren't expected to understand the code of this test.
    >>> # Check that the bodies of the functions are just return statements.
    >>> # If this errors, make sure you have removed the "***YOUR CODE HERE***".
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(product_using_accumulat
    ['Expr', 'Return']
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

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

Takeaway: Notice how quick it is now to create accumulator functions with different `merger` functions! This is because we abstracted away the logic of `product` and `summation` into the `accumulate` function. Without this abstraction, our code for a `summation` function would be just as long as our code for the `product` function from Question 1, and the logic would be highly redundant!

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Exam Practice

Homework assignments will also contain prior exam questions for you to try. These questions have no submission component; feel free to attempt them if you'd like some practice!

> Note that exams from Spring 2020, Fall 2020, and Spring 2021 gave students access to an interpreter, so the question format may be different than other years. Regardless, the questions below are good problems to try *without* access to an interpreter.

1. Fall 2019 MT1 Q3: You Again (https://cs61a.org/exam/fa19/mt1/61a-fa19-mt1.pdf#page=4) [Higher Order Functions]
2. Spring 2021 MT1 Q4: Domain on the Range (https://cs61a.org/exam/sp21/mt1/61a-sp21-mt1.pdf#page=14) [Higher Order Functions]
3. Fall 2021 MT1 Q1b: tik (https://cs61a.org/exam/fa21/mt1/61a-fa21-mt1.pdf#page=4) [Functions and Expressions]