Homework 2: Higher-Order Functions

hw02.zip (hw02.zip)

Due by 11:59pm on Thursday, September 12

Instructions

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 the assignment by uploading all code files you've edited to Gradescope. 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 Gradescope. See Lab 0 (../../lab/lab00#task-c-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://www.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. **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
```

Higher-Order Functions

Q1: Product

Write a function called `product` that returns the product of the first `n` terms of a sequence. Specifically, `product` takes in an integer `n` and `term`, a single-argument function that determines a sequence. (That is, `term(i)` gives the `i` th term of the sequence.) `product(n, term)` should return `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(fuse, start, n, term):
    """Return the result of fusing together the first n terms in a sequence
    and start.  The terms to be fused are term(1), term(2), ..., term(n).
    The function fuse 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 (fuse is never used)
    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
    """
    "*** YOUR CODE HERE ***"
```

`accumulate` has the following parameters:

- `fuse`: a two-argument function that specifies how the current term is fused with the previously accumulated terms
- `start`: value at which to start the accumulation
- `n`: a non-negative integer indicating the number of terms to fuse
- `term`: a single-argument function; `term(i)` is the `i` th term of the sequence

Implement `accumulate`, which fuses the first `n` terms of the sequence defined by `term` with the `start` value using the `fuse` function.

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

```
add(11,  add(square(1), add(square(2),  square(3)))) =
    11 +     square(1) +    square(2) + square(3)    =
    11 +     1         +    4         + 9            = 25
```

> Assume that `fuse` is commutative, `fuse(a, b) == fuse(b, a)`, and associative, `fuse(fuse(a, b), c) == fuse(a, fuse(b, c))`.

Then, implement `summation` (from lecture) and `product` as one-line calls to `accumulate`.

> **Important:** Both `summation_using_accumulate` and `product_using_accumulate` should be implemented with a single line of code starting with `return`.

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

    >>> summation_using_accumulate(5, square) # square(1) + square(2) + ... + square(4)
    55
    >>> summation_using_accumulate(5, triple) # triple(1) + triple(2) + ... + triple(4)
    45
    >>> # This test checks that the body of the function is just a return statement.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(summation_using_accumula
    ['Expr', 'Return']
    """
    return ____

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

    >>> product_using_accumulate(4, square) # square(1) * square(2) * square(3) * squar
    576
    >>> product_using_accumulate(6, triple) # triple(1) * triple(2) * ... * triple(5) *
    524880
    >>> # This test checks that the body of the function is just a return statement.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(product_using_accumulate
    ['Expr', 'Return']
    """
    return ____
```

Use Ok to test your code:

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

Q3: Make Repeater

Implement the function `make_repeater` which takes a one-argument function `f` and a positive integer `n`. It returns a one-argument function, where `make_repeater(f, n)(x)` returns the value of `f(f(...f(x)...))` in which `f` is applied `n` times to `x`. For example, `make_repeater(square, 3)(5)` squares 5 three times and returns 390625, just like `square(square(square(5)))`.

```
def make_repeater(f, n):
    """Returns 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, 3)(5) # square(square(square(5)))
    390625
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_repeater
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit Assignment

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../../lab/lab00/#submit-with-gradescope) has detailed instructions.

[Optional] Exam Practice

Here are some related questions from past exams for you to try. These are optional. There is no way to submit them.

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