Instructions

Please download homework materials hw02.zip from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 2. The starter code for these problems is provided in hw02.py, which is distributed as part of the homework materials in the code directory.

We have also prepared an optional problem just for fun in section 3. You can find further descriptions there.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- [Section 1.6](#)

The `construct_check` module in `construct_check.py` 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 Problems

In this section, you are required to complete the problems below and submit your code to OJ website.

Several doctests refer to these functions:

```python
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
```

Remember, you can use `ok` to test your code:

```
$ python ok  # test all functions
$ python ok -q <func>  # test single function
```

Compose Function (50pts)

Define a function compose so that `compose(h, g)(x)` returns `h(g(x))`. That is, `compose(h, g)` returns another function a function f, such that `f(x) = h(g(x))`.

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

    >>> compose(square, triple)(5)
    225
    >>> double_inc = compose(increment, increment)
    >>> double_inc(3)
    5
    >>> double_inc(4)
    6
    """
    "*** YOUR CODE HERE ***"
```

Product (100pts)

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)`.

```python
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
    """
    "*** YOUR CODE HERE ***"
```

Accumulate (150pts)

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

```python
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
    """
    "*** YOUR CODE HERE ***"
```

`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`:

```python
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
    """
    "*** YOUR CODE HERE ***"

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
    """
    "*** YOUR CODE HERE ***"
```

Make Repeater (100pts)

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 zero times!
    5
    """
    "*** YOUR CODE HERE ***"
```

For an extra challenge, try defining `make_repeater` in a single one-line return statement using your `compose` function (defined in problem 1) and your `accumulate` function (defined in problem 3).

Protected Secret (100pts)

Write a function `protected_secret` which takes in a `password`, `secret`, and `num_attempts`.

`protected_secret` should return another function which takes in a password and prints `secret` if the password entered matches the `password` given as an argument to `protected_secret`. Otherwise, the returned function should print "INCORRECT PASSWORD". After `num_attempts` incorrect passwords are used, the secret is locked forever and the function should print "SECRET LOCKED".

We recommend you using self-referencing functions to achieve this problem.

For example:

```
>>> my_secret = protected_secret("oski2021", "The view from the top of the
Campanile.", 1)
>>> my_secret = my_secret("oski2021")
The view from the top of the Campanile.
>>> my_secret = my_secret("goBears!")
INCORRECT PASSWORD # 0 Attempts left
>>> my_secret = my_secret("zoomUniversity")
SECRET LOCKED
```

See the doctests for a detailed example.

```python
def protected_secret(password, secret, num_attempts):
    """
    Returns a function which takes in a password and prints the SECRET if the
    password entered matches
    the PASSWORD given to protected_secret. Otherwise it prints "INCORRECT
    PASSWORD". After NUM_ATTEMPTS
    incorrect passwords are entered, the secret is locked and the function
    should print "SECRET LOCKED".

    >>> my_secret = protected_secret("correcthorsebatterystaple", "I love
    UCB", 2)
    >>> my_secret = my_secret("hax0r_1") # 2 attempts left
    INCORRECT PASSWORD
    >>> my_secret = my_secret("correcthorsebatterystaple")
    I love UCB
    >>> my_secret = my_secret("hax0r_2") # 1 attempt left
    INCORRECT PASSWORD
    >>> my_secret = my_secret("hax0r_3") # No attempts left
    SECRET LOCKED
    >>> my_secret = my_secret("correcthorsebatterystaple")
    SECRET LOCKED
    """
    def get_secret(password_attempt):
        "*** YOUR CODE HERE ***"
    return get_secret
```

Just for fun Problems

This section is out of scope for our course, so the problems below is optional. That is, the problems in this section **don't** count for your final score and **don't** have any deadline. Do it at any time if you want an extra challenge or some practice with higher order function and abstraction!

To check the correctness of your answer, you can submit your code to Contest 'Just for fun'.

Church numerals (Optional, 4 test cases)

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:

```python
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 `successor(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. For deeper understanding, `church2int`, iteration or recursion is forbidden from your implementation.

```python
def one(f):
    """Church numeral 1: same as successor(zero)"""
    "*** YOUR CODE HERE ***"

def two(f):
    """Church numeral 2: same as successor(successor(zero))"""
    "*** YOUR CODE HERE ***"

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
    """
    "*** YOUR CODE HERE ***"

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
    """
    "*** YOUR CODE HERE ***"

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
    """
    "*** YOUR CODE HERE ***"

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
    """
    "*** YOUR CODE HERE ***"
```

Remember to use Ok to test your code:

```
$ python ok -q <func>
```