**Expressions**

**QUESTIONS**

1. Determine the result of evaluating \( f(4) \) in the Python interpreter if the following functions are defined:

   ```python
   from operators import add
   def double(x):
       return x + x

   def square(y):
       return y * y

   def f(z):
       add(square(double(z)), 1)

   f(4)
   
   None!
   But if we include the return keyword under the definition of \( f \) the answer is 65.
   ```

**Pure Functions vs. Non-Pure Functions (pay attention to domain and range!)

**QUESTIONS**

2. What do you think Python will print for the following? Assume this definition of the `om` and `nom` procedure:

   ```python
   def om(foo):
       return -foo

   def nom(foo):
       print(foo)

   >>> nom(4)
   4
   >>> om(-4)
   4
   >>> save1 = nom(4)
   4
   >>> save1 + 1
   error
   >>> save2 = om(-4)
   >>> save2 + 1
   5
   ```
Expression Trees

QUESTIONS

3. \(\text{add} (\text{square} (\text{mul} (5, 6)), \text{add} (\text{mul} (1, 2), 7))\)

4. \((2 + 3 \times 4) / 7\)

5. \(\text{print} (\text{print} (\text{abs} (-4)))\)
First Edition Environment Diagrams

Fill out the environment diagram, the environments and values as well as the expression tree, for the following code:

1. >>> def a(n):
   ...     return n + 2
>>> def b(m):
   ...     return a(n)/2
>>> def e(n):
   ...     return a(b(n * 2) + 1)
>>> n = 5
>>> e(n)
6.5

2. >>> def bar(x):
   ...     return x + x
>>> def foo(x):
   ...     return bar(add(bar(x),1))
>>> garply = foo
>>> garply(5)
22
Procedures as arguments

```python
def double_every_number(n):
    i = 1
    while n >= i:
        print(double(i))
        i += 1

def square_every_number(n):
    i = 1
    while n >= i:
        print(square(i))
        i += 1
```

QUESTIONS

1. Now implement the function `every` which takes in a function `func` and a number `n`, and applies that function to the first `n` numbers from 1 and prints the result along the way:

```python
def every(func, n):
    i = 1
    while n >= i:
        print(func(i))
        i += 1
```

2. Similarly, try implementing the function `keep`, which takes in a predicate `pred` and a number `n`, and only prints a number from 1 to `n` to the screen if it fulfills the predicate:

```python
def keep(pred, n):
    i = 1
    while n >= i:
        if pred(i):
            print(i)
        i += 1
```

Procedures as return values

QUESTIONS

Draw the expression tree for the following expression calls assuming we have imported the correct functions.

3. Write a procedure `and_add_one` that takes a function `f` as an argument (such that `f` is a function of one argument). It **should return a function** that takes one argument, and does the same thing as `f`, except adds one to the result.

```python
def and_add_one(f):
    def new_function(x):
        return f(x) + 1
    return new_function
```

4. Write a procedure `and_add` that takes a function `f` and a number `n` as arguments. It **should return a function** that takes one argument, and does the same thing as the function argument, except adds `n` to the result.
def and_add(f, n):
    def new_func(x):
        return f(x) + n
    return new_func

5. Python represents a programming community, and in order for things to run smoothly, there are some standards in order to keep things consistent. The following is the recommended style for coding so that collaboration with other python programmers becomes standard and easy. Write your code at the very end:

def identity(x):
    return x

def lazy_accumulate(f, start, n, term):
    ""
    Takes the same arguments as accumulate from homework and returns a function that takes a second integer m and will return the result of accumulating the first n numbers starting at 1 using f and combining that with the next m integers.
    Arguments:
    f - the function for the first set of numbers.
    start - the value to combine with the first value in the sequence.
    n - the stopping point for the first set of numbers.
    term - function to be applied to each number before combining.
    Returns:
    A function (call it h) h(m) where m is the number of additional values to combine.
    >>> # This does (1 + 2 + 3 + 4 + 5) + (6 + 7 + 8 + 9 + 10)
    >>> lazy_accumulate(add, 0, 5, identity)(5)
    55
    ""
    def second_accumulate(m):
        return accumulate(f, 0, n + m, term)
    return second_accumulate