1. (12 points) In-N-Out

For each of the expressions in the tables below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. No errors occur.

The first two rows have been provided as examples of the behavior of the built-in `pow` and `print` functions.

Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started Python 3 and executed the following statements:

```
from operator import add

def re(peat):
    return print(peat, peat)

def cheap(eat):
    car, seat = re, print
    seat(car(eat))
    return double(eat)

def double(double):
    if double:
        return double + double
    elif car(double)(print)(print):
        return 1000
    else:
        return seat(3)

seat = double
car = lambda c: lambda a: lambda r: r(5, a(c))
```

Expression	Interactive Output
`pow(2, 3)`	8
`print(4, 5)`	4 5
`print(re(1+2), print(4))`	
`cheap(3)`	
`cheap(seat(2))`	

Expression	Interactive Output
`car(1)(double)(pow)`	
`double(print(1))`	
`car(0)(seat)(add)`	

2. (14 points) Supernatural

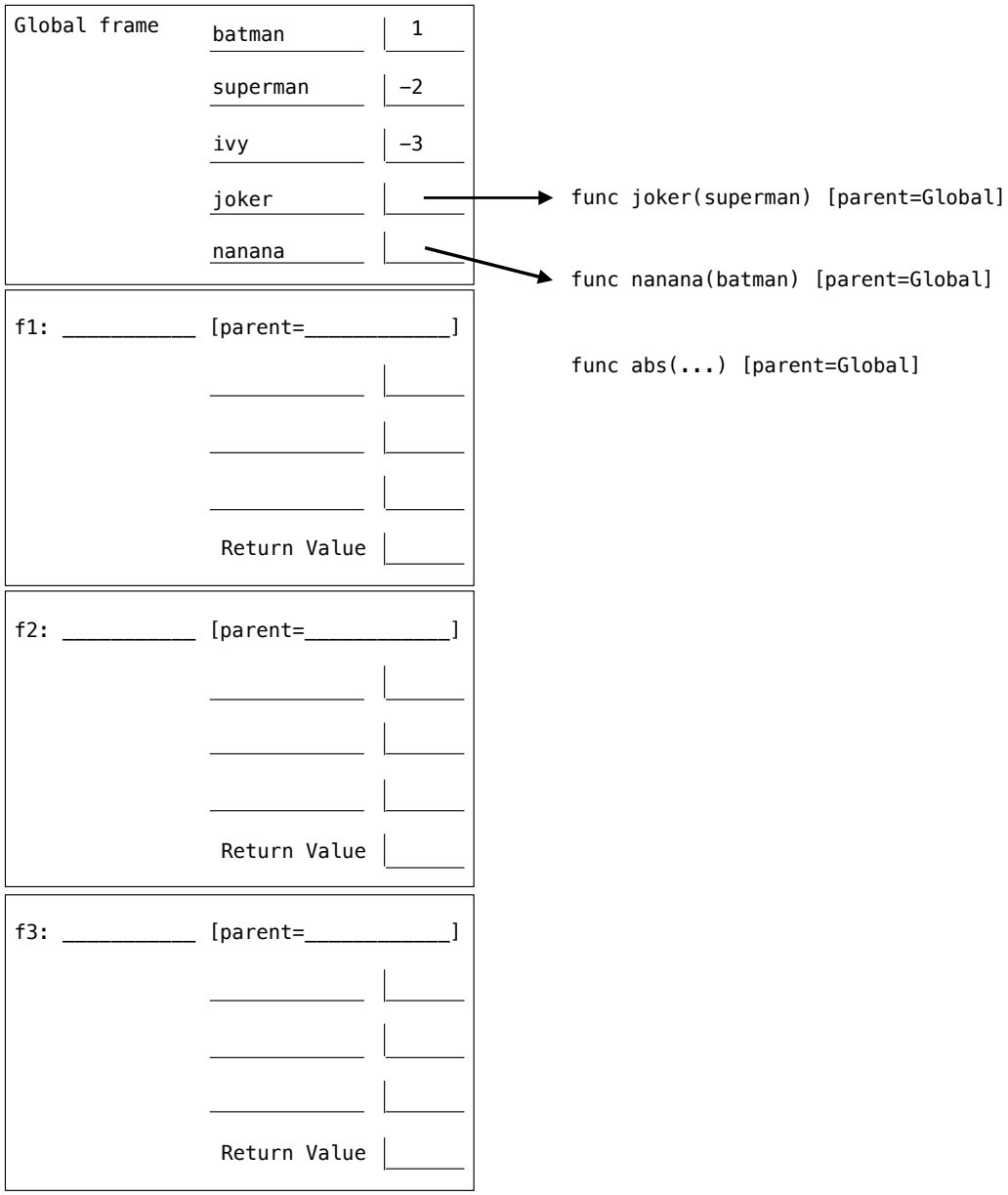
(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

Remember: Do not add a new frame when calling a built-in function (such as `abs`).

```
1  batman, superman, ivy = 1, -2, -3
2
3  def nanana(batman):
4      while batman(superman) > ivy:
5          def batman(joker):
6              return ivy
7      return -ivy
8
9  def joker(superman):
10     if superman(batman):
11         ivy = -batman
12     return nanana
13
14 joker(abs)(abs)
```

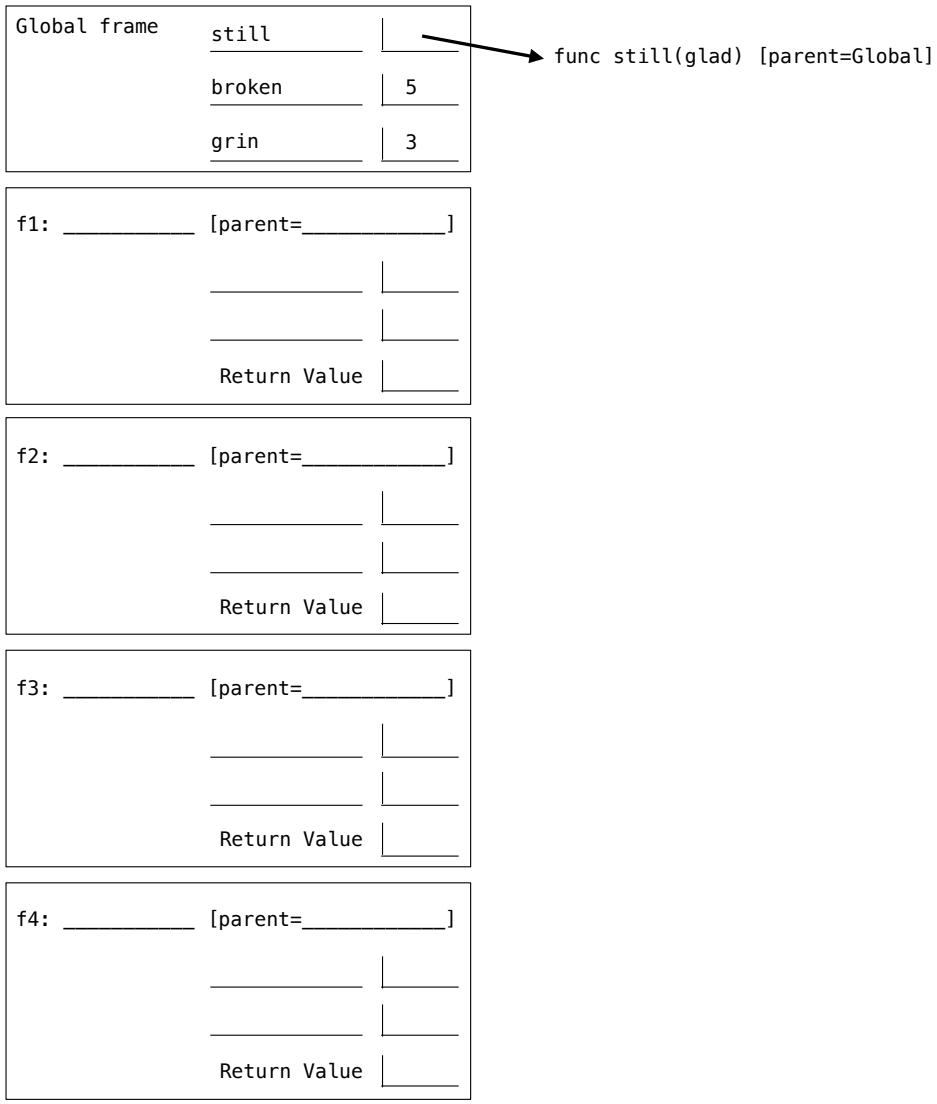


(b) **(8 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1 def still(glad):
2     def heart(broken):
3         glad = lambda heart: lambda: heart-broken
4         return glad(grin)
5     return heart(glad-grin)()
6
7 broken, grin = 5, 3
8 still(broken-1)
9
```



3. (14 points) You Complete Me

(a) (4 pt) Implement the `longest_increasing_suffix` function, which returns the longest suffix (end) of a positive integer that consists of strictly increasing digits.

```
def longest_increasing_suffix(n):
    """Return the longest increasing suffix of a positive integer n.

    >>> longest_increasing_suffix(63134)
    134
    >>> longest_increasing_suffix(233)
    3
    >>> longest_increasing_suffix(5689)
    5689
    >>> longest_increasing_suffix(568901) # 01 is the suffix, displayed as 1
    1
    """

    m, suffix, k = 10, 0, 1

    while n:

        ______________________________________________, last = n // 10, n % 10


        if ____________________________________________________________________:


            m, suffix, k = _____________, ___________________________, 10 * k

        else:

            return suffix

    return suffix
```

(b) (3 pt) Add parentheses and single-digit integers in the blanks below so that the expression on the second line evaluates to 2015. **You may only add parentheses and single-digit integers.** You may leave some blanks empty.

```
lamb = lambda lamb: lambda: lamb + lamb


lamb(1000)______ + (lambda b, c: b______  *  b______  -  c______)(lamb(______), 1)______
```

(c) (3 pt) Implement the `combine` function, which takes a non-negative integer `n`, a two-argument function `f`, and a number `result`. It applies `f` to the first digit of `n` and the result of combining the rest of the digits of `n` by repeatedly applying `f` (see the doctests). If `n` has no digits (because it is zero), `combine` returns `result`.

```
from operator import add, mul

def combine(n, f, result):
    """Combine the digits in non-negative integer n using f.

    >>> combine(3, mul, 2) # mul(3, 2)
    6
    >>> combine(43, mul, 2) # mul(4, mul(3, 2))
    24
    >>> combine(6502, add, 3) # add(6, add(5, add(0, add(2, 3)))
    16
    >>> combine(239, pow, 0) # pow(2, pow(3, pow(9, 0)))
    8
    """
    if n == 0:

        return result

    else:

        return combine(__________, __________, ____________________________)
```

(d) (4 pt) Implement the `memory` function, which takes a number `x` and a single-argument function `f`. It returns a function with a peculiar behavior that you must discover from the doctests. **You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.**

```
square = lambda x: x * x
double = lambda x: 2 * x

def memory(x, f):
    """Return a higher-order function that prints its memories.

    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):

        print(_____________________________________________________________)


        return ____________________________________________________________

    return g
```

1. (12 points) "If (s)he can wield the Hammer..."

For each of the expressions in the tables below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error".

The first three rows have been provided as examples.

Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started `python3` and executed the following statements:

```
from operator import add

avengers = 6

def vision(avengers):
    print(avengers)
    return avengers + 1

def hawkeye(thor, hulk):
    love = lambda black_widow: add(black_widow, hulk)
    return thor(love)

def hammer(worthy, stone):
    if worthy(stone) < stone:
        return stone
    elif worthy(stone) > stone:
        return -stone
    return 0

capt = lambda iron_man: iron_man(avengers)
```

Expression	Interactive Output
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`capt(vision)`	
`print(print(1), vision(2))`	
`hawkeye(hammer, 3)`	
`hawkeye(capt, 3)`	
`hammer(lambda ultron: ultron, -1)`	
`hammer(vision, avengers)`	

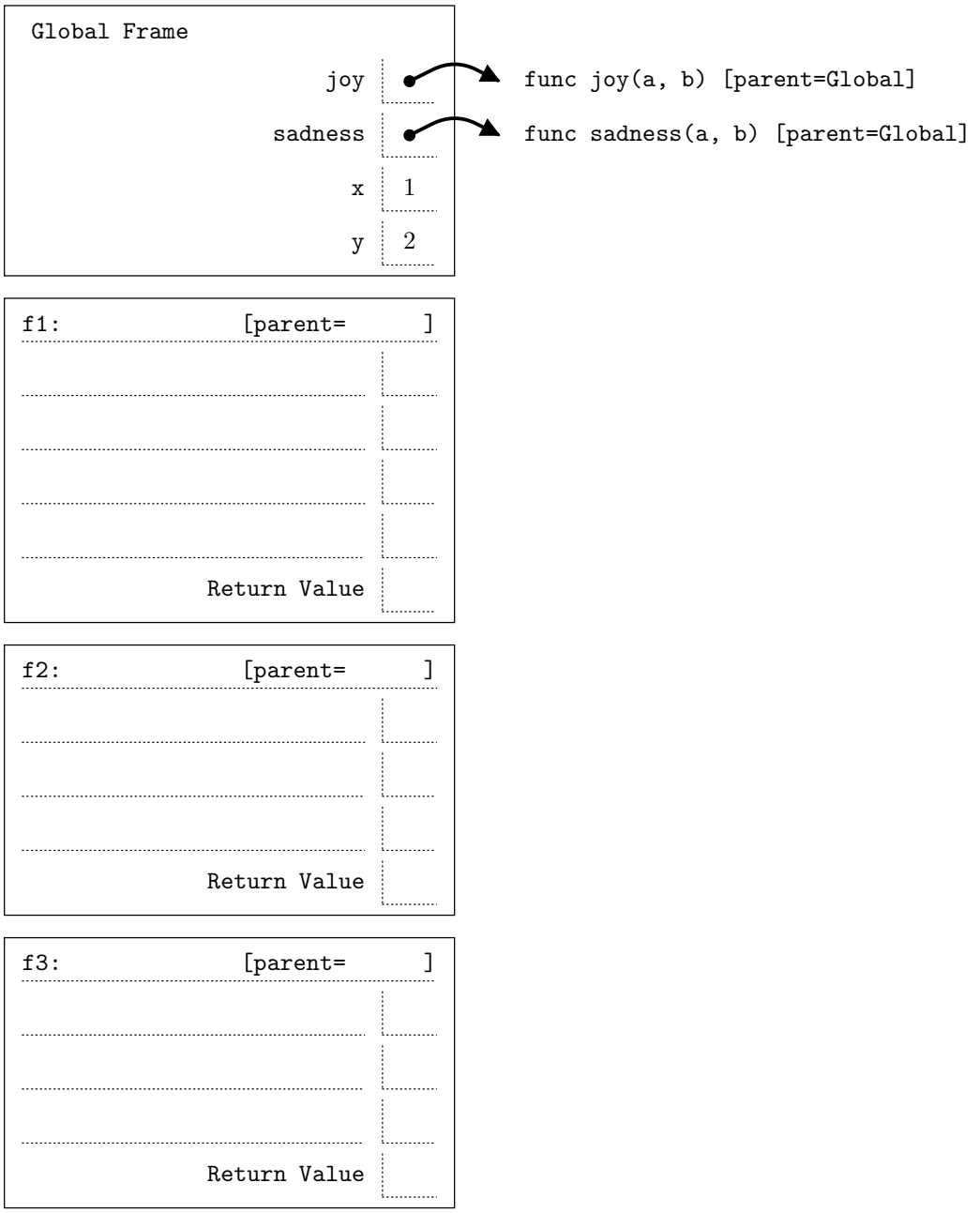
2. (14 points) "You'll get lost in there." "C'mon! Think positive!"

(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You only need to show the final state of each frame. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  def joy(a, b):
2      x = 42
3      y = sadness(a + b, x)
4      return x - y
5
6  def sadness(a, b):
7      a, b = a + x, a - y
8      return a // b
9
10 x, y = 1, 2
11 joy(x, y)
```

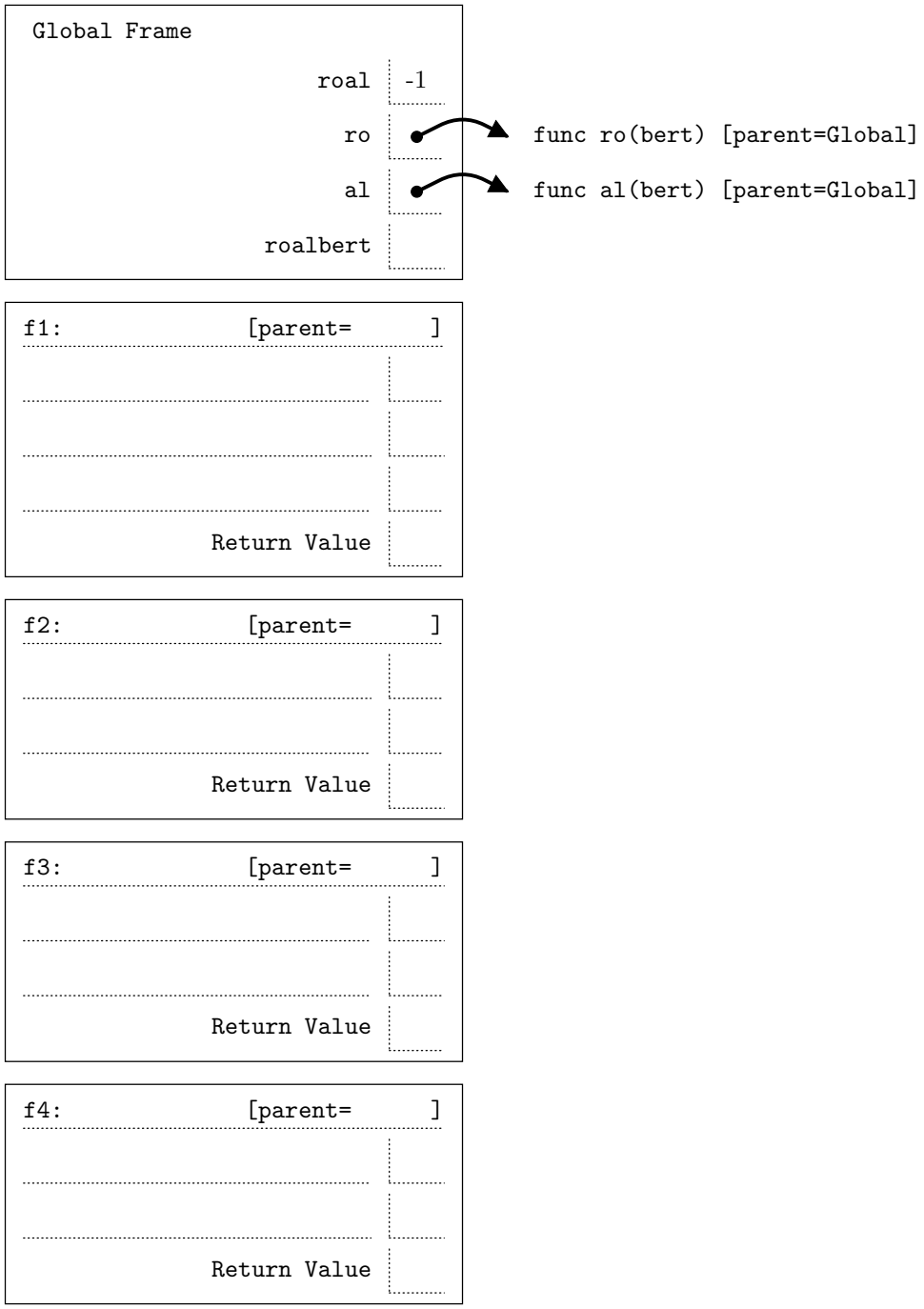


(b) **(8 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You only need to show the final state of each frame. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  roal = -1
2
3  def ro(bert):
4      ar = lambda: bert + 3
5      bert = 2 * bert
6      return ar
7
8  def al(bert):
9      roal = 5
10     return bert()
11
12 roalbert = ro(al(lambda: roal))
13 roalbert()
```



3. (12 points) "After a long day of Turing tests, you gotta unwind."

(a) (3 pt) Ava and Caleb are playing a guessing game: Ava chooses a secret number between a pre-determined range of integers, and Caleb has to figure out the secret with as few guesses as possible. Every time Caleb guesses, Ava will tell him if the secret is higher (1), lower (-1), or equal (0) to his guess.

For example, Caleb and Ava agree the secret will be between 0 and 100 (inclusive). Ava picks the secret to be 40. Caleb first guesses 50; since 40 (the secret) is lower than 50 (Caleb's guess), Ava responds with -1. The process continues until Caleb guesses the secret.

First, help Ava implement `make_direction(secret)`, which takes in a `secret` number and returns another function called `direction`. `direction` takes one argument called `guess` and compares the `guess` to the `secret` number. See the doctests for example behavior.

```
def make_direction(secret):
    """Returns a function that compares the guess to the secret.

    >>> direction = make_direction(40)
    >>> direction(50)    # 40 is lower than 50
    -1
    >>> direction(13)    # 40 is higher than 13
    1
    >>> direction(40)    # 40 is equal to 40
    0
    """
```

(b) (3 pt) Caleb will first try a naive guessing method: his first guess will be the number in the middle of the range; depending on what Ava's `direction` function says, Caleb will move one number at a time closer to the secret.

For example, suppose the range is 0 to 100 (inclusive). Caleb starts at 50. Ava's `direction` function returns -1, indicating that the `secret` is lower than 50. Caleb then tries 49; Ava again responds with -1. This continues until Caleb reaches the secret number 40, at which point Ava responds with 0. During this process, Caleb makes 11 guesses (50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40).

Implement a function called `naive_search` that takes in a `direction` function (as returned by `make_direction`) and returns the number of guesses it takes for Caleb's naive searching method to find the secret. `naive_search` should also print out the sequence of guesses.

```
def naive_search(low, high, direction):
    """Guess the secret number, as specified by direction, in a naive way;
    returns the number of guesses made.  Starts with an initial guess and
    moves up or down one number at a time.

    >>> count1 = naive_search(0, 100, make_direction(50))
    50
    >>> count1
    1

    >>> count2 = naive_search(0, 20, make_direction(14))
    10
    11
    12
    13
    14
    >>> count2
    5
    """
    guess, count = (low + high) // 2, 1

    print(guess)

    sign = ___________________________________

    while ___________________________________:

        guess = ____________________

        count = ____________________

        sign = ______________________________

        ____________________________________

    return count
```

(c) (1 pt) Let $n = \texttt{high} - \texttt{low}$ (that is, n is the number of integers that could be the secret). Circle the order of growth that best describes the runtime of Caleb's naive search method. Choose the tightest bound.

$O(1)$ $O(\log n)$ $O(n)$ $O(n^2)$ $O(2^n)$

(d) (4 pt) Ava suggests using a technique called *binary search*. Caleb's first guess should still be the number in the middle of the range.

- If the secret is lower than the guess, Caleb performs binary search again on a new range from `low` to `guess`.
- If the secret is higher than the guess, Caleb performs binary search again on a new range from `guess` to `high`.

For example, suppose the range is 0 to 100 (inclusive). Caleb starts at 50. Ava responds with -1, indicating the secret is lower than 50. Caleb then performs binary search on the range 0 to 50 (inclusive); his next guess is 25. Ava now responds with 1, indicating the secret is higher than 25. Continuing with this process, Caleb makes 5 guesses in total (50, 25, 37, 43, 40).

Implement `binary_search`, which takes in a `direction` function (as returned by `make_direction`) and returns the number of guesses it takes for binary search to find the secret. `binary_search` should also print out the sequence of guesses.

```
def binary_search(low, high, direction):
    """Guesses the secret number, as specified by direction, using binary
    search; returns the number of guesses made.

    >>> count1 = binary_search(0, 100, make_direction(50))
    50
    >>> count1
    1

    >>> count2 = binary_search(0, 100, make_direction(40))
    50
    25
    37
    43
    40
    >>> count2
    5
    """
    guess = (low + high) // 2     # midpoint

    ------------------------------------

    sign = ______________________________

    if _________________________________:

        return 1

    elif sign < 0:

        return ______________________________________________

    else:

        return ______________________________________________
```

(e) (1 pt) Again, let $n = \texttt{high} - \texttt{low}$. Circle the order of growth that best describes the runtime of binary search. Choose the tightest bound.

$O(1)$ $O(\log n)$ $O(n)$ $O(n^2)$ $O(2^n)$

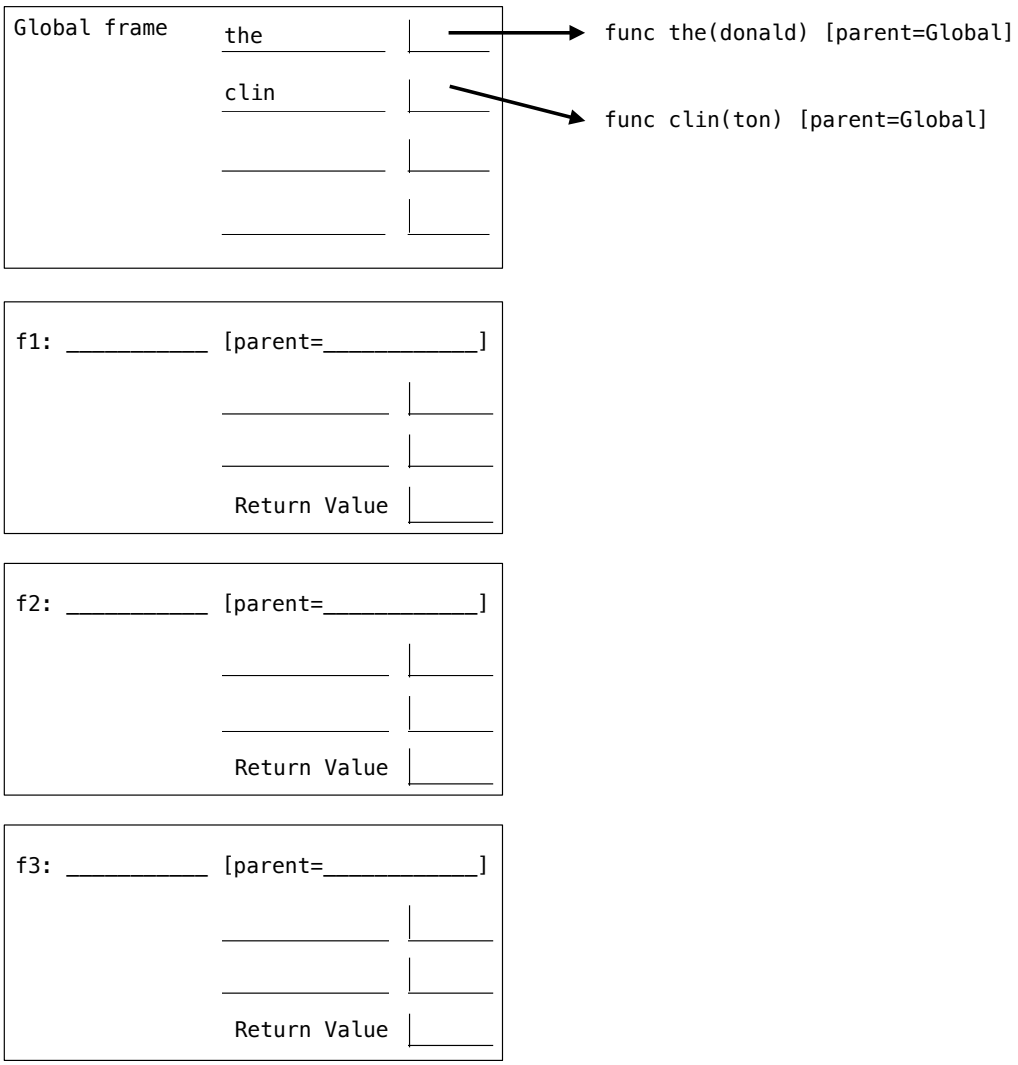
2. (12 points) Environmental Policy

(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  def the(donald):
2      return donald + 5
3
4  def clin(ton):
5      def the(race):
6          return donald + 6
7      def ton(ga):
8          donald = ga-1
9          return the(4)-3
10     return ton
11
12 donald, duck = 2, clin(the)
13 duck = duck(8)
```

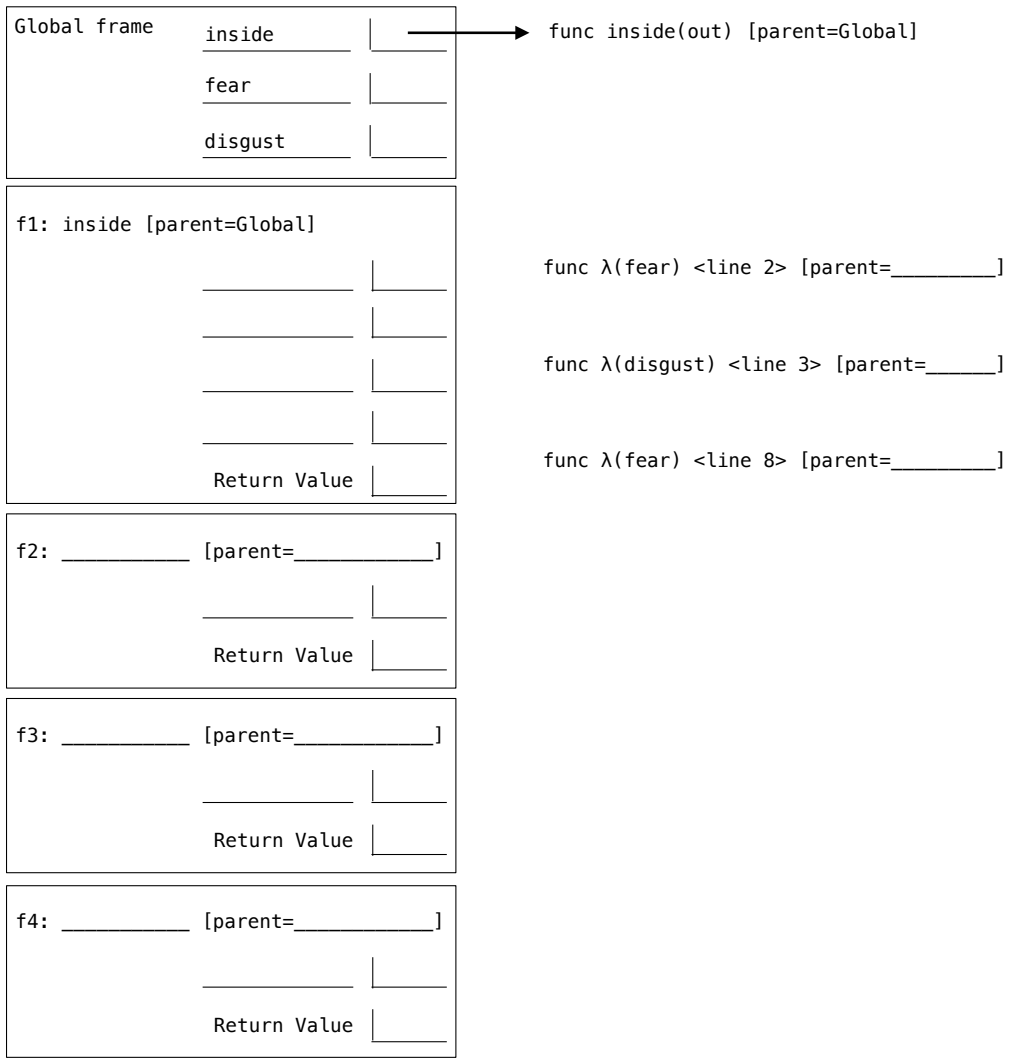


(b) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* The `<line ...>` annotation in a lambda value gives the line in the Python source of a lambda expression.

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Add all missing parents of function values.
- Show the return value for each local frame.

```
1 def inside(out):
2     anger = lambda fear: fear(disgust)
3     fear = lambda disgust: anger(out)
4     disgust = 3
5     fear(5)
6
7 fear, disgust = 2, 4
8 inside(lambda fear: fear + disgust)
```



4. (6 points) Zombies!

****IMPORTANT**** In this question, assume that all of `f`, `g`, and `h` are functions that take **one** non-negative integer argument and return a non-negative integer. You *do not* need to consider negative or fractional numbers.

(a) (4 pt) Implement the higher-order function `decompose1`, which takes two functions `f` and `h` as arguments. It returns a function `g` that relates `f` to `h` in the following way: For any non-negative integer `x`, `h(x)` equals `f(g(x))`. Assume that `decompose1` will be called only on arguments for which such a function `g` exists. Furthermore, assume that there is no recursion depth limit in Python.

```
def decompose1(f, h):
    """Return g such that h(x) equals f(g(x)) for any non-negative integer x.

    >>> add_one = lambda x: x + 1
    >>> square_then_add_one = lambda x: x * x + 1
    >>> g = decompose1(add_one, square_then_add_one)
    >>> g(5)
    25
    >>> g(10)
    100
    """


    def g(x):


        def r(y):


            if ____________________________________________________________:


                return ____________________________________________________


            else:


                return ____________________________________________________


        return r(0)


    ____________________________________________________________________
```

(b) (2 pt) Write a number in the blank so that the final expression below evaluates to 2015. Assume `decompose1` is implemented correctly. The `make_adder` and `compose1` functions appear on the left column of page 2 of your study guide.

```
e, square = make_adder(1), lambda x: x*x

decompose1(e, compose1(square, e))(3) + ______________________________________
```

1. (12 points) Evaluate This!

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "`Error`". If an expression yields (or prints) a function, write "`<Function>`". No answer requires more than 3 lines. (It's possible that all of them require even fewer.) The first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`, plus all values passed to `print`.

(a) (10 pt) Assume that `python3` has executed the statements on the left:

```
y = 7

def b(x):
  return lambda y: x(y)

def c(x):
  return 3

w = b(c)

def c(x):
  return x
```

Expression	Interactive Output
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`(3 and abs)(-1)`	
`print(3) or 1/0`	
`print`	
`([1, 2, 3] if y // (y+1) else [4, 5])[1]`	
`w(5)`	

(b) (2 pt) Assume that `python3` has executed the following statements:

```
def d(S):
    def f(k):
        return k < len(S) and (S[k] == S[0] or f(k+1))
    if len(S) == 0:
        return False
    else:
        return f(1) or d(S[1:])
```

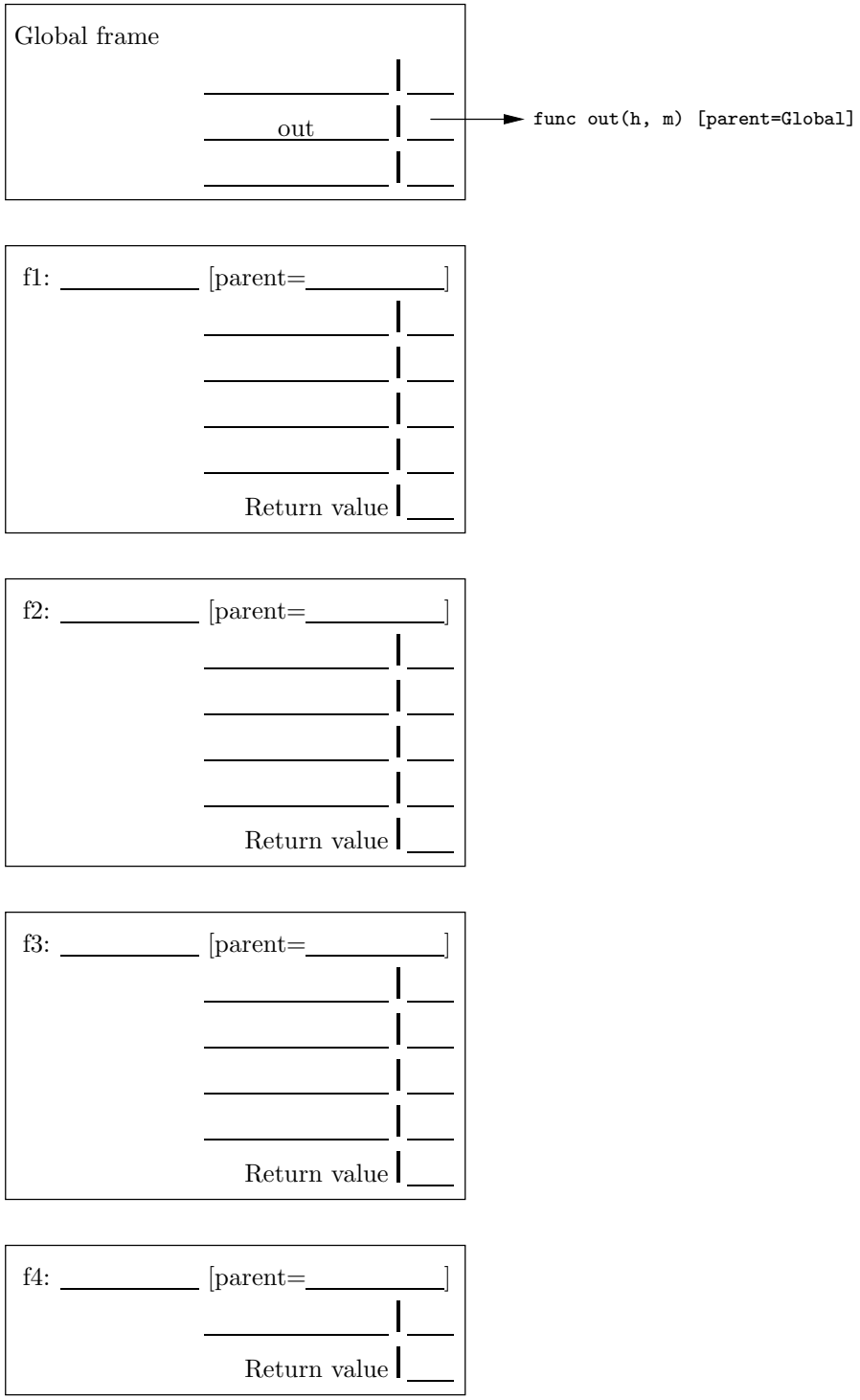
Expression	Interactive Output
`print(d([1, 2, 3]), d([0, 1, 2, 1, 0]))`	

2. (12 points) Environmental Policy

(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  y = 3
2  def out(h, m):
3      y = 5 * m
4      def inner():
5          return y
6      if m == 0:
7          return h
8      else:
9          return out(inner, m-1)
10 v = out(None, 1)()
```

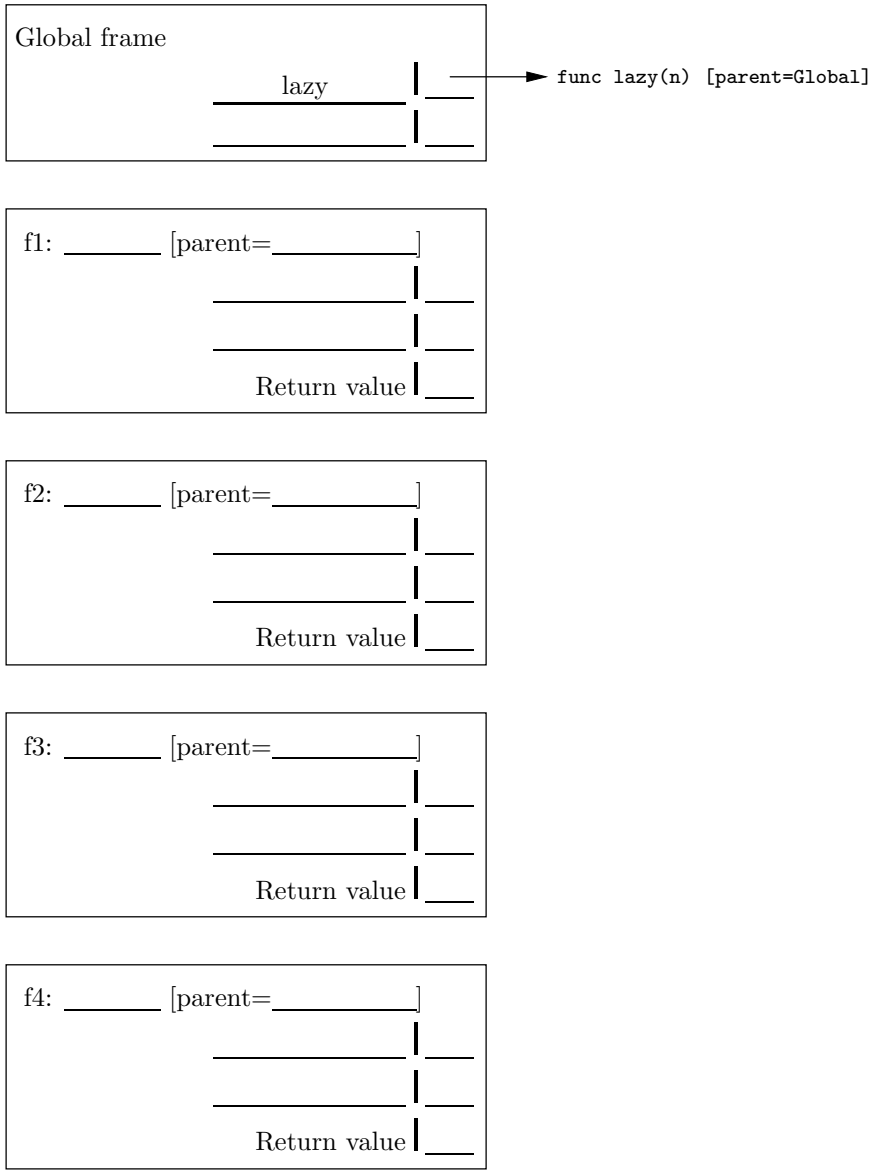


(b) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* The `<line ...>` annotation in a lambda value gives the line in the Python source of a lambda expression.

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Add all missing parents of function values.
- Show the return value for each local frame.

```
1 def lazy(n):
2     return lambda k: (n if k==0 else lazy(n+1))
3 v = lazy(4)(1)(0)
```



6. (8 points) Amazing

In lecture, we did some problems involving finding one's way through a maze, representing the maze as a predicate (boolean function)—the parameter `blocked`. The idea was that `blocked(x, y)` iff the grid square at row `y` and column `x` was blocked. Let's consider a different representation.

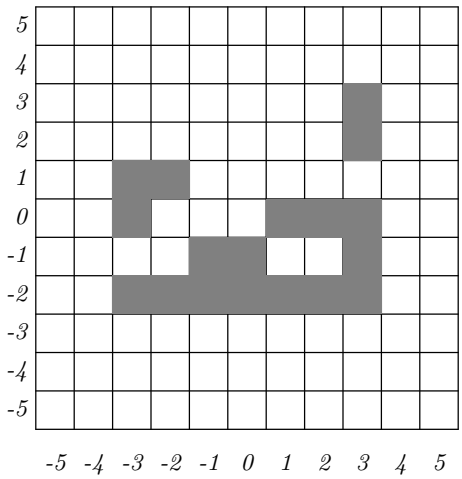
Again, mazes will be represented by functions, but with a different specification. A maze function, say `M`, in this formulation describes the state of the maze from the point of view of a maze runner. `M` accepts a single argument, `direction`, which describes the action the runner tries to take: either the string "south" or "west", indicating that the runner tries to take one step south or one step west, respectively. `M` then returns either:

- Another maze function that represents the state of the runner after taking the indicated step. It again accepts a direction and returns one of these three things;
- The string "exit", signifying that the move causes the runner to successfully reach the exit; or
- The string "dead end", which indicates that the runner was unable to take the requested action because of being blocked by a wall.

(a) (4 pt) Define a function `pred_maze` that returns a maze function (as described above).
If `M = pred_maze`(x, y, P, e), then `M` represents a runner in a maze in which

- The runner is at coordinates (x, y).
- Squares (a, b) for which $a \leq e$ are unblocked and are exit squares.
- A square at (a, b) is otherwise open iff $P(a, b)$.

For example, suppose that `Q` is a function such that `Q(a,b)` is true on all the white squares in the following grid (where, just to be different, the lower-left corner represents position (-5, -5)):



Then we'd have

```
>>> M0 = pred_maze(0, 1, Q, -4)
>>> M1 = M0('west')
>>> M1
<function ...>
>>> M1('west')
'dead end'
>>> M1('south')
<function ...>
>>> M2 = pred_maze(-3, -1, Q, -4)
>>> M2('west')
'exit'
```

Fill in your solution on next page.

```
def pred_maze(x0, y0, open, exit):
    """Return a maze in which the runner is at (X0, Y0), every square
    (a, b) where a <= EXIT is an exit, and otherwise a square (a, b)
    is open iff OPEN(a, b).  It is assumed that (X0, Y0) is open."""

    def maze(____________________________):

        x, y = (x0, y0 - 1) if dir == "south" else (x0 - 1, y0)


        if ______________________________________________:


            return ______________________________________


        elif ____________________________________________:


            return ______________________________________

        else:

            return ______________________________________


    return maze
```

(b) (4 pt) Define a function `path_out` that takes a maze and returns a string describing a path to an exit, or returns `None` iff there is no such path. A call `path_out(M)` will return a string such as

```
'south west south west west'
```

(as in the doctest below) to indicate that the necessary path takes one step south, then a step west, then south, and then two steps west. (Don't worry about extra spaces in the returned string.) When multiple paths exist, return any one. Using the previous value of function `Q`, for example, we'd see:

```
>>> M = pred_maze(-1, 1, Q, -4)
>>> path_out(M)
'south west south west west'
>>> M = pred_maze(2, -1, Q, -4)
>>> path_out(M)   # Returns None
```

FYI: The '`+`' operator on strings is string concatenation ('`+=`') also works.)

```
def path_out(M):
    """Given a maze function M in which the runner is not yet out
    of the maze, returns a string of the form "D1 D2 ...", where each
    Di is either south or west, indicating a path from M to an exit,
    or None iff there is no such path."""

    for dir in ["south", "west"]:

        next = _____________________________________________

        if _____________________________________________:

            return _____________________________________

        elif _____________________________________________:

            rest_of_path = _____________________________

            if _____________________________________________:

                return _____________________________________

    return None
```