CS 61A Structure and Interpretation of Computer Programs

Summer 2015

MIDTERM 1 SOLUTIONS

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A midterm 1 study guide.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
BearFacts email (`_@berkeley.edu`)	
TA	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

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)`	6 7
`print(print(1), vision(2))`	1 2 None 3
`hawkeye(hammer, 3)`	Error
`hawkeye(capt, 3)`	9
`hammer(lambda ultron: ultron, -1)`	0
`hammer(vision, avengers)`	6 6 -6

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.

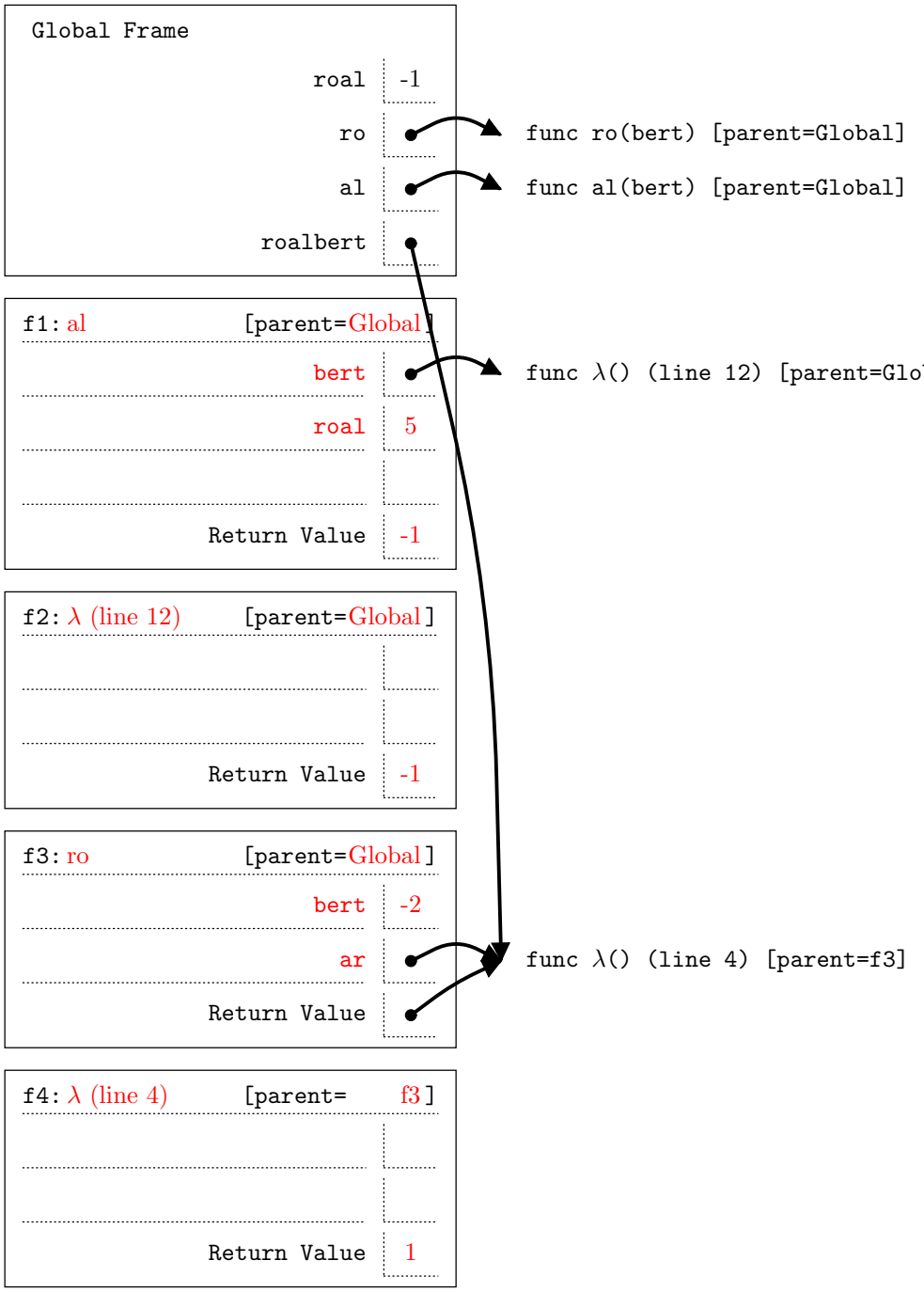
```
1  def joy(a, b):
2      x = 42
3      y = sadness(a + b, x)
4      return x - y
5
6  def sadness(a, b):
7      a, b = a + x, a - y
8      return a // b
9
10 x, y = 1, 2
11 joy(x, y)
```

(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):
    """
    Todo
    """
    def direction(guess):

        if secret > guess:

            return 1

        elif secret < guess:

            return -1

        else:

            return 0

    return direction
```

(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):
    """Guesses the secret number, as specified by direction, in a naive way;
    returns the number of guess 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 = direction(guess)

    while sign != 0:

        guess = guess + sign

        count = count + 1

        sign = direction(guess)

        print(guess)

    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

    print(guess)

    sign = direction(guess)

    if sign == 0:

        return 1

    elif sign < 0:

        return 1 + binary_search(low, guess, direction)

    else:

        return 1 + binary_search(guess, high, direction)
```

(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.

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

4. (12 points) "A highly intelligent animal."

(a) (3 pt) Implement the `subset_sum(target, lst)` function: given a target integer `target` and a list of integers `lst`, return `True` if it is possible to add together any of the integers in `lst` to get the `target`. For example, `subset_sum(10, [-1, 5, 4, 6])` will return `True` (either `-1 + 5 + 6 = 10` or `4 + 6 = 10`), while `subset_sum(4, [5, -2, 12])` will return `False`.

Note: an integer may appear multiple times in `lst` (for example, `[2, 4, 2, 3]`). An integer in `lst` can only be used once (for example, `subset_sum(4, [2, 3])` is `False` because we can only use the `2` once).

```
def subset_sum(target, lst):
    """Returns True if it is possible to add some of the integers in lst
    to get target.

    >>> subset_sum(10, [-1, 5, 4, 6])
    True
    >>> subset_sum(4, [5, -2, 12])
    False
    >>> subset_sum(-3, [5, -2, 2, -2, 1])
    True
    >>> subset_sum(0, [-1, -3, 15])     # Sum up none of the numbers to get 0
    True
    """
    if target == 0:

        return True

    elif not lst:

        return False

    else:

        a = subset_sum(target - lst[0], lst[1:])

        b = subset_sum(target, lst[1:])

        return a or b
```

(b) (5 pt) Implement `intersection(lst_of_lsts)`, which takes a list of lists and returns a list of elements that appear in all the lists in `lst_of_lsts`. If no number appears in all of the lists, return the empty list. You may assume that `lst_of_lsts` contains at least one list.

Hint: recall that you can check if an element is in a list with the `in` operator:

```
>>> x = [1, 2, 3, 4]
>>> 4 in x
True
>>> 5 in x
False
>>>
```

```
def intersection(lst_of_lsts):
    """Returns a list of elements that appear in every list in
    lst_of_lsts.

    >>> lsts1 = [[1, 2, 3], [1, 3, 5]]
    >>> intersection(lsts1)
    [1, 3]
    >>> lsts2 = [[1, 4, 2, 6], [7, 2, 4], [4, 4]]
    >>> intersection(lsts2)
    [4]
    >>> lsts3 = [[1, 2, 3], [4, 5], [7, 8, 9, 10]]
    >>> intersection(lsts3)          # No number appears in all lists
    []
    """

    elements = []

    for elem in lst_of_lsts[0]:

        condition = True

        for lst in lst_of_lsts[1:]:

            if elem not in lst:

                condition = False

        if condition:

            elements = elements + [elem]

    return elements
```

(c) (4 pt) A number `n` contains a *sandwich* if a digit in `n` is surrounded by two identical digits. For example, the number `242` contains a sandwich because `4` is surrounded by `2` on both sides. `1242` also contains a sandwich, while `12532` does not contain a sandwich.

Implement the `sandwich(n)` function, which takes in a nonnegative integer `n`. It returns `True` if `n` contains a sandwich and `False` otherwise. If `n` has fewer than three digits, it cannot contain a sandwich.

```
def sandwich(n):
    """Returns True if n contains a sandwich and False otherwise

    >>> sandwich(416263)    # 626
    True
    >>> sandwich(5050)      # 505 or 050
    True
    >>> sandwich(4441)      # 444
    True
    >>> sandwich(1231)
    False
    >>> sandwich(55)
    False
    >>> sandwich(4456)
    False
    """
    a, b = (n // 10) % 10, n % 10

    n = n // 100

    while n > 0:

        if n % 10 == b:

            return True

        else:

            a, b = n % 10, a

            n = n // 10

    return False
```

5. (0 points) The real guessing game

In the box below, write a positive integer. The student who writes the lowest unique integer will receive one extra credit point. In other words, write the smallest positive integer that you think no one else will write.