CS 61A Structure and Interpretation of Computer Programs

Spring 2014

Test 1 Solutions

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is open book and open notes. You may not use a computer, calculator, or anything responsive.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.
- Since some students will be taking the exam later, PLEASE DO NOT DISCUSS IT before discussion sections.

Last name	
First name	
SID	
Login	
TA & section time	
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)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Total
/12	/15	/18	/	/5	/50

1. (12 points) What would Python Do?

For each of the following call expressions, write the value to which it evaluates *and* what would be output by the interactive Python interpreter.

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

```
def a(f):
    def i(x):
        return f(x)(x)
    return i

def go(z):
    print(z)
    if z < 10:
        return z
    elif z < 20:
        return lambda c: c - 7
    else:
        return print(z + 10)

bears = a(go)
```

Expression	Evaluates to	Interactive Output
abs(-3)	3	3
1/0	Error	Error
go(5)	5	5 5
go(25)	None	25 35
go(15)(10)	3	15 3
bears(5)	Error	5 Error
bears(15)	8	15 8
bears(25)	Error	25 35 Error

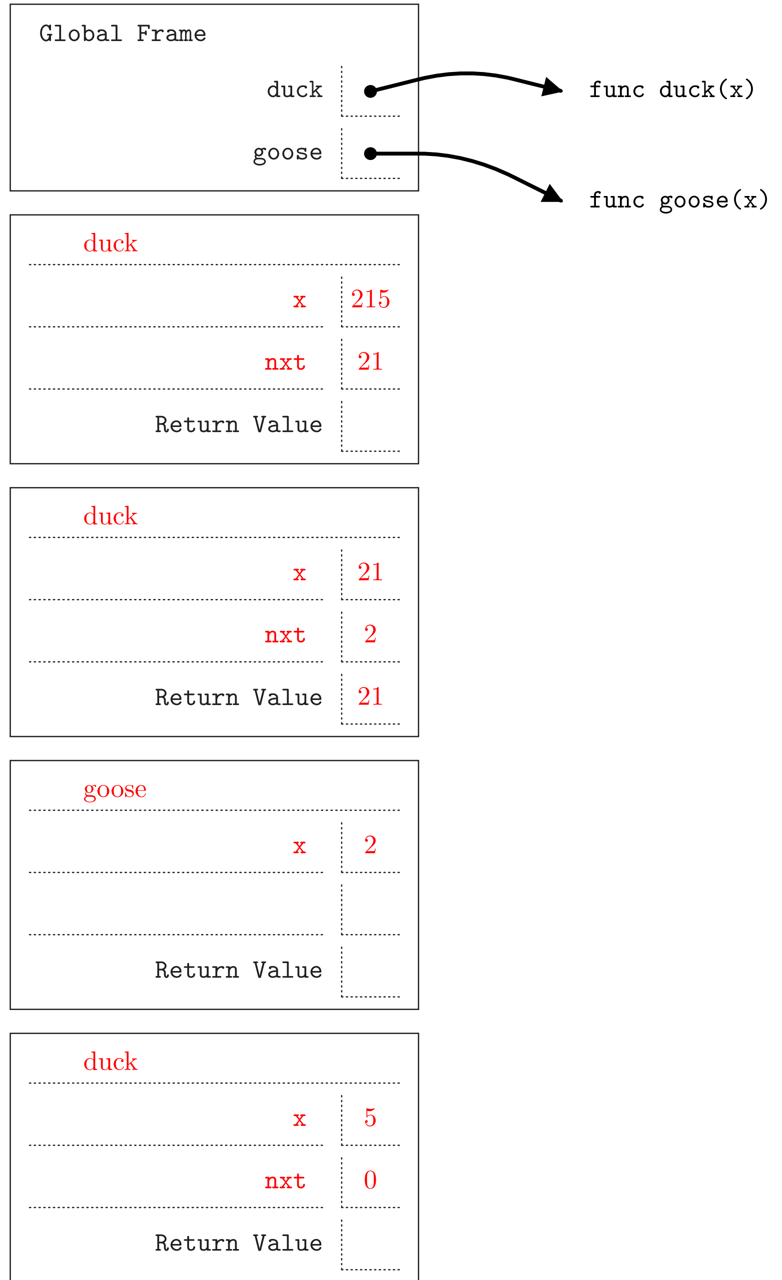
2. (15 points) Environments

For each of the following code segments, fill in the environment diagram that results from executing it 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, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame resulting from a call that has completed when execution stops (leave other return values blank).

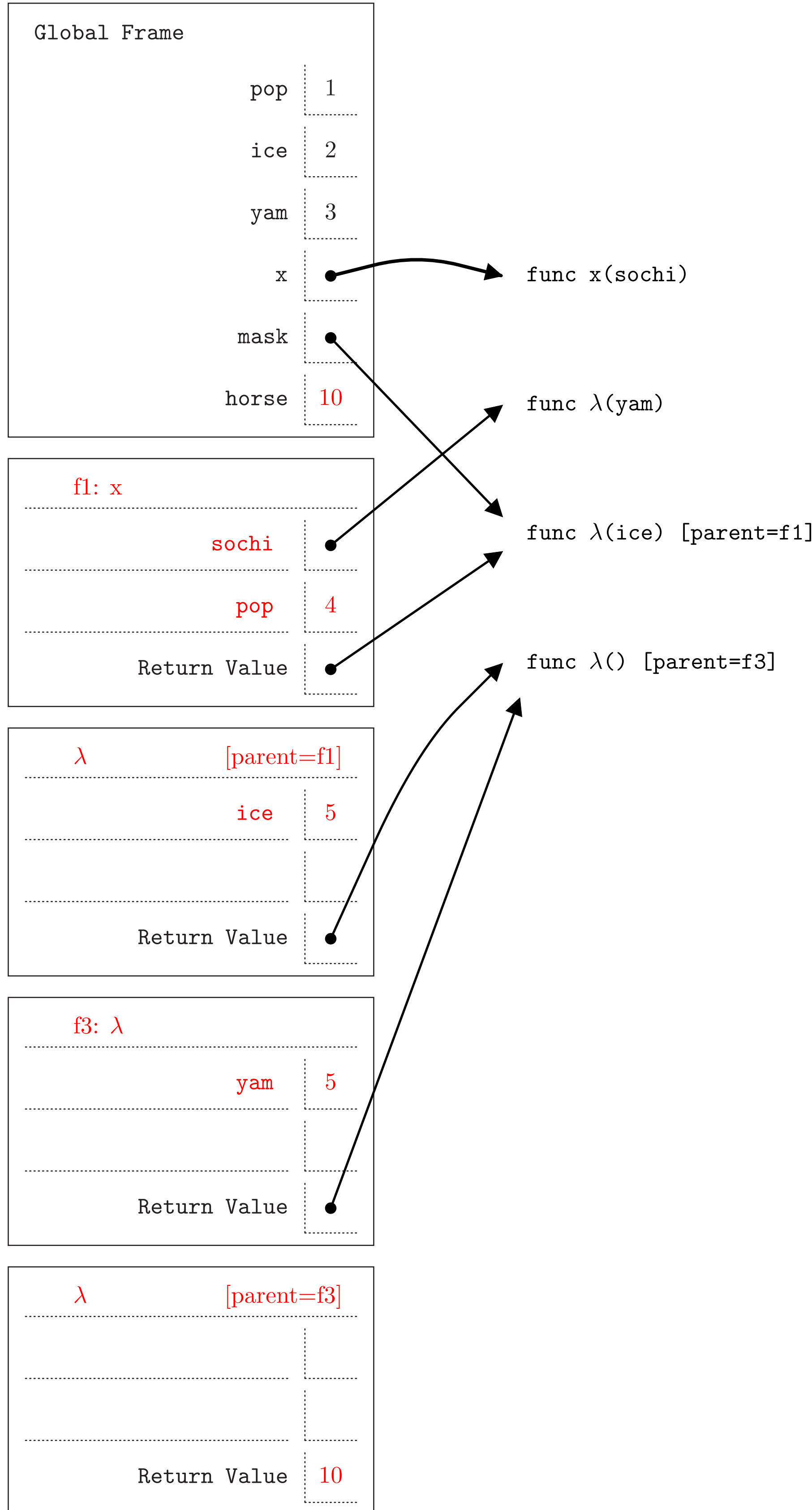
(a) (7 points) Duck, Duck, Goose

```
def duck(x):
    nxt = x // 10
    if x >= 100:
        duck(nxt)
        return goose(nxt // 10)
    elif x >= 10:
        return x
    else:
        return nxt(x)

def goose(x):
    return duck(x + 3)

duck(215)
```



(b) (8 points) Miscellaneous

```
pop, ice, yam = 1, 2, 3
def x(sochi):
    pop = 4
    return lambda ice: sochi(ice)

mask = x(lambda yam:
            lambda: yam*pop*ice)
horse = mask(5)()
```



3. (19 points) Implementations

(a) (4 points) Fill in the definition of `best_prefix_sum` to match its comments. Assume that the function argument, `f`, is pure (has no side effects and always returns the same result for the same input).

```
# tup is for testing purposes only.
def tup(vals):
    """Given a tuple VALS, return a function of one integer
    argument, k, that returns VALS[k].
    >>> h = tup( (3, 1, 4) )
    >>> h(0)
    3
    >>> h(2)
    4
    """
    return lambda k: vals[k]

def best_prefix_sum(f, low, limit):
    """The largest sum obtainable by adding F(LOW)+F(LOW+1)+...+F(u)
    where u<LIMIT. (The best u might be LOW-1, indicating the empty
    prefix, in which case the sum is 0.)  You may assume LOW<=LIMIT.
    >>> best_prefix_sum(tup((1, 2, 3)), 0, 3)
    6
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 3)
    5
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 2)
    2
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 1)
    0
    >>> best_prefix_sum(tup((1, -2, 4)), 0, 3)
    3
    >>> best_prefix_sum(tup((1, -5, 4)), 0, 3)
    1
    >>> best_prefix_sum(tup((-2, -3, 6)), 0, 3)
    1
    >>> best_prefix_sum(tup((-2, -3, 4)), 0, 3)
    0
    """

    if ______________low >= limit______________:

        return ____________0____________

    else:

        return max(__________0, f(low) + best_prefix_sum(f, low+1, limit)__________)
```

(b) **(4 points)** Re-implement `best_prefix_sum` from the previous problem using iteration (no recursive calls). As before, assume that the function argument, `f`, is pure (has no side effects and always returns the same result for the same input).

```
def best_prefix_sum(f, low, limit):
    """The largest sum obtainable by adding F(LOW)+F(LOW+1)+...+F(u)
    where u<LIMIT. (The best u might be LOW-1, indicating the empty
    prefix, in which case the sum is 0.)  You may assume LOW<=LIMIT.
    >>> best_prefix_sum(tup((1, 2, 3)), 0, 3)
    6
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 3)
    5
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 2)
    2
    >>> best_prefix_sum(tup((1, 2, 3)), 1, 1)
    0
    >>> best_prefix_sum(tup((1, -2, 4)), 0, 3)
    3
    >>> best_prefix_sum(tup((1, -5, 4)), 0, 3)
    1
    >>> best_prefix_sum(tup((-2, -3, 6)), 0, 3)
    1
    >>> best_prefix_sum(tup((-2, -3, 4)), 0, 3)
    0
    """
# NOTE: you can put multiple assignments in one blank:
#       a, b, c = f(a), b*2, 5     or
#       a = f(a); b = b*2; c = 5

    k, sum, best_sum = low, 0, 0

    while k < limit:

        sum, best_sum, k = sum+f(k), max(sum+f(k), best_sum), k+1

    return best_sum
```

(c) **(3 points)** Complete the following expression so that it evaluates to 5.

```
(lambda ________ : lambda func: lambda : func(1) )()(lambda x: 5)()
```

(d) (4 points) Complete `chooser` to fulfill its comment.

```
def chooser(f, g, x, y):
    """Given that F and G are increasing, one-argument integer functions,
    returns true iff some combination of applying F and G to X yields
    Y. Thus, chooser(f, g, x, y) is true if. for example, f(f(g(f(g(x)))))==y.
    >>> f, g = lambda x: x + 2, lambda x: x*2
    >>> chooser(f, g, 2, 3)
    False
    >>> chooser(f, g, 1, 8)  # (1 + 2) * 2 + 2
    True
    """

    if ________________x>y________________:
        return False

    elif ________________x == y________________:
        return True
    else:

        return ________chooser(f, g, f(x), y) or chooser(f, g, g(x), y)________
```

(e) (3 points) The boolean values `True` and `False` are actually instances of a kind of integer that Python treats *as if* they represented our logical notions of true and false. Suppose instead we use *functions* to represent boolean values. That is, we'll represent "true" with a value `true` (lower case) that is a function, and likewise "false" with a function `false`:

```
true = lambda a, b: a
false = lambda a, b: b
```

The function `py_truth` is supposed to convert these values into the normal Python `True` and `False`. Fill in the blanks so as to make the functions below conform to their comments. Do not use any conditional expressions or statements (**if**, **and**, **or**, **not**, **while**), and do not call `py_truth`.

```
def py_truth(p):
    """The Python boolean value represented by P, a functional truth value.
    >>> py_truth(true)
    True
    >>> py_truth(false)
    False
    """
    return p(True, False)

def functional_and(p1, p2):
    """
    >>> py_truth(functional_and(true, true))
    True
    >>> py_truth(functional_and(false, true))
    False
    >>> py_truth(functional_and(true, false))
    False
    >>> py_truth(functional_and(false, false))
    False
    """
    return ________________p1(p2, false)________________
```

4. (1 point) Sum of Human Knowledge.

Whom do the following quotations have in common? Robert Heinlein

"TANSTAAFL"

"Anyone who cannot cope with mathematics is not fully human. At best he is a tolerable subhuman who has learned to wear shoes, bathe, and not make messes in the house."

"The Bird is Cruel!"

5. (5 points) Iterative Solver

A number x is called a *fixed point* of f if $f(x) = x$. For some functions, we can locate a fixed point by beginning with an initial guess and applying f repeatedly: $x, f(x), f(f(x)), \ldots$. Write a function `fixed_point` that takes in a function, f, and a starting point and returns its fixed point, using this procedure until the difference between $f(x)$ and x is sufficiently small (for x the current guess). To guard against an infinite loop, `fixed_point` also takes in a maximum repetition count, `limit`, and simply returns the current guess if f has been applied `limit` times. Use the `iter_solve2` function from class and any built-in functions and lambda expressions you need.

```
def iter_solve2(guess, done, update, other_stuff):
    """Return the result of repeatedly applying UPDATE to GUESS
    and OTHER_STUFF, until DONE yields a true value when applied to
    GUESS and OTHER_STUFF.  UPDATE returns an updated guess and other stuff."""
    while not done(guess, other_stuff):
        guess, other_stuff = update(guess, other_stuff)
    return guess

def fixed_point(f, start, tolerance, limit=100):
    """Find the fixed point of F by applying F to itself enough times
    starting at START such that F(x) is no more than TOLERANCE from x.
    (round(y,N) rounds y to N decimal places.  Trailing 0s are not printed).
    >>> f = lambda x: -(x * x + x) / 6 + 3
    >>> round(fixed_point(f, 1, 1e-12, 200),  8)
    2.0
    >>> round(fixed_point(lambda x: 1 + 1 / x, 1, 1e-12), 8)   # Golden ratio
    1.61803399
    >>> fixed_point(lambda x: x+1, 1, 1e-12, 10)  # No fixed point
    11
    """

    done = ______lambda x, k: k >= limit or abs(x-f(x)) <= tolerance______

    update = ______lambda x, k: (f(x), k+1)______

    return iter_solve2(______start______, done, update, ______0______)
```