CS 61A Structure and Interpretation of Computer Programs

Fall 2011 Midterm Exam 1

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except a one-page crib sheet of your own creation and the official 61A Midterm 1 Study Guide.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation. All short answer sections can be successfully answered in a few sentences at most.

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	Total
/10	/16	/14	/10	/50

THIS PAGE INTENTIONALLY LEFT BLANK

1. (10 points) Call Expressions

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

```
from operator import add, mul

def square(x):
    return mul(x, x)

def curry(f):
    def g(x):
        return lambda y: f(x, y)
    return g
```

For each of the following call expressions, write the value *to which it evaluates* in the current environment (which may differ from what is printed). If evaluating the call expression causes an error, write "error" *and* the reason for the error.

(a) (2 pt) `add(4, square(print(2)))`

(b) (2 pt) `add(4, square(square(2)))`

(c) (2 pt) `print(4, square(square(2)))`

(d) (2 pt) `curry(add)(square(2))(4)`

(e) (2 pt) `add(curry(square)(2), 4)`

2. (16 points) Defining Functions

(a) (5 pt) A function `streak` returns the greatest number of 1's returned consecutively in `n` calls to `coin`. Its second argument `coin` is a random, non-pure function that returns 1 sometimes and 0 other times. For instance, `streak` would return 3, if 10 calls to `coin` returned these 10 values, in order:

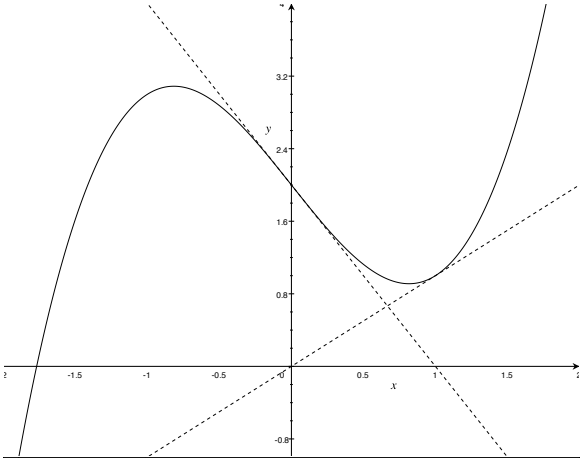
0, 1, 1, 0, 1, 1, 1, 0, 0, 1

Your partner hands you the following implementation of `streak` moments before this question is due. You don't have time to rewrite it; you can only indent lines, dedent (un-indent) lines, and delete things. Mark changes to the code below so that it correctly implements `streak`, using the following symbols:

- Place a **right arrow** (→) before a line to indent it by an additional 4 spaces.
- Place a **left arrow** (←) before a line to dedent it by 4 spaces.
- **Circle** a part of the code to delete it.

```
def streak(n, coin):

    """Return the greatest number of 1's returned consecutively

    in n calls to coin.

    coin -- A function that takes zero arguments and returns 0 or 1.

    n -- A positive integer

    """

    n, k = 0, 0, 1

    total = 0  # consecutive 1's since a 0

    greatest = 0 # The greatest number of consecutive 1's

    while k <= n and coin() == 1:

        if coin == 0 or coin(n) == 0 or coin() == 0:

            greatest, total = max(greatest, total), total, 0

        else:

            total = total + 1

            k = k + 1

            n = n + 1

        return greatest
```

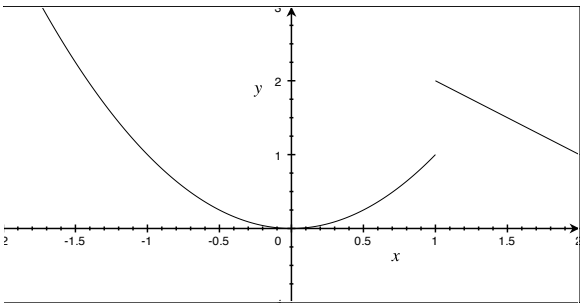
(b) (5 pt) In Newton's method, the function `iter_improve` may cycle between two different guesses. For instance, consider finding the zero of $f(x) = x^3 - 2x + 2$ (graphed below) using exact derivatives. A starting guess of 1 updates to 0, which updates to 1 again. The cyclic pattern $1, 0, 1$ is called a length-two cycle, because it contains only two distinct elements before repeating.



Write a new version of `iter_improve` that returns `False` as soon as a length-two cycle of guesses occurs, but otherwise behaves the same as the implementation from class (and printed on your study guide).

```
def iter_improve(update, done, guess=1, max_updates=1000):
```

(c) (4 pt) The figure below depicts an example piecewise function, $f(x) = \begin{cases} x^2 & x < 1 \\ 3 - x & x \geq 1 \end{cases}$



Define a Python function called `piecewise` that takes three arguments: two functions `lower_func` and `upper_func`, and a number `cutoff`. `lower_func` and `upper_func` each take a single number argument and return a single number value.

`piecewise` returns a new function that takes a single number argument `x` and returns `lower_func(x)` if `x` is less than `cutoff`, or `upper_func(x)` if `x` is greater than or equal to `cutoff`. For example:

```
>>> f = piecewise(lambda x: x*x, lambda x: 3-x, 1)
>>> f(0.5)
0.25
>>> f(5)
-2
```

```
def piecewise(lower_func, upper_func, cutoff):
```

(d) (2 pt) Assume that you have started Python 3 and executed the following statement:

```
def self_apply(f):
    return f(f)
```

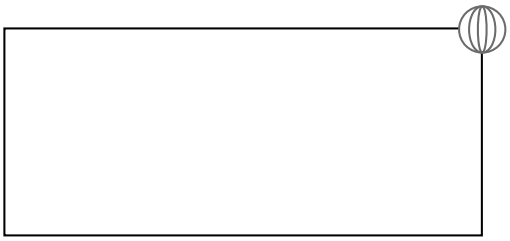
Write an expression in the blank operand position below, so that this call to `self_apply` evaluates to 5:

```
self_apply(                                                    )
```

3. (14 points) Environments

(a) (6 pt) Fill in this environment diagram and expression tree that result from executing the Python code in the lower right of the page. A complete answer will:

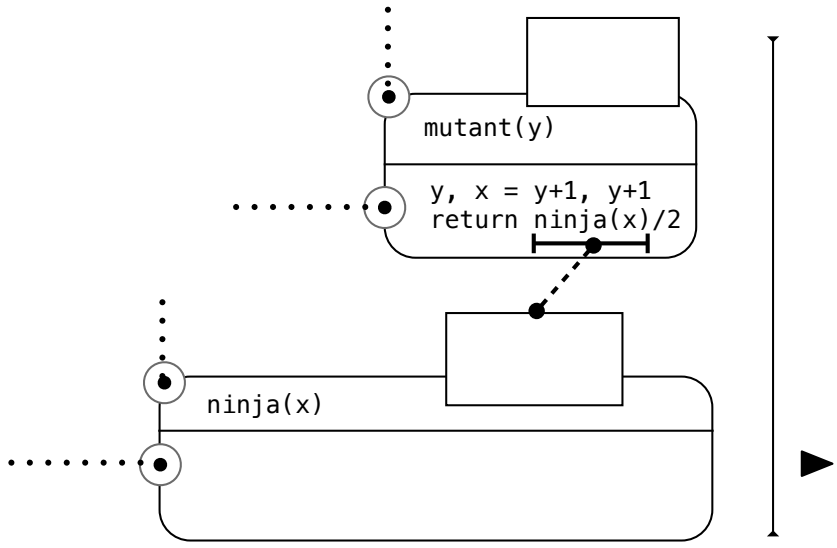
- Complete all missing arrows.
- Add all local frames created by applying user-defined functions.
- Add all missing names and values.
- Add all missing statements, expressions, and return values in the expression tree.



```
mutant(y):
    y, x = y+1, y+2
    return ninja(y)/2
```

```
ninja(x):
    return x + 2
```

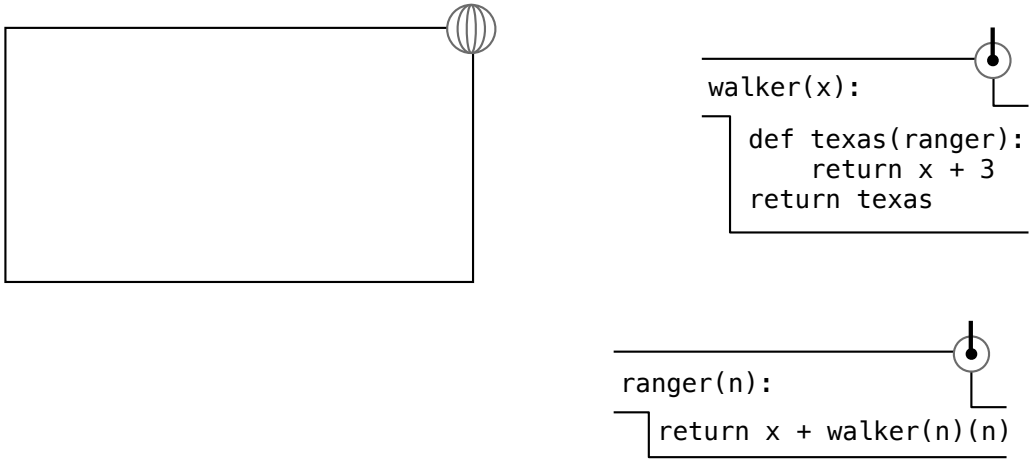
```
turtle(x)
    return mul(x, y) + 2
```




```
def mutant(y):
    y, x = y+1, y+2
    return ninja(y)/2

def ninja(x):
    return x + 2

def turtle(x):
    return mul(x, y) + 2

y, ninja = 5, turtle

mutant(y)
```

(b) (6 pt) Fill in this diagram of environments and values that result from executing the Python code in the box below. You *do not* need to draw an expression tree. A complete answer will:

- Complete all missing arrows.
- Add all local frames created by applying user-defined functions.
- Add all missing names and values.




```
x = 10
def walker(x):
    def texas(ranger):
        return x + 3
    return texas

def ranger(n):
    return x + walker(n)(n)

texas = ranger
ranger(2)
```

(c) (2 pt) What is the value of the nested expression `texas(ranger(2))`, evaluated in the current environment that results from executing the code in the previous problem? If evaluating this expression causes an error, write "error" and the reason for the error.

4. (10 points) Data Abstraction

An *amount* is a collection of nickels (5 cents) and pennies (1 cent), with the following behavior condition:

If an amount `a` is constructed with n nickels and p pennies, then $5 \cdot$ `nickels(a)` + `pennies(a)` $= 5 \cdot n + p$.

Consider the following implementation of an abstract data type for amounts.

```
def make_amount(n, p):
    return (n, p)

def nickels(a):
    return a[0]

def pennies(a):
    return a[1]
```

(a) (2 pt) Circle all abstraction barrier violations for this abstract data type in the following implementation of `add_amounts`, if there are any.

```
def add_amounts(a1, a2):

    n1 = nickels(a1)

    n2 = nickels(a2)

    n = n1 + n2

    p = a1[1] + a2[1]

    return (n, p)
```

(b) (4 pt) An amount is minimal if it has no more than 4 pennies. Redefine the constructor so that all amounts are minimal, but the behavior condition of the type still holds.

```
def make_amount(n, p):
```

(c) (4 pt) Suppose we redefine `make_amount` functionally, as follows:

```
def make_amount(n, p):
    def dispatch(message):
        if message == 0:
            return n
        elif message == 1:
            return p
    return dispatch
```

Redefine the selectors to implement a valid abstract data type for amounts using this constructor, in which *all amounts are minimal*. That is, `pennies` should never return a number greater than 4.

```
def nickels(a):
```

```
def pennies(a):
```