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.

<table>
<thead>
<tr>
<th>Last Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td></td>
</tr>
<tr>
<td>SID</td>
<td></td>
</tr>
<tr>
<td>Login</td>
<td></td>
</tr>
<tr>
<td>TA &amp; Section Time</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your right</td>
<td></td>
</tr>
</tbody>
</table>

*All the work on this exam is my own. (please sign)*

<table>
<thead>
<tr>
<th></th>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/10</td>
<td>/16</td>
<td>/14</td>
<td>/10</td>
<td>/50</td>
</tr>
</tbody>
</table>
1. (10 points) Call Expressions

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

```python
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)))

```
Error: print(2) evaluates to None. Cannot multiply None and None in the body of square.
```

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

```
20
```

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

```
None
```

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

```
8
```

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

```
Error: Cannot add a function and an int.
```
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, 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.

```python
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 max(greatest, total)
```

This circle is optional
(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).

```python
def iter_improve(update, done, guess=1, max_updates=1000):
    k = 0
    previous_guess = None
    while not done(guess) and k < max_updates:
        next_guess = update(guess)
        if previous_guess == next_guess:
            return False
        previous_guess, guess = guess, next_guess
        k = k + 1
    return guess
```
(c) (4 pt) The figure below depicts an example piecewise function, \( f(x) = \begin{cases} x^2 & x < 1 \\ 3 - x & x \geq 1 \end{cases} \)

![Graph of the piecewise function](image)

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:

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

```python
def piecewise(lower_func, upper_func, cutoff):
    def f(x):
        if x < cutoff:
            return lower_func(x)
        else:
            return upper_func(x)
    return f
```

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

```python
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:

```python
self_apply( lambda f: 5 )
```
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.

```python
from operator import mul
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, x = 5, 6
mutant(y)
```

Environment Diagram and Expression Tree

- Mutant function: `mutant(y)`
- Ninja function: `ninja(x)`
- Turtle function: `turtle(x)`
- Variables: `y`, `x`

---

```
from operator import mul
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, x = 5, 6
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.

28
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 $5 \cdot \text{nickels}(a) + \text{pennies}(a) = 5 \cdot n + p$.

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

```python
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.

```python
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.

```python
def make_amount(n, p):
    return (n + p // 5, p % 5)
```
(c) (4 pt) Suppose we redefine make_amount functionally, as follows:

```python
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.

```python
def nickels(a):
    return a(0) + a(1) // 5

def pennies(a):
    return a(1) % 5
```