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 61A midterm 1 study guide attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a brief explanation.

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

<table>
<thead>
<tr>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Q. 5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/12</td>
<td>/12</td>
<td>/6</td>
<td>/12</td>
<td>/8</td>
<td>/50</td>
</tr>
</tbody>
</table>
1. (12 points) Call Me Maybe

For each of the following call expressions, write the value to which it evaluates and what would be output by the interactive Python interpreter. The first two rows have been provided as examples.

- In the **Evaluates to** column, write the value to which the expression evaluates. If it evaluates to a function value, write **FUNCTION**. If evaluation causes an error, write **ERROR**.
- In the column labeled **Interactive Output**, write all output that would be displayed during an interactive session, after entering each call expression. This output may have multiple lines. Whenever the interpreter would report an error, write **ERROR**. You should include any lines displayed before an error.

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

```python
from operator import add, mul

def mulled(x, y):
    return mul(x, add(y, x))

def fauxpose(f, g):
    print('maybe ')
    def h(x, y):
        f(x, y)
        return g(x, y)
    return h
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulled(5, 5)</code></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td><code>1/0</code></td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td><code>mulled(4, 1)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add(mulled(3, 2), print(4))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>print(3, print(5, print(1)))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fauxpose(add, mul)(3, 2)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fauxpose(mul, print)(4, 1)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fauxpose(fauxpose, mulled)(2, 5)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. (12 points) We Are All Environmentalists

(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 need only 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, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```python
def maybe(x):
    next = lambda y: y - 1
    x += 3
    def year(z):
        return next(z + x) * 2
    return year

x = maybe(2)(4)
```

Global frame

```python
def maybe(x):
    next = lambda y: y - 1
    x += 3
    def year(z):
        return next(z + x) * 2
    return year

x = maybe(2)(4)
```

Return Value
(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 need only 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, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```python
def snow(snow, x):
    if snow(x, x) == x:
        def x(x):
            return 32
        return x(x)
    else:
        return snow(snow, x)

def flake(x, y):
    return y + x - 1

griffin = snow(flake, 1)
```
3. (6 points) A Higher Order of Protection

Louis Reasoner is making a web application, and he wants to secure it. (Good for him!) One of the ways he wants to secure it is through checking to make sure that the user is an admin when it tries to visit certain confidential pages. So, being a silly programmer, he does the following.

```python
def delete_everything(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    confirmation = do_bad_stuff(request)  # BAD STUFF HAPPENS HERE
    return confirmation

def steal_credit_card_info(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    cc_info = hack_a_shaq(request)  # DO SOME 1337 HAXORING
    return cc_info
```

However, Alyssa P. Hacker comes across this code, and realizes that there is a better way to do this using higher-order functions! She modifies the above as follows.

```python
def delete_everything(request):
    confirmation = do_bad_stuff(request)  # BAD STUFF HAPPENS HERE
    return confirmation
def steal_credit_card_info(request):
    cc_info = hack_a_shaq(request)  # DO SOME 1337 H4X0RING
    return cc_info
```

Help her to complete the code by filling in the function below. The new code should provide the same functionality as the original code. For example, calling `delete_everything(True, my_request)` should have the same effect in both versions of the code.

You may leave lines blank if you do not need them.

```python
def protect_me(fn):
    if not is_admin:
        print('ERROR: not admin')
        return
    return fn
```
4. (12 points)  Da Visors Provide Protection from Puns

An integer \( d \) is a divisor of another integer \( n \) if it evenly divides \( n \), i.e. the remainder is 0 when dividing \( n \) by \( d \). The divisors of \( n \) include 1 and \( n \) itself.

(a) Complete the function below to compute the number of positive divisors of a positive integer. Fill in the blanks. You may leave a line blank if you do not need it.

```python
def num_divisors(n):
    """Computes the number of positive divisors of a positive integer."
    i, count = 1, 0
    while ________________________________________________________________:
        if ________________________________________________________________:
            count = count + 1
    return count
```

(b) Write a function that computes the sum of the positive divisors of a positive integer.

```python
def sum_divisors(n):
    """Computes the sum of the positive divisors of a positive integer."
    >>> sum_divisors(4)  # 1, 2, and 4
    7
    """

(Continued on next page)
(c) A positive integer $n$ is called abundant if the sum of its divisors (except $n$ itself) is strictly larger than $n$. It is called perfect if the sum of its divisors (except $n$ itself) is exactly equal to $n$. Finally, $n$ is deficient if the sum of its divisors (excluding $n$) is strictly less than $n$. Write a function that returns the string 'abundant' if the input $n$ is abundant, 'perfect' if $n$ is perfect, and 'deficient' if $n$ is deficient. You may call sum_divisors and assume that it works correctly.

```python
def describe(n):
    """Returns whether n is abundant, perfect, or deficient."""

    >>> describe(4)  # 1 + 2 < 4
    'deficient'
```

5. **(8 points) Lambda the Free**

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

```python
def muckluck(x):
    square(square(x))

def apply(f, x):
    return f(x)

def apply_many(f, x, n):
    while n > 0:
        x = f(x)
    return x

def wut(f):
    return lambda f: f(x)

def pair(x, y):
    return lambda k: x if k == 0 else y
```

For each of the following call expressions, write the value to which it evaluates. If the value is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square</code></td>
<td></td>
</tr>
<tr>
<td><code>muckluck(2)</code></td>
<td></td>
</tr>
<tr>
<td><code>apply(pair, 3)</code></td>
<td></td>
</tr>
<tr>
<td><code>wut(square)</code></td>
<td></td>
</tr>
<tr>
<td><code>pair(3, 4)(1)</code></td>
<td></td>
</tr>
<tr>
<td><code>pair(apply, square)(0)</code></td>
<td></td>
</tr>
<tr>
<td><code>apply_many(square, 3, 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>apply_many(square, 3, 2)</code></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation rule for call expressions:
1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:
1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Execution rule for def statements:
1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:
Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Execution rule for conditional statements:
Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for and expressions:
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:
Evaluate <exp>;
The value is True if the result is a false value, and False otherwise.

A name is bound to a value
in a frame, there is at most one binding per name.

A two-frame environment

The environment with only the global frame

An environment is a sequence of frames
An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame.

Non-Pure Functions

Higher-order function:
A function that takes a function as an argument value or returns a function as a return value

Nested def statements:
Functions defined within other function bodies are bound to names in the local frame.
```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7

def square(x):
    """Square computes the square by calling mul."
    return x * x

def sum_squares(x, y):
    """Return the sum of squares of x and y."
    return square(x) + square(y)
```

### Facts about `print`
- **Non-pure function**
- Returns None
- Multiple arguments are printed with a space between them

### How to find the square root of 2?
1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: x - f(x) / f'(x)

```python
def find_root(f, guess):
    """Return a guess of a zero of the function f, near guess."
    return
```

```python
def square(x):
    """def square(x):
        return x * x
    ""
    def make_adder(n):
        """Return a function that takes one argument k and returns k + n."
        def adder(k):
            return k + n
        return adder
    def square(x):
        """Square computes the square by calling mul."
        return x * x
    def sum_squares(x, y):
        """Return the sum of squares of x and y."
        return square(x) + square(y)
    square = lambda x, y: x * y
    A function
    with formal parameters x and y
    and body "return x * y"
    Must be a single expression
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
```

### Recursive functions
- **Recursive functions** have two important components:
  1. **Base case(s)**, where the function directly computes an answer without calling itself
  2. **Recursive case(s)**, where the function calls itself as part of the computation

```python
def square(x):
    """def square(x):
        return x * x
    """
    def make_adder(n):
        """Return a function that takes one argument k and returns k + n."
        def adder(k):
            return k + n
        return adder
    def square(x):
        """Square computes the square by calling mul."
        return x * x
    def sum_squares(x, y):
        """Return the sum of squares of x and y."
        return square(x) + square(y)
    square = lambda x, y: x * y
    A function
    with formal parameters x and y
    and body "return x * y"
    Must be a single expression
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
```

### Recursive case(s)
- **Recursive case(s)**, where the function calls itself as part of the computation
- 1. **Recursive functions** have two important components:
  1. **Base case(s)**, where the function directly computes an answer without calling itself
  2. **Recursive case(s)**, where the function calls itself as part of the computation

```python
def square(x):
    """def square(x):
        return x * x
    """
    def make_adder(n):
        """Return a function that takes one argument k and returns k + n."
        def adder(k):
            return k + n
        return adder
    def square(x):
        """Square computes the square by calling mul."
        return x * x
    def sum_squares(x, y):
        """Return the sum of squares of x and y."
        return square(x) + square(y)
    square = lambda x, y: x * y
    A function
    with formal parameters x and y
    and body "return x * y"
    Must be a single expression
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
```

### Recursive case(s)
- **Recursive case(s)**, where the function calls itself as part of the computation
- 1. **Recursive functions** have two important components:
  1. **Base case(s)**, where the function directly computes an answer without calling itself
  2. **Recursive case(s)**, where the function calls itself as part of the computation

```python
def square(x):
    """def square(x):
        return x * x
    """
    def make_adder(n):
        """Return a function that takes one argument k and returns k + n."
        def adder(k):
            return k + n
        return adder
    def square(x):
        """Square computes the square by calling mul."
        return x * x
    def sum_squares(x, y):
        """Return the sum of squares of x and y."
        return square(x) + square(y)
    square = lambda x, y: x * y
    A function
    with formal parameters x and y
    and body "return x * y"
    Must be a single expression
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
```