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.

<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)

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>Q. 6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/12</td>
<td>/5</td>
<td>/3</td>
<td>/11</td>
<td>/7</td>
<td>/12</td>
<td>/50</td>
</tr>
</tbody>
</table>
1. (12 points) Proceed with call-tion

For each of the following 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 mul
x = 3
def square(x):
    return mul(x, mul(x, 1))
def blaster(y):
    return print(square(y) + x)
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>square(7)</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>1/0</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>square(2) + square(x)</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>print(square(3))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blaster(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print(blaster(2) + 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blaster(blaster(3))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 or (5 / 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. (5 points) Lambda? No thanks, I prefer chicken
(a) (2 pt) Fill in the blanks below so that foo(5)(10)() returns [5, 10]. You may not write any numbers in your solution, and you may only add expressions in the blanks.

```
foo = lambda __________: lambda y: ________________________________
```

(b) (3 pt) Fill in the blanks below so that the final call expression below evaluates to a tuple value. For this section, you may write numbers, but not tuples, and you may only add expressions in the blanks.

```python
def love(x):
    if x == 'zedd':
        return [1, 2, lambda: (2, 3)]
    else:
        return lambda: 5

(lambda _____________, banana: foxes_____________________________)(love, 'clarity')
```

3. (3 points) Tracing through the facts
Consider the following portion of code:

```python
def tracer(fn):
    def traced(x):
        print('Calling', fn, '(\', x, '\)')
        result = fn(x)
        print('Got \', result, 'from \', fn, '(\', x, '\)')
        return result
    return traced
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
new_fact = tracer(fact)
```

Circle the Choice X heading of one of the options below corresponding to what Python would display if we ran `new_fact(2)` in an interpreter session. You may assume that the “ADDRESS” in each output is correct.

```
<table>
<thead>
<tr>
<th>Choice A</th>
<th>Choice B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling &lt;function fact at ADDRESS&gt; ( 2 )</td>
<td>Calling &lt;function fact at ADDRESS&gt; ( 2 )</td>
</tr>
<tr>
<td>Calling &lt;function fact at ADDRESS&gt; ( 1 )</td>
<td>Got 2 from &lt;function fact at ADDRESS&gt; ( 2 )</td>
</tr>
<tr>
<td>Calling &lt;function fact at ADDRESS&gt; ( 0 )</td>
<td>Calling &lt;function fact at ADDRESS&gt; ( 1 )</td>
</tr>
<tr>
<td>Got 1 from &lt;function fact at ADDRESS&gt; ( 0 )</td>
<td>Got 1 from &lt;function fact at ADDRESS&gt; ( 1 )</td>
</tr>
<tr>
<td>Got 1 from &lt;function fact at ADDRESS&gt; ( 1 )</td>
<td>Calling &lt;function fact at ADDRESS&gt; ( 0 )</td>
</tr>
<tr>
<td>Got 2 from &lt;function fact at ADDRESS&gt; ( 2 )</td>
<td>Got 1 from &lt;function fact at ADDRESS&gt; ( 0 )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Choice C</th>
<th>Choice D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling &lt;function fact at ADDRESS&gt; ( 2 )</td>
<td>2</td>
</tr>
<tr>
<td>Got 2 from &lt;function fact at ADDRESS&gt; ( 2 )</td>
<td></td>
</tr>
</tbody>
</table>

Choice E
The output is none of the above. If you select this choice, please briefly explain why:
4. (11 points) Save the environment (diagrams)!

(a) (5 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
x = 1
def thrift(x, y):
    def inner(z):
        return foo(5, 10) + z
    return inner

def foo(y, z):
    return x + y + z

shop = thrift(2, 3)
shop(7)
```

![Environment Diagram]

```
x 1
thrift
foo
```

**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 make_test(num, checker):
    def test(subm):
        return checker(subm)
    return test

def q5_checker(subm):
    return subm(10) == 15

num = 2
result = test_q5(lambda x: x + num)
```

Global frame

```
make_test
q5_checker
```

Global frame

```
func make_test(num, checker)
func q5_checker(subm)
```

Return Value

```
```

Return Value

```
```
```
5. (7 points) Testing our (pot)luck

While planning the potluck, the 61A staff decided to try and guess the number of people that would show up. In order to do this, they decided to define a new abstract data type to record everyone’s predictions. Of course, the 61A staff is bad at computer science, so they need your help to make this work!

(a) (2 pt) We want to make a prediction abstract data type that will record both a person’s name as well as their guess for the number of attendees. Based on the provided constructor `make_prediction`, fill in the definitions for the get_name and get_guess selectors.

```python
def make_prediction(name, guess):
    return (name, guess)

def get_name(prediction):
    """Gets the name of the person who made the given prediction."

    >>> get_name(make_prediction('eric', 25))
    'eric'
    ""

def get_guess(prediction):
    """Gets the number of attendees that this prediction expected to show up to the potluck."

    >>> get_name(make_prediction('eric', 25))
    25
    """
(b) (5 pt) Now complete the `print_winner` function. It takes a sequence of predictions and the actual number of attendees, and prints a congratulatory message based on whose guess was closest. You may assume that the sequence of predictions is non-empty. Ties should go to the person whose prediction appears earliest in the sequence. *Remember to respect data abstraction.*

```python
def print_winner(predictions, correct_num):
    """Given a sequence of predictions (predictions) and the actual number of attendees (correct_num), print the message '___ is the winner', where the blank is filled in with the name of the person who made the winning prediction.

>>> albert_pred = make_prediction('albert', 10000)
>>> brian_pred = make_prediction('brian', 85)
>>> mark_pred = make_prediction('mark', 97)
>>> preds = (albert_pred, brian_pred, mark_pred)
>>> print_winner(preds, 83)
brian is the winner
>>> preds2 = (make_prediction('rohan', 90), make_prediction('jeffrey', 70))
>>> print_winner(preds2, 80)
rohan is the winner
"""
```
6. (12 points) Learning to count

Steven likes to have a timer with him during lectures so that he knows how much time is left until the end of lecture. He has a timer he really likes that counts the number of seconds that have elapsed since the beginning of lecture.

Unfortunately, it turns out that the timer he bought was manufactured before humans had discovered the number six! The timer works normally, except it skips every number containing a 6 as one of its digits. For example, here are the first twenty numbers displayed by this timer:

0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21

In this example, note how it skips the numbers 6 and 16, because they both have at least one digit that is a 6. This means that when the timer displays 21, in reality only 19 seconds have passed!

Obviously, the way the timer is now isn’t very helpful for Steven. Help him solve his problem by writing a function to compute the true number of seconds that have elapsed in his lectures.

For this entire problem, do not use any loop statements. Use recursion only.

(a) (3 pt) First, complete the has_six helper function, which takes an integer and returns whether or not said integer has a 6 as one of its digits. Do not use any loop statements. Use recursion. Additionally, do not convert n to a string.

```python
def has_six(n):
    """Determines whether the integer n has a 6 as one of its digits."

    >>> has_six(123)
    False
    >>> has_six(567)
    True
    """
```
(b) (4 pt) Now, use your has_six function to complete the previous function. previous takes an integer n and determines the number that would have appeared before it on the timer. In other words, it determines the largest integer less than n that does not have a 6 as any of its digits. You may assume that n will always be a positive integer. Once again, do not use any loop statements. Use recursion.
Note: you may assume that you have a working version of has_six. You can receive full credit on this section without completing part (a).

```python
def previous(n):
    """Determines the number that showed on the timer just before n."

    >>> previous(3)
    2
    >>> previous(7)
    5
    >>> previous(70)
    59
    """
```

(c) (5 pt) Now, use your previous function to complete the num_seconds function, which takes an integer representing the number shown on the timer and returns the actual number of seconds that have elapsed. As before, do not use any loop statements. Use recursion.
Note: you may assume that you have a working version of previous. You can receive full credit on this section without completing part (b).

```python
def num_seconds(n):
    """Based on the number currently displayed on the timer, n, returns the true number of seconds that have elapsed."

    >>> num_seconds(8) # skips 6
    7
    >>> num_seconds(20) # skips 6 and 16
    18
    """
```
7. (0 points) Extra credit

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.

This is the end of the test. Feel free to use the rest of the space for scratch work. You could also draw us a picture, if you’re so inclined!
Login: ________________________________

(This page intentionally left blank)
(This page intentionally left blank)
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

**Evaluation rule for call expressions:**
1. Evaluate the operand and operand subexpressions.
2. Apply the function that is the value of the operand 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 if expressions:**
1. Evaluate the condition's expression.
2. If the result is a true value, then the expression evaluates to True.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

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

**Evaluation rule for while statements:**
1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

**Execution rule for def statements:**
1. A three-frame environment.
2. A two-frame environment.
3. A global frame.

A frame extends the environment that begins with its parent.

**Functions:**
- Built-in function: `sqrt` (square root), `abs` (absolute value), `pow` (power).
- User-defined function: `square`.
- Defining: `def`.
- Call expression: `square(2+2)`.
- Calling/Applying: `square(x)`.

**Control flow:**
- Execution rules for `if`, `for`, `while`, `return` statements.
- Control flow constructs: `if`, `else`, `elif`, `while`, `for`.

**Frames:**
- Local frame: `return mul(x, y)`.
- Global frame: `func(x, y)`.

**Environments:**
- Environment with only the global frame.
- Global environment: The environment with only the global frame.

**Non-Pure Functions:**
- `abs(number):` returns the absolute value.
- `pow(x, y):` raises `x` to the power of `y`.
- `print(...):` displays the argument value.

**Pure Functions:**
- `mul(x, y):` returns the product of `x` and `y`.
- `sum(n, term):` returns the sum of the first `n` terms of a sequence.
- `cube(k):` returns `k^3`.
- `sum(n, term):` returns the sum of the first `n` terms of a sequence.

**Error handling:**
- `Error` occurs when a frame or function body has no label and its parent is always the global frame.
- `Error` occurs when a frame or function body has no label and its parent is always the global frame.

**Higher-order functions:**
- `def abs_value(x):` takes an argument and returns its absolute value.
- `def cube(k):` takes an argument and returns its cube.
- `def sum(n, term):` takes two arguments and returns their sum.

**Examples:**
- `def cube(k): return pow(k, 3)`
- `def summation(n, term): return sum(range(n), term)`
- `>>> summation(5, cube)`
- `255`
- `The cube function is passed as an argument value`
A function is recursive if the body calls the function itself, either directly or indirectly. 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.

def square(x):
    return x * x

def make_adder(n):
    return lambda k: k + n

def find_zero(f, guess):
    while not done(guess)
        update = -f(guess)/f'(guess)
        guess += update

f = lambda x: x**2 - 2
find_zero(f, 1)