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>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/12</td>
<td>/12</td>
<td>/14</td>
<td>/8</td>
<td>/46</td>
</tr>
</tbody>
</table>
1. (12 points) The Call Express is Hijacked

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 evaluation causes an error, write Error. If an expression evaluates to a function, write Function.
- 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 square(x):
    return mul(x, x)

def pirate(arggg):
    print('matey')
    def plunder(arggg):
        return arggg
    return plunder
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square(5)</code></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td><code>1/0</code></td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td><code>print(square(4))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>square(square(print(2)))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>print(square(3), print(5))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pirate(square)(3)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add(pirate(3)(square)(4), 1)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pirate(pirate(pirate))(5)(7)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. (12 points) Protect the Environment

(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 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 horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```

Global frame

```
Global frame

horse

mask

func horse(mask)

func λ(horse)
```

Local frames:

```
Global frame

horse

mask

func horse(mask)

func λ(horse)
```

```
Local frame

Return Value
```

```
Local frame

Return Value
```

```
Local frame

Return Value
```

```
Local frame

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 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.

```
p, s, y = 1, 2, 3

def gang(p):
    nam = style(p)
    return (nam(4), 5)

def style(s):
    return lambda y: (p, s, y)

gang(3)
```
3. (14 points) Sequences

(a) (2 pt) Fill in the blanks so that the final call expression below evaluates to a tuple value.

```python
def tuple(x):
    if x == None:
        return lambda: (1, 2, 3)
    else:
        return lambda: 4
```

```python
(lamba___________: lambda soda: hall___________)(tuple)("sequence")
```

(b) (2 pt) Draw a box and pointer diagram for the following rlist:

```python
a = rlist(1, rlist((2, 3, 4), rlist(rlist(5, (6, empty_rlist)), empty_rlist)))
```

(c) (2 pt) What is the element at index 2 of this rlist, returned by getitem_rlist(a, 2)?

```python
def getitem_rlist(s, i):
    """Return the element at index i of recursive list s.""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

(d) (2 pt) What is the length of this rlist, returned by len_rlist(a)?

```python
def len_rlist(s):
    """Return the length of recursive list s.""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
```
(e) **(6 pt)** When the `int` constructor is called on a `float` value, it “truncates toward zero,” meaning that it returns the largest integer less than any positive argument, or the least integer greater than any negative argument. For example:

```python
>>> int(2)
2
>>> int(2.7)
2
>>> int(-1.5)
-1
```

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

```python
def alt(f, g, z):
    while g(z) > 0 and z != 5:
        f, g = g, f
        z = g(z)
    return z

def grow(x):
    return int((x * 3) / 2)

def shrink(x):
    return x - 2

def flip(x):
    return int(10 / (x-2))
```

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

- `alt(shrink, grow, 6)`
- `alt(shrink, grow, 7)`
- `alt(flip, shrink, 3)`
4. (12 points) In Verse

The inverse of some function $F$ is a function of argument $X$ that returns you the $Y$, such that when you apply $F$ to $Y$ you recover the $X$.

An invertible function is a function that takes and returns a single numeric value, is differentiable, and never returns the same value for two different arguments. Some examples:

```python
def double(y):
    """Return twice the value of y.""
    return 2 * y

def cube(y):
    """Return y raised to the third power.""
    return pow(y, 3)

def pow2(y):
    """Return 2 raised to the power of y.""
    return pow(2, y)
```

(a) (4 pt) Implement a function `invert` that takes an invertible function argument and returns its inverse. You may call `find_root`, `newton_update`, `approx_deriv`, and/or `iter_improve`. You cannot use any assignment, conditional, while, or for statements.

```python
def invert(f):
    """Return the inverse of invertible function f."

>>> halve = invert(double)
>>> halve(12)
6.0
>>> cube_root = invert(cube)
>>> cube_root(27)
3.0
>>> log2 = invert(pow2)
>>> log2(32)
5.0
"""
```
(b) (4 pt) A numpair is a pair of integers that have the same one’s digit. Fill in the two missing expressions in the constructor below, which takes two non-negative integers less than 100, asserts that they have the same one’s digit, and returns a numpair represented as a pair of tens digits and the shared one’s digit.

```python
from operator import floordiv, mod  # Use these functions or // and %

def numpair(first, second):
    """Return a numpair as a pair of ten’s digits and a shared one’s digit.

    >>> numpair(24, 64)
    ((2, 6), 4)
    >>> numpair(67, 7)
    ((6, 0), 7)
    """

    assert first >= 0 and first < 100 and second >= 0 and second < 100

    assert ________________________________, "different one’s"

    return ________________________________
```
Execution rule for while statements:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.
3. Otherwise, the statement ends.

Evaluation rule for conditional statements:
1. Evaluate the header's expression.
2. If it is a true value, execute the (sub)expression <right>.
3. Otherwise, the expression evaluates to the value of the (sub)expression <left>.

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:
1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the frame of the function body.

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:
1. Evaluate the expression exp; The value is True if the result is a false value, and False otherwise.

Execution 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.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for call expressions:
1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values in the first frame of the current environment.
3. Evaluate the subexpression <left>.
4. If the result is a true value, execute the suite, then skip the remaining clauses in the statement.
5. Otherwise, the expression evaluates to the value of the subexpression <right>.

Calling/Applying:
1. Evaluate <left>.
2. Evaluate <right>.

Calling: def square(x):
3. Evaluate the subexpression <right>.
4. Evaluate the operator and operand subexpressions.
5. Apply the function that is the value of the operator to the arguments that are the values of the operand subexpressions.
6. Return value is not a binding!
A sequence has an abstraction barrier between how data are manipulated (as units) and how data are represented (as parts).

Abstract data type lets us manipulate compound objects as units while keeping their parts hidden.

compound objects combine objects together

- An abstract data type lets us manipulate compound objects as units
- Programs that use data isolate two aspects of programming:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use.

A function has a parent frame
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame
- The parent of the frame in which the def statement was defined

A function's signature has all the information to create a local frame

• Every user-defined function has a parent frame
• The parent of a function is the frame in which it was defined
• Every local frame has a parent frame
• The parent of the frame in which the def statement was defined

A recursive list is a pair
- A recursive list is a pair

rlist = None
def rlist(first, rest):
    • """Make a recursive list from its first element and the rest."""
    return (first, rest)
def first(s):
    • """Return the first element of a recursive list s."""
    return s[0]
def rest(s):
    • """Return the rest of the elements of a recursive list s."""
    return s[1]

If a recursive list s is constructed from a first element f and a recursive list r, then
  • first(s) returns f, and
  • rest(s) returns r, which is a recursive list.

def len_rlist(s):
    • """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
def getitem_rlist(s, i):
    • """Return the element at index i of list s.""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Begin with a function f and an initial guess x
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)

How to find the square root of 2?
• Define a function f that returns a function
• Define a function f that returns a function
• Define a function f that returns a function

1. Define a function f that returns a function
2. Define a function f that returns a function
3. Define a function f that returns a function

 def add_three(k):
     return k + 3

def square(x):
    return x * x

def triple(x):
    return 3 * x

def add_three(x):
    return x + 3

• Both create a function with the same arguments & behavior
• Both of those functions are associated with the environment in which they are defined
• Both bind that function to the name "square"
• Only the def statement gives the function an intrinsic name

```
>>> f = lambda x: x ** 2
>>> f(3)
9

>>> g = lambda x: x ** 3
>>> g(3)
27

>>> f, g = lambda x: x ** 2, lambda x: x ** 3
>>> f(3), g(3)
(9, 27)
```

```python
from math import sin
def find_zero(f, guess, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value."
    idx = 0
    while not done(guess) and idx < max_updates:
        guess = update(guess)
        idx += 1
    return guess

def approx_derivative(f, x, delta=1e-5):
    df = (f(x + delta) - f(x)) / delta
    return df

def find_root(f, guess=0.0):
    """Guess a root of the zero of the function f, near guess."
    return find_zero(f, guess)

>>> find_root(lambda x: sin(x), 3.141592653589793)
3.141592653589793

def square(x):
    return x * x

>>> find_zero(square, 1.0)
1.0

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder

>>> make_adder(2)(3)
5
```