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 two official 61A midterm study guides 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)

<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>/14</td>
<td>/14</td>
<td>/12</td>
<td>/10</td>
<td>/50</td>
</tr>
</tbody>
</table>

For staff use only
1. (14 points) Classy Costumes

For each of the following expressions, write the value to which it evaluates. The first two rows have been provided as examples. If evaluation causes an error, write Error. If evaluation never completes, write Forever.

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

```python
class Monster:
    vampire = {2: 'scary'}
    def werewolf(self):
        return self.vampire[2]

class Blob(Monster):
    vampire = {2: 'night'}
    def __init__(self, ghoul):
        vampire = {2: 'frankenstein'}
        self.witch = ghoul.vampire
        self.witch[3] = self

spooky = Blob(Monster)
spooky.werewolf = lambda self: Monster.vampire[2]
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square(5)</code></td>
<td>25</td>
</tr>
<tr>
<td><code>1/0</code></td>
<td>Error</td>
</tr>
<tr>
<td><code>[k+2 for k in range(4)]</code></td>
<td></td>
</tr>
<tr>
<td><code>Monster.vampire[2][3]</code></td>
<td></td>
</tr>
<tr>
<td><code>repr(len(spooky.witch))</code></td>
<td></td>
</tr>
<tr>
<td><code>spooky.witch[3] is not spooky</code></td>
<td></td>
</tr>
<tr>
<td><code>spooky.witch[2][0:4]</code></td>
<td></td>
</tr>
<tr>
<td><code>spooky.werewolf()</code></td>
<td></td>
</tr>
<tr>
<td><code>Monster.werewolf(spooky)</code></td>
<td></td>
</tr>
</tbody>
</table>
2. (14 points) Wreaking Ball

(a) (8 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 miley(ray):
    def cy():
        def rus(billy):
            nonlocal cy
            cy = lambda: billy + ray
            return (1, billy)
        if len(rus(2)) == 1:
            return (3, 4)
        else:
            return (cy(), 5)
    return cy()[1]
billy = 6
miley(7)
```

Global frame

```
miley
billy 6
```

Return Value

```text``
(b) (6 pt) Write the letter of the environment diagram that would result from executing each code snippet below, just after starting Python. The first blank is filled for you. Two different snippets may result in the same environment diagram. If none of the environment diagrams are correct, write N.

1. `s = [1, 2]`  
   => 2. `t = list(s)`

 options:

- A
- B
- C
- D
- E
- F

N: None of the above
3. (12 points) Mutants

(a) (4 pt) Given two Rlist arguments a and b, the function merge(a, b) changes a so that it also includes all elements of b at the end, but does not change b. After merging, changes to b should not affect a. Assume that a is not empty, but b may be empty. Complete the implementation by filling the blanks with expressions. The Rlist class is defined in your study guide.

```python
def merge(a, b):
    """Add the elements of b to the end of a, mutating a but not b."

    >>> a = Rlist(1, Rlist(2, Rlist(3)))
    >>> b = Rlist(4, Rlist(5, Rlist(6)))
    >>> merge(a, b)
    >>> a  # a should be modified
    Rlist(1, Rlist(2, Rlist(3, Rlist(4, Rlist(5, Rlist(6))))))
    >>> b  # b should not be modified
    Rlist(4, Rlist(5, Rlist(6)))
    >>> b.first = 7  # modify the elements of b
    >>> b  # b should be modified
    Rlist(7, Rlist(5, Rlist(6)))
    >>> a  # a should not be modified
    Rlist(1, Rlist(2, Rlist(3, Rlist(4, Rlist(5, Rlist(6))))))
    ""

    assert a is not Rlist.empty

    if b is Rlist.empty:
        return  # No entries to add to a.

    elif _________________________________________________________:
        _________________________________________________________
        _________________________________________________________

    else:
        _________________________________________________________
```

(b) (2 pt) Define a mathematical function \( f(m, n) \) such that evaluating a correct and efficient implementation of merge(a, b) on Rlist a of length \( m \) and Rlist b of length \( n \) requires \( \Theta(f(m, n)) \) function calls.

\[
f(m, n) = \]

(c) (6 pt) The function \texttt{fold\_tree} takes in a three-argument function, a zero value, and a \texttt{Tree}. It returns the value of replacing \texttt{Tree} with the function and empty branches with the zero value. For each of \texttt{size}, \texttt{reverse}, and \texttt{repeated}, complete the inner function \texttt{f}. Each \texttt{f} cannot be recursive.

```python
def fold_tree(fn, zero, tree):
    """Replaces the tree constructor with a 3-argument function.

    >>> t = Tree(3, Tree(5, None, Tree(3)), Tree(2))
    >>> f = lambda a, b, c: a + b + c
    >>> fold_tree(f, 0, t)  # is equivalent to the expression...
    13
    >>> f(3, f(5, 0, f(3, 0, 0)), f(2, 0, 0))
    13
    """
    if tree is None:
        return zero
    return fn(tree.entry, fold_tree(fn, zero, tree.left),
               fold_tree(fn, zero, tree.right))

def size(tree):
    """Return the number of trees contained in tree.

    >>> size(Tree(3, Tree(5, None, Tree(3)), Tree(2)))
    4
    """
    def f(entry, left, right):
        return fold_tree(f, 0, tree)

def reverse(tree):
    """Return a new tree swapping all left and right branches of tree.

    >>> reverse(Tree(3, Tree(5, None, Tree(3)), Tree(2)))
    Tree(3, Tree(2), Tree(5, Tree(3), None))
    """
    def f(entry, left, right):
        return fold_tree(f, None, tree)

def repeated(tree):
    """Return how many times the root entry of tree appears in tree.

    >>> repeated(Tree(3, Tree(5, None, Tree(3)), Tree(2)))  # 3 appears twice
    2
    """
    def f(entry, left, right):
        return fold_tree(f, 0, tree)
```

4. (10 points) Expansion Mansion

For a fraction \( \frac{n}{d} \) with \( n < d \), its decimal expansion is written as a series of digits following a decimal point. For example, \( \frac{5}{8} \) expands to 0.625. We can compute this result recursively. The numerator 5 times 10 is 50. 50 divided by 8 is 6 with remainder 2. 20 divided by 8 is 2 with remainder 4. 40 divided by 8 is 5 with remainder 0. The quotients in bold are the digits of the expansion. Each subsequent digit is the quotient of dividing 10 times the remainder of the previous digit by the denominator of the fraction.

(a) (4 pt) Assume that the decimal expansion of \( \frac{n}{d} \) is finite, \( n \) is positive, and \( n < d \). The function `expand_finite` returns the digits of the decimal expansion as an `Rlist`. Complete it by filling in each blank with an expression. The `Rlist` class is defined in your study guide.

```python
def expand_finite(n, d):
    """Return the finite decimal expansion of \( \frac{n}{d} \) as an Rlist. Assume \( n < d \)."
    dividend = n * 10
    quotient, remainder = dividend // d, dividend % d
    if dividend == 0:
        return Rlist.empty
    else:
        return Rlist(quotient, expand_finite(10 * remainder, d))
```

(b) (2 pt) The function `coerce_to_float` returns the float equal to an input `Rlist` representing the series of digits following the decimal point in a finite decimal expansion. Complete its implementation below.

```python
def coerce_to_float(s):
    """Return a float equal to an Rlist encoding a series of digits."
    if s is Rlist.empty:
        return 0
    else:
        return coerce_to_float(s[1]) * 10 + s[0]
```

Repeating Decimal Expansions. The decimal expansion of a rational number may be infinite, but can always be described by a finite (and possibly repeating) series of digits. We can represent a series of digits as a recursive list, which may contain itself.

(c) **(4 pt)** Assume that \( n \) is positive and \( n < d \). The `expand` function returns representations of both finite and infinite decimal expansions. Complete it by filling in each blank with an expression or assignment statement.

```python
def expand(n, d):
    """Return the decimal expansion of n/d as an Rlist. Assume n < d."

    >>> expand(1, 2)  # 1/2 = 0.5
    Rlist(5)
    >>> expand(5, 8)  # 5/8 = 0.625
    Rlist(6, Rlist(2, Rlist(5)))
    >>> third = expand(1, 3)  # 1/3 = 0.333333...
    >>> [third[i] for i in range(10)]
    [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
    >>> third.rest is third  # There is only one unique digit in 1/3
    True
    >>> fourteenth = expand(1, 14)  # 1/14 = 0.0714285714285...
    >>> [fourteenth[i] for i in range(10)]
    [0, 7, 1, 4, 2, 8, 5, 7, 1, 4]
    """

    return expand_using(n, d, {})

def expand_using(n, d, known):
    """Return the decimal expansion of n/d as an Rlist.
    known -- a dictionary from integer k to the decimal expansion of k/d."
    """

    if n in known:
        return known[n]

    else:
        dividend = n * 10
        quotient, remainder = dividend // d, dividend % d

        digits = ______________________________________________________

        ____________________________________________________________

        if ____________________________________________________________:
            __________________________________________________________

        return digits
```
Evaluation rule for not expressions:
1. Evaluate the header's expression.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to False.

Evaluation rule for and expressions:
1. Evaluate the header's expression.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to False.

Evaluation rule for or expressions:
1. Evaluate the header's expression.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to True.

Execution rule for conditional 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. If it is a false value, execute the (remaining clauses in the statement).

Execution rule for def statements:
1. Create a new function value with the specified name, parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the arguments to the function's formal parameter names in that frame.
4. Execute the body of the function in the environment beginning at that frame.

Execution rule for assignment statements:
1. Evaluate the expression 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.

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 formal parameters:
1. Formal parameters are the names on the left of an assignment statement.
2. Each clause is considered in order.
3. If a formal parameter name is found in the current frame, it evaluates to the value bound to that name in the frame.
4. Otherwise, the expression evaluates to the value of the operand subexpression <right>.

Evaluation rule for def statements:
1. Create a new function value with the specified name, parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the arguments to the function's formal parameter names in that frame.
4. Execute the body of the function in the environment beginning at that frame.

Execution rule for if statements:
1. Evaluate the expression on the left of the if statement.
2. If it is a true value, execute the suite, then skip the (true clause).
3. Otherwise, execute the (false clause).

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, execute the (false clause).
4. The loop repeats until the header's expression evaluates to True.

Higher-order functions:
A function that takes a function value as an argument.

Nested function definition: Functions defined within other function bodies are bound to names in the local frame.
CS 61A Midterm 1 Study Guide – Page 2

• 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 (often global).
  • The parent of a frame is the parent of the function called.

Recursive decomposition:

1. Create a function value: func <name>-(<formal parameters>)
2. If the parent frame of that function is not the global frame, add matching labels to the parent frame and the function value (such as f1, f2, or f3).

    f1: make_adder
    func adder(k) [parent=f1]

3. Bind <name> to the function value in the first frame of the current environment.

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. If the function has a parent label, copy it to the local frame.
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

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))

Recursive decomposition:

```python
def count_partitions(n, m):
    return partitions(n, m)
def partitions(n, m):
    if n == 0:
        return 1
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

Recursive decomposition:

```python
def square(x):
    return x ** 2
```

When a function is defined:

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

Multiple assignment:

```python
f1: make_adder
func adder(k) [parent=f1]
```

```python
f2: make_adder
func f2(k) [parent=f2]
```

```python
f3: make_adder
func f3(k) [parent=f3]
```

```python
def adder(k):
    return k + n
```

```python
def square(x, y):
    return x * y
```

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

```python
def count_partitions(n, m):
    return partitions(n, m)
def partitions(n, m):
    if n == 0:
        return 1
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

```python
square = lambda x: x ** 2
```
The sequence abstraction is a collection of behaviors:

There isn't just one sequence class or abstract data type, but we can implement recursive lists as pairs. We'll use two-element tuples to encode pairs. We cannot be an instance of a mutable built-in type.

Rational number:

class ComplexRI:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return math.sqrt(real**2 + imag**2)
    @property
    def angle(self):
        return math.atan2(imag, real)
    def __repr__(self):
        return ComplexRI(self.real, self.imag)

Type dispatching: Look up a cross-type implementation of an operation based on both the operator and types of its arguments

Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

A range is a sequence of consecutive integers.

Length: ending value – starting value

Element selection: starting value + index

Tuples are immutable sequences

Dictionaries are unordered collections of key-value pairs.

Sets are unordered collections of values.

Two dictionary keys or set elements cannot be equal.

A dictionary key or set element cannot be an instance of a mutable built-in type.

Strings are sequences too:

Defining a function with a starting balance:

Creating a new frame extending the current frame.

Re-bind balance in the first non-local frame of the current environment.

A function with a non-global parent frame.

The parent contains local state.

Every call changes the balance.

Status

Effect

No non-local statement

*x* is not bound locally

Create a new binding from name *x* to object 2 in the first frame of the current environment.

No non-local statement

*x* is bound locally

Re-bind *x* to 2 in the first non-local frame of the current environment in which it is bound.

No non-local statement

*x* is not bound in a non-local frame

SyntaxError: no binding for nonlocal *x* found

No non-local statement

*x* is bound in a non-local frame

SyntaxError: name *x* is parameter and nonlocal
A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment. Statements in the `<suite>` create attributes of the class. As soon as an instance is created, it is passed to `_init_`, which is a class attribute called the constructor method.

## Dot Expression

Objects receive messages via dot notation. Dot notation accesses attributes of the instance or its class.

- `<expression>` - The `<expression>` can be any valid Python expression.
- `<name>` - The `<name>` must be a simple name.

Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`.

### To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, `<name>` is looked up in the class, which yields a class attribute value (see inheritance below).
4. That value is returned unless it is a function, in which case a bound method is returned instead.

### Assignment Statements with a Dot Expression on Their Left-Hand Side

- For an instance, then assignment sets an instance attribute
- For a class, then assignment sets a class attribute

### Inheritance: To look up a name in a class.

1. If it names an attribute in the class, return its value.
2. Otherwise, look up the name in the base class, if it exists.

### Code Examples:

**Class:**
- `Account`
- `CheckingAccount`
- `Rlist`

**Method Examples:**
- `deposit`
- `withdraw`
- `__init__`
- `check_account`

**Experiments:**
- `Rlist()`, `Rlist(0)`, `Rlist(0, 1)`
- `Tree(3)`, `Tree(1, Tree(2), Tree(2), Tree(1))`
- `Account('Jim')`, `Account.__init__(self, account_holder)`
- `deposit`, `withdraw`

**Identity Testing:**
- Binding an object to a new name using assignment does not create a new object.
- `a = Account('Jim')` creates a new account instance. There is only one Account class.

### Tree Set:
- A set is a collection of unique elements.
- It is unordered.
- No duplicates.
- The number of distinct elements is called the size.
- The set is empty if it contains no elements.
- The elements are ordered in some way.

### Growth Analysis:
- Linear growth.
- Exponential growth.
- Quadratic growth.
- Logarithmic growth.

### Example:

```python
# Set example
my_set = {1, 2, 3, 4, 5}
print(len(my_set))  # Output: 5
```

### Tree Set Example:

```python
# Tree Set
my_set = {1, 2, 3, 4, 5}
my_set.add(6)  # Add an element to the set
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}
```