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 2 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>Q. 5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/12</td>
<td>/14</td>
<td>/8</td>
<td>/8</td>
<td>/8</td>
<td>/50</td>
</tr>
</tbody>
</table>
1. (12 points) Class Hierarchy

For each row below, write the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.
Whenever the interpreter would report an error, write Error. You should include any lines displayed before an error. Reminder: The interactive interpreter displays the repr string of the value of a successfully evaluated expression, unless it is None. Assume that you have started Python 3 and executed the following:

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'My job is to gather wealth'

class Proletariat(Worker):
    greeting = 'Comrade'
    def work(self, other):
        other.greeting = self.greeting + ' ' + other.greeting
        other.work() # for revolution
        return other

ejack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>1/0</td>
<td>Error</td>
</tr>
<tr>
<td>Worker().work()</td>
<td></td>
</tr>
<tr>
<td>jack</td>
<td></td>
</tr>
<tr>
<td>jack.work()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>john.work()[10:]</td>
<td></td>
</tr>
<tr>
<td>Proletariat().work(john)</td>
<td></td>
</tr>
<tr>
<td>john.elf.work(john)</td>
<td></td>
</tr>
</tbody>
</table>
2. (14 points) Space

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

```
def locals(only):
    def get(out):
        nonlocal only
        def only(one):
            return lambda get: out
        out = out + 1
        return [out + 2]
    out = get(-only)
    return only

only = 3
earth = locals(only)
earth(4)(5)
```
(b) (6 pt) Fill in the blanks with the shortest possible expressions that complete the code in a way that results in the environment diagram shown. You can use only brackets, commas, colons, and the names `luke`, `spock`, and `yoda`. You *cannot* use integer literals, such as 0, in your answer! You also cannot call any built-in functions or invoke any methods by name.

```python
spock, yoda = 1, 2

luke = [______________________________________________________________________]

yoda = 0

yoda = [______________________________________________________________________]

yoda.append(_______________________________________________________________)
```
3. (8 points) This One Goes to Eleven

(a) (4 pt) Fill in the blanks of the implementation of \texttt{sixty\_ones} below, a function that takes a \texttt{Link} instance representing a sequence of integers and returns the number of times that 6 and 1 appear consecutively.

```python
def sixty\_ones(s):
    """Return the number of times that 1 directly follows 6 in linked list s."

    >>> once = Link(4, Link(6, Link(1, Link(6, Link(0, Link(1))))))
    >>> twice = Link(1, Link(6, Link(1, once)))
    >>> thrice = Link(6, twice)
    >>> apply\_to\_all(sixty\_ones, [Link.empty, once, twice, thrice])
    [0, 1, 2, 3]

    if ____________________________):
        return 0
    elif ____________________________:
        return 1 + ____________________________:
    else:
        return ____________________________
```

(b) (4 pt) Fill in the blanks of the implementation of \texttt{no\_eleven} below, a function that returns a list of all distinct length-\(n\) lists of ones and sixes in which 1 and 1 do not appear consecutively.

```python
def no\_eleven(n):
    """Return a list of lists of 1's and 6's that do not contain 1 after 1."

    >>> no\_eleven(2)
    [[6, 6], [6, 1], [1, 6]]
    >>> no\_eleven(3)
    [[6, 6, 6], [6, 6, 1], [6, 1, 6], [1, 6, 6], [1, 6, 1]]
    >>> no\_eleven(4)[:4] # first half
    [[6, 6, 6, 6], [6, 6, 6, 1], [6, 6, 1, 6], [6, 1, 6, 6]]
    >>> no\_eleven(4)[4:] # second half
    [[6, 1, 6, 1], [1, 6, 6, 6], [1, 6, 6, 1], [1, 6, 1, 6]]

    if n == 0:
        return ____________________________
    elif n == 1:
        return ____________________________
    else:
        a, b = no\_eleven(__________), no\_eleven(__________)
        return [__________ for s in a] + [__________ for s in b]
```
4. (8 points) Tree Time

(a) (4 pt) A GrootTree $g$ is a binary tree that has an attribute parent. Its parent is the GrootTree in which $g$ is a branch. If a GrootTree instance is not a branch of any other GrootTree instance, then its parent is BinaryTree.empty.

BinaryTree.empty should not have a parent attribute. Assume that every GrootTree instance is a branch of at most one other GrootTree instance and not a branch of any other kind of tree.

Fill in the blanks below so that the parent attribute is set correctly. You may not need to use all of the lines. Indentation is allowed. You should not include any assert statements. Using your solution, the doctests for fib_groot should pass. The BinaryTree class appears on your study guide.

Hint: A picture of fib_groot(3) appears on the next page.

class GrootTree(BinaryTree):
    """A binary tree with a parent."""
    def __init__(self, entry, left=BinaryTree.empty, right=BinaryTree.empty):
        BinaryTree._init_(self, entry, left, right)

        def fib_groot(n):
            """Return a Fibonacci GrootTree.

            >>> t = fib_groot(3)
            >>> t.entry
            2
            >>> t.parent.is_empty
            True
            >>> t.left.parent.entry
            2
            >>> t.right.left.parent.right.parent.entry
            1
            """
            if n == 0 or n == 1:
                return GrootTree(n)
            else:
                left, right = fib_groot(n-2), fib_groot(n-1)
                return GrootTree(left.entry + right.entry, left, right)
(b) (4 pt) Fill in the blanks of the implementation of `paths`, a function that takes two arguments: a `GrootTree` instance `g` and a list `s`. It returns the number of paths through `g` whose entries are the elements of `s`. A path through a `GrootTree` can extend either to a branch or its `parent`.

You may assume that the `GrootTree` class is implemented correctly and that the list `s` is non-empty.

The two paths that have entries `[2, 1, 2, 1, 0]` in `fib_groot(3)` are shown below (left). The one path that has entries `[2, 1, 0, 1, 0]` is shown below (right).

```
def paths(g, s):
    """The number of paths through g with entries s."
    
    >>> t = fib_groot(3)
    >>> paths(t, [1])
    0
    >>> paths(t, [2])
    1
    >>> paths(t, [2, 1, 2, 1, 0])
    2
    >>> paths(t, [2, 1, 0, 1, 0])
    1
    >>> paths(t, [2, 1, 2, 1, 2, 1])
    8
    """

    if g is BinaryTree.empty: ________________________________:
        return 0
    
    elif ________________________________:
        return 1
    
    else:
        
        xs = [______________________________]
        
        return sum([________________________ for x in xs])
```
5. (8 points) Abstraction and Growth

(a) (6 pt) Your project partner has invented an abstract representation of a sequence called a *slinky*, which uses a *transition* function to compute each element from the previous element. A *slinky* explicitly stores only those elements that cannot be computed by calling *transition*, using a *starts* dictionary. Each entry in *starts* is a pair of an index key and an element value. See the doctests for examples.

Help your partner fix this implementation by crossing out as many lines as possible, but leaving a program that passes the doctests. Do not change the doctests. The program continues onto the following page.

```python
def length(slinky):
    return slinky[0]
def starts(slinky):
    return slinky[1]
def transition(slinky):
    return slinky[2]

def slinky(elements, transition):
    """Return a slinky containing elements."

    >>> t = slinky([2, 4, 10, 20, 40], lambda x: 2*x)
    >>> starts(t)
    {0: 2, 2: 10}
    >>> get(t, 3)
    20
    >>> r = slinky(range(3, 10), lambda x: x+1)
    >>> length(r)
    7
    >>> starts(r)
    {0: 3}
    >>> get(r, 2)
    5
    >>> slinky([], abs)
    [0, {}, <built-in function abs>]
    >>> slinky([5, 4, 3], abs)
    [3, {0: 5, 1: 4, 2: 3}, <built-in function abs>]
    """
    starts = {}
    last = None
    for e in elements[1:]:
        for index in range(len(elements)):
            if not e:
                if index == 0:
                    return [0, {}, transition]
                if last is None or e != transition(last):
                    if e == 0 or e != transition(last):
                        if index == 0 or elements[index] != transition(elements[index-1]):
                            starts[index] = elements[index]
                            starts[index] = elements.pop(index)
                            starts[e] = transition(last)
                            starts[e] = last
                        last = e
                    return [len(starts), starts, transition]
            return [len(elements), starts, transition]
    return [len(elements), elements, transition]
```

def get(slinky, index):
    """Return the element at index of slinky."""
    if index in starts(slinky):
        return starts(slinky)[index]
    start = index
    start = 0
    f = transition(slinky)
    while start not in starts(slinky):
        while not f(get(start)) == index:
            start = start + 1
            start = start - 1
        value = starts(slinky)[start]
        value = starts(slinky)[0]
        value = starts(slinky)[index]
        while start < index:
            while value < index:
                value = f(value)
                value = value + 1
                start = start + 1
                start = start + index
    return value
    return f(value)

(b) (2 pt) Circle the \( \Theta \) expression below that describes the number of operations required to compute slinky(elements, transition), assuming that

- \( n \) is the initial length of elements,
- \( d \) is the final length of the starts dictionary created,
- the transition function requires constant time,
- the pop method of a list requires constant time,
- the len function applied to a list requires linear time,
- the len function applied to a range requires constant time,
- adding or updating an entry in a dictionary requires constant time,
- getting an element from a list by its index requires constant time,
- creating a list requires time that is proportional to the length of the list.

\[
\Theta(1) \quad \Theta(n) \quad \Theta(d) \quad \Theta(n^2) \quad \Theta(d^2) \quad \Theta(n \cdot d)
\]
Scratch Paper
Scratch Paper
Execution rule for while statements:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.
3. If it is a false value, continue with the next line.

Evaluation rule for not expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to not v.
3. Otherwise, the expression evaluates to v.

Evaluation rule for and 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 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 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 else clause, if present.

Evaluation rule for call expressions:
1. Evaluate the operand subexpression.
2. If the result is a function value, apply it to the arguments.
3. Otherwise, the expression evaluates to the value of the operand subexpression.

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 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 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 true 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 operand subexpression.
2. If the result is a true value v, then the expression evaluates to not v.
3. Otherwise, the expression evaluates to v.

Evaluation rule for conditional statements:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.
3. If it is a false value, use the else clause, if present.

High-order functions:
- A function that takes a function as an argument.
- A function that returns a function.
def square(x):
    return x * x

square = lambda x: x * x

Evaluate to a function. No "return" keyword!

A function

with formal parameters x and y

that returns the value of "k + y"

Must be a single expression

def add_three(n):
    """Add 3 to the argument n."""
    return n + 3

>>> add_three = make_adder(3)
>>> add_three(12)
15

A function that returns a function

That name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

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

def curry2(f):
    """Returns a function g such that g(x, y) returns f(x,y)."""
    def g(x, y):
        return f(x, y)
    return g

def curve2(f):
    """Returns a function g such that g(x,y) returns f(x,y)."""
    def g(x, y):
        return f(x, y)
    return g

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Anatomy of a recursive function:
- The def statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls
- Recursive cases are evaluated with recursive calls
- The parent of a frame is the parent of the function called

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = n % 10, n // 10
        return sum_digits(all_but_last) + last

Recursive decomposition:
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        return count_parts(n, m-1) + count_partitions(n-m, m)

Recursive decomposition:

def count_partitions(n):
    """Return the number of ways to write n as a sum of positive integers, separated by commas."
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        return count_partitions(n-1) + count_partitions(n, m-1)

Recursive decomposition:

def count_partitions(n):
    """Return the number of ways to write n as a sum of positive integers, separated by commas."
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        return count_partitions(n-1) + count_partitions(n, m-1)

Recursive decomposition:

def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        return count_partitions(n, m-1) + count_partitions(n-m, m)

Recursive decomposition:

def count_partitions(n):
    """Return the number of ways to write n as a sum of positive integers, separated by commas."
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        return count_partitions(n-1) + count_partitions(n, m-1)

Recursive decomposition:

def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        return count_partitions(n, m-1) + count_partitions(n-m, m)
List comprehensions:

```python
[<map exp> for <name> in <iter exp> if <filter exp>]
```

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty result list that is the value of the expression
3. For each element in the iterable value of `<iter exp>`:
   A. Bind `<name>` to that element in the new frame from step 1
   B. If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list

```python
def apply_to_all(map_fn, s):
    """Apply map_fn to each element of s."
    return apply_all(lambda x: map_fn(x), range(len(s)))

>>> apply_to_all(lambda x: x**3, range(5))
[0, 1, 8, 27, 64]
```

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

Type coercion: For convenience, convert one type to another, then apply a type-specific implementation.

```python
def pair(x, y):
    """Return a functional pair."
    return (x, y)

def get(index):
    if index == 0:
        return x
    else:
        return y

def select(p, i):
    """Return element i of pair p."
    return p(i)
```

Functional pair implementation:

```python
def make_withdrawal(parent=GlobalFrame):
    def withdraw(amount):
        if amount > balance:
            return 'No funds'
        balance -= amount
        return balance
```

Short version:

```python
[(1, 2, 3, 4)]
```

Type coercion:

```python
>>> x = 'Berkeley'
>>> city(x)
'Berkeley'
```

If both `<exp0>` and `<exp1>` evaluate to equal values, `<exp2>` evaluates to True if both `<exp0>` and `<exp1>` evaluate to equal values.

Identity:

```python
>>> x = 'sword' 'sword'
>>> x is 'sword'
True
```

Equality:

```python
>>> x = 5
>>> y = 5
>>> x == y
True
```

Lists:

```python
>>> digits = [1, 8, 2, 8]
```

Unpacking in a for statement:

```python
>>> x, y in pairs:
>>> same_count = 0
```

Type coercion:

```python
>>> x = 10
>>> y = 20
```

Equality:

```python
>>> x = 10
>>> y = 20
>>> x == y
True
```

Execution a for statement:

```python
for name in expression:
    suite
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
   A. Bind `<name>` to that element in the current frame
   B. Execute the `<suite>`

Unpacking in a for statement:

```python
>>> pairs = [[10, 20], (30, 40)]
```

Type coercion:

```python
>>> x = 20
>>> 10
```

Equality:

```python
>>> x = 10
>>> y = 10
>>> x == y
True
```

Lists & dictionary mutation:

```python
>>> nums = {3: 9, 4: 16, 5: 25}
```

Type coercion:

```python
>>> x = 10
>>> 10
```

Equality:

```python
>>> x = 10
>>> y = 10
>>> x == y
True
```

List & dictionary mutation:

```python
>>> a = [10, 10, 10]
>>> b = [10, 10, 10]
```

Type coercion:

```python
>>> x = 10
>>> 10
```

Equality:

```python
>>> x = 10
>>> y = 10
>>> x == y
True
```
Linked list data abstraction:

```python
def link(first, rest):
    return [first, rest]

def first(s):
    return s[0]
def rest(s):
    return s[1]
def len_link(s):
    x = 0
    while s != empty:
        s, x = rest(s), x + 1
    return x

def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)

def extend(s, t):
    assert is_link(s) and is_link(t)
    if s == empty:
        return t
    else:
        return link(first(s), extend(rest(s), t))

def apply_to_all_link(f, s):
    if s == empty:
        return s
    else:
        return link(apply_to_all_link(f, first(s)), apply_to_all_link(f, rest(s)))
```

Python object system:

**Idea:** All bank accounts have a `balance` and an account `holder`.
The Account class should add those attributes to each of its instances.

When a class is called:
1. A new instance of that class is created.
2. The `__init__` method of the class is called with the new object as its first argument (name `self`), along with any additional arguments provided in the call expression.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

def deposit(self, amount):
    self.balance = self.balance + amount
    return self.balance

def withdraw(self, amount):
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance
```

Function call: all arguments within parentheses

Method invocation:
One object before the dot and other arguments within parentheses

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression:
- If the object is a class, then assignment sets a class attribute.
- If it names an attribute in the class, its value is returned.
- Otherwise, look up the name in the base class, if there is one.

The `__len__` function for a list.

```python
The 0-indexed element of the pair is the first element of the linked list

The 1-indexed element of the pair is the rest of the linked list
```