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)

<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>/14</td>
<td>/8</td>
<td>/6</td>
<td>/40</td>
</tr>
</tbody>
</table>

For staff use only
1. (12 points) World Cup

(a) (10 pt) For each of the expressions in the tables below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. Whenever the interpreter would report an error, write ERROR. You should include any lines displayed before an error.

Reminder: the interactive interpreter displays the value of a successfully evaluated expression, unless it is None. The first three rows have been provided as examples. Assume that you have started Python 3 and executed the following statements:

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

def argentina(n):
    print(n)
    if n > 0:
        return lambda k: k(n+1)
    else:
        return 1 / n

def germany(n):
    if n > 1:
        print('hallo')
        if argentina(n-2) >= 0:
            print('bye')
        return argentina(n+2)
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>print(5)</td>
<td>5</td>
</tr>
<tr>
<td>1/0</td>
<td>ERROR</td>
</tr>
<tr>
<td>print(1, print(2))</td>
<td></td>
</tr>
<tr>
<td>argentina(0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>argentina(1)(square)</td>
<td></td>
</tr>
<tr>
<td>germany(1)(square)</td>
<td></td>
</tr>
<tr>
<td>germany(2)(germany)</td>
<td></td>
</tr>
</tbody>
</table>

(b) (2 pt) Fill in the blank with an expression so that the whole expression below evaluates to a number. Hint: The expression abs > 0 causes a TypeError.

```
(lambda t: argentina(t)(germany)(square))(______________________________)
```
2. (14 points) Envy, Iron, Mint

(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 peace(today):
    harmony = love+2
    return harmony + today(love+1)

def joy(peace):
    peace, love = peace+2, peace+1
    return love // harmony

love, harmony = 3, 2
peace(joy)
```

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>peace</td>
</tr>
<tr>
<td></td>
<td>joy</td>
</tr>
<tr>
<td></td>
<td>love</td>
</tr>
<tr>
<td></td>
<td>harmony</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>f1: [parent=_]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>f2: [parent=_]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>f3: [parent=_]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

func peace(today) [parent=Global]  
func joy(peace) [parent=Global]
(b) (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 k(g, b):
    def n(s, a):
        return g - p
    return b(n(b, p))
g, p = 3, 7
k(p+1, lambda s: g+3)
```

<table>
<thead>
<tr>
<th>Global frame</th>
<th>k</th>
<th></th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>g</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1: ________ [parent=__________]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f2: ________ [parent=__________]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f3: ________ [parent=__________]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
3. (8 points) Express Yourself

(a) (3 pt) A k-bonacci sequence starts with K-1 zeros and then a one. Each subsequent element is the sum of the previous K elements. The 2-bonacci sequence is the standard Fibonacci sequence. The 3-bonacci and 4-bonacci sequences each start with the following ten elements:

\[ n: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots \]
\[ k\text{bonacci}(n, 2): 0, 1, 1, 2, 3, 5, 8, 13, 21, 35, \ldots \]
\[ k\text{bonacci}(n, 3): 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, \ldots \]
\[ k\text{bonacci}(n, 4): 0, 0, 0, 1, 1, 2, 4, 8, 15, 29, \ldots \]

Fill in the blanks of the implementation of kbonacci below, a function that takes non-negative integer n and positive integer k and returns element n of a k-bonacci sequence.

```python
def kbonacci(n, k):
    """Return element N of a K-bonacci sequence."
    if n < k - 1:
        return 0
    elif n == k - 1:
        return 1
    else:
        total = 0
        i = ________________
        while i < n:
            total = total + ________________
            i = i + 1
        return total
```

```
(b) (5 pt) Fill in the blanks of the following functions defined together in the same file. **Assume that all arguments to all of these functions are positive integers that do not contain any zero digits.** For example, 1001 contains zero digits (not allowed), but 1221 does not (allowed). You may assume that `reverse` is correct when implementing `remove`.

```python
def combine(left, right):
    """Return all of LEFT’s digits followed by all of RIGHT’s digits."""
    factor = 1
    while factor <= right:
        factor = factor * 10
    return left * factor + right

def reverse(n):
    """Return the digits of N in reverse.
    >>> reverse(122543)
    345221
    >>>
    if n < 10:
        return n
    else:
        return combine(__________________________ , __________________________)

def remove(n, digit):
    """Return all digits of N that are not DIGIT, for DIGIT less than 10.
    >>> remove(243132, 3)
    2412
    >>> remove(243132, 2)
    4313
    >>> remove(remove(243132, 1), 2)
    433
    >>>
    removed = 0
    while n != 0:
        __________ , __________ = ____________________ , ____________________
        if ___________________________________________________________________
            removed = _________________________________________________________
    return reverse(removed)
```

4. (6 points) Lambda at Last

(a) (2 pt) Fill in the blank below with an expression so that the second line evaluates to 2014. You may only use the names two_thousand, two, k, four, and teen and parentheses in your expression (no numbers, operators, etc.).

\[
\text{two_thousand} = \lambda \text{two}: \lambda \text{k}: \quad \text{\ldots}
\]
\[
\text{two_thousand}(7)(\lambda \text{four}: \lambda \text{teen}: 2000 + \text{four} + \text{teen})
\]

(b) (4 pt) The if_fn returns a two-argument function that can be used to select among alternatives, similar to an if statement. Fill in the return expression of factorial so that it is defined correctly for non-negative arguments. You may only use the names if_fn, condition, a, b, n, factorial, base, and recursive and parentheses in your expression (no numbers, operators, etc.).

```python
def if_fn(condition):
    if condition:
        return lambda a, b: a
    else:
        return lambda a, b: b

def factorial(n):
    """Compute N! for non-negative N. \( N! = 1 \times 2 \times 3 \times \ldots \times N. \)"
    def base():
        return 1
    def recursive():
        return n * factorial(n-1)

    return ____________________________________________________________________
```

```python
>>> factorial(3)
6
>>> factorial(5)
120
>>> factorial(0)
1
"""
### Execution rule for while statements:
- Evaluate the header's expression.
- If the result is a true value, execute the (remaining clauses in the statement).
- If the result is a false value v, then the expression evaluates to the value of the operand subexpressions.

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

### Evaluation rule for assignment statements:
- Simultaneously bind the names on the left to those values, its parent is the first frame of the current environment.

### 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 <left>.

### 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 <left>.

### Evaluation rule for not expressions:
1. Evaluate <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.
```python
def square(x):
    return x * x

# vs
def square(x):
    return x * x
```

- 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.

When a function is defined:

1. Create a function value: `func <name> (<formal parameters>)`
2. Its parent is the current frame.
   ```python
   f1: make_adder = func adder(k)[parent=f1]
   ```
3. Bind `<name>` to the function value in the current frame (which is the first frame of the current environment).
4. Execute the body of the function in the environment that starts with the local frame.

```python
1 def fact(n):
   2 if n == 0:
   3 return 1
   4 else:
   5 return n * fact(n-1)
   6
   7 fact(3)
   ```

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.

```python
def sum_digits(n):
    return the sum of the digits of positive integer n.
```

```python
s = lambda x, y: x + y
A function
with formal parameters x and y
that returns the value of "x + y"
```