INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn’t work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address <EMAILADDRESS>. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with circular bubbles, you should select exactly one choice.

- You must choose either this option
- Or this one, but not both!

For questions with square checkboxes, you may select multiple choices.

- You could select this choice.
- You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.
Preliminaries
You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?
1. (8.0 points) Political Environment

Fill in each blank in the code example below so that executing it would generate the following environment diagram.

**REQUIREMENTS.** You must use all of the blanks. Each blank can only include one statement or expression. Click here to open the diagram in a new window/tab.

```python
def v(o, t, e):
    def m(y):
        (a)
    def n(o):
        o.append((b))

def v(o, t, e):  # (parent=Global)
    def m(y):
        (a)
    def n(o):
        o.append((b))
```

```python
def v(o, t, e):
    def m(y):
        (a)
    def n(o):
        o.append((b))
```

---

```
def v(o, t, e):
    def m(y):
        (a)
    def n(o):
        o.append((b))
```

---

```
def v(o, t, e):
    def m(y):
        (a)
    def n(o):
        o.append((b))
```

---

```
def v(o, t, e):
    def m(y):
        (a)
    def n(o):
        o.append((b))
```
(c)  
```python
o.append(_________)
```

(d)  
```python
m(e)  
n([t])  
e = 2
```

```python
m = [3, 4]
```

```python
v(m, 5, 6)
```

(a) (2.0 pt) Fill in blank (a). You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

```
nonlocal o
```

(b) (2.0 pt) Fill in blank (b). You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

```
o = [o, t]
```

(c) (2.0 pt) Fill in blank (c). You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

```
[c]
```

(d) (2.0 pt) Which of these could fill in blank (d)? Check all that apply.

- [ ] o
- [ ] [o]
- [x] list(o)
- [ ] [list(o)]
- [ ] list([o])
- [x] o + []
- [x] [o[0], o[1]]
- [x] o[:]

```
2. (10.0 points) Yield, Fibonacci!

(a) (4.0 points)
Implement `fibs`, a generator function that takes a one-argument pure function `f` and yields all Fibonacci numbers `x` for which `f(x)` returns a true value.

The Fibonacci numbers begin with 0 and then 1. Each subsequent Fibonacci number is the sum of the previous two. Yield the Fibonacci numbers in order.

```python
def fibs(f):
    """Yield all Fibonacci numbers x for which f(x) is a true value."
    n, m = 0, 1
    while _________:
        (a)
        if _________:
            (b)
            _________
            _________
            (c)
            _________
            (d)
```

i. (1.0 pt) Which of these could fill in blank (a)?

- f(n)
- f(m)
- f(n) or f(m)
- f(n) and f(m)
- True
- False
ii. (1.0 pt) Which of these could fill in blank (b)?
   - f(n)
   - f(m)
   - f(n) or f(m)
   - f(n) and f(m)
   - True
   - False

iii. (1.0 pt) Fill in blank (c).

   \[ \text{yield n} \]

iv. (1.0 pt) Fill in blank (d).

   \[ n, m = m, n + m \]
Definition. For a linked list $s$, the index of an element is the number of times rest appears in the smallest dot expression containing only $s$, rest, and first that evaluates to that element. For example, in the linked list $s = \text{Link}(5, \text{Link}(7, \text{Link}(9, \text{Link}(11))))$,

- The index of 5 ($s\.first$) is 0.
- The index of 7 ($s\.rest\.first$) is 1.
- The index of 11 ($s\.rest\.rest\.rest\.first$) is 3.

Implement \texttt{filter\_index}, a function that takes a one-argument pure function $f$ and a Link instance $s$. It returns a Link containing all elements of $s$ that have an index $i$ for which $f(i)$ returns a true value.

Assume that $s$ is a finite linked list of numbers that contains no repeated elements. The Link class appears on Page 2 (left column) of the Midterm 2 Study Guide.

```python
def filter_index(f, s):
    """Return a Link containing the elements of Link s that have an index i for which f(i) is a true value."""
    >>> powers = Link(1, Link(2, Link(4, Link(8, Link(16, Link(32))))))
    >>> filter_index(lambda x: x < 4, powers)
    Link(1, Link(2, Link(4, Link(8))))
    >>> filter_index(lambda x: x % 2 == 1, powers)
    Link(2, Link(8, Link(32)))
    """
    
    def helper(i, s):
        if s is Link.empty:
            return s
        filtered_rest = _________
        (a)
        if _________:
            (b)
            return _________
            (c)
        else:
            return filtered_rest
        return _________
        (d)
```
i. (1.0 pt) Which of these could fill in blank (a)?
   - helper(i + 1, s.rest)
   - helper(i + 1, s.rest.rest)
   - filter_index(f, s.rest)
   - filter_index(f, s.rest.rest)
   - Link(helper(i + 1, s.rest))
   - Link(helper(i + 1, s.rest.rest))
   - Link(filter_index(f, s.rest))
   - Link(filter_index(f, s.rest.rest))

ii. (1.0 pt) Fill in blank (b).

   f(i)

iii. (2.0 pt) Fill in blank (c).

   Link(s.first, filtered_rest)

iv. (2.0 pt) Fill in blank (d).

   helper(0, s)
3. (12.0 points) Sparse Lists

The `most_common` function returns the most common element in a non-empty list. You do not need to implement this function. Assume that it is implemented for you.

```python
def most_common(s):
    """Return the most common element in non-empty list s. In case of a tie, return the most common element that appears first in s."
    >>> most_common([3, 1, 4, 1, 5, 9])
    1
    >>> most_common([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])
    5
    >>> most_common([2, 7, 1, 8, 2, 8, 1, 8, 2, 8])
    8
    >>> most_common([3, 5, 7, 7, 7, 5, 5])
    5
    >>> most_common([3, 7, 5, 5, 7, 7])
    7
    ""
```

Implement the `SparseList` class. A `SparseList` instance represents a non-empty list `s`.

- Its `common` attribute is the most common value in `s`.
- Its `others` dictionary has a value for every element in `s` that is not `common`. The corresponding key is the `index` for that value in `s`.
- Its `item` method takes a non-negative integer `i` and returns `s[i]` or the string `'out of range'` if `i` is not smaller than the length of `s`.
- Its `items` method returns a list with the same elements as `s` in the same order as `s`.

class SparseList:
    """Represent a non-empty list as a most common value and a dictionary from indices to values that contains only values that are not the most common.
    >>> pi = SparseList([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])
    >>> pi.common
    5
    >>> pi.others
    {0: 3, 1: 1, 2: 4, 3: 1, 5: 9, 6: 2, 7: 6, 9: 3}
    >>> [pi.item(0), pi.item(1), pi.item(2), pi.item(3), pi.item(4)]
    [3, 1, 4, 1, 5]
    >>> pi.item(10)
    5
    >>> pi.item(11)
    'out of range'
    >>> pi.items()
    [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
    ""
    def __init__(self, s):
        ...
    def item(self, i):
        ...
    def items(self):
        ...

(a) (5.0 points)

Implement the `__init__` method, which takes a list `s`.
def __init__(self, s):
    assert s, 's cannot be empty'
    self.n = len(s)
    self.common = most_common(_________)
                     (a)
    self.others = { _________: _________ for i in range(_________) if _________ }
                     (b)        (c)        (d)        (e)

i. (1.0 pt) Fill in blank (a).

\[
\text{s}
\]

ii. (1.0 pt) Which of these could fill in blank (b)?

- i
- \#self.i
- \#n
- \#self.n
- \#s[i]
- s

iii. (1.0 pt) Fill in blank (c).

\[
\text{s[i]}
\]

iv. (1.0 pt) Which of these could fill in blank (d)? **Check all that apply.**

- s
- \#len(s)
- \#self.s
- \#len(self.s)
- n
- \#len(n)
- \#self.n
- \#len(self.n)

v. (1.0 pt) Fill in blank (e).

\[
\text{s[i] != self.common}
\]
(b) (3.0 points)

Implement the `item` method, which takes a non-negative integer `i`.

```python
def item(self, i):
    """Return `s[i]` or 'out of range' if `i` is not smaller than the length of `s`."""

    assert i >= 0, 'index `i` must be non-negative'

    if ___________
        (a)

        return 'out of range'

    elif ___________
        (b)

        return __________

    else:

        return self.common

i. (1.0 pt) Fill in blank (a).

```
i >= self.n```

ii. (1.0 pt) Fill in blank (b).

```
i in self.others```

iii. (1.0 pt) Which of these could fill in blank (c)?

- others[i]
- self.others[i] [Correct]
- others[self.i]
- self.others[self.i]
- others
- self.others
(c) (4.0 points)
Implement the `items` method.

```python
def items(self):
    """Return a list with the same elements as s in the same order as s."""
    return [________ for i in _______]
    (a)                       (b)

i. (2.0 pt) Fill in blank (a). You may not use and, or, if, else, [ ], or get.
   *Hint:* Don’t repeat yourself.

   ```python
   self.item(i)
   ```

ii. (2.0 pt) Fill in blank (b).

   ```python
   range(self.n)
   ```
4. (20.0 points) Fork It

The tree data abstraction, which is implemented by the constructor tree, selectors branches and label, and helper functions is_leaf and is_tree appear on Page 2 (left column) of the Midterm 2 Study Guide. You may call these functions. Do not violate the abstraction barriers of the tree data abstraction.

(a) (4.0 points)

Implement max_path, which takes a tree t whose labels are all positive numbers and returns the largest sum of the labels along a path from the root of t to one of its leaves.

You may call tree, label, branches, is_leaf, is_tree, and max_path.

def max_path(t):
    """Return the largest sum of labels along any path from the root to a leaf of tree t, which has positive numbers as labels.
    >>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
    >>> max_path(a) # 1 + 4 + 5
    10
    >>> b = tree(6, [a, a, a])
    >>> max_path(b) # 6 + 1 + 4 + 5
    16
    """

    return __________ + max(_________ + _________)
    (a) (b) (c)

i. (1.0 pt) Which of the following could fill in blank (a)?

  ○ t
  ● label(t)
  ○ [t]
  ○ [label(t)]
  ○ [0]
  ○ sum([b for b in branches(t)])
  ○ sum([label(b) for b in branches(t)])

ii. (1.0 pt) Which of the following could fill in blank (b)?

  ○ t
  ○ label(t)
  ○ [t]
  ○ [label(t)]
  ● [0]
  ○ [label(b) for b in branches(t)]

iii. (2.0 pt) Fill in blank (c). You may not use the word default.

[max_path(b) for b in branches(t)]
(b) (8.0 points)

**Definition.** A *fork* is a tree in which exactly one node has more than one child.

---

**Forks:**
(\textit{The blue node is the one with more than one child.})

---

**Not Forks:**

---

Implement `is_fork` and its helper function `slide`. The `is_fork` function takes a `tree t` and returns `True` if `t` is a fork and `False` otherwise.

You may call `tree`, `label`, `branches`, `is_leaf`, `is_tree`, `max_path`, `is_fork`, and `slide`.

```python
def is_fork(t):
    """Return whether tree t is a fork."
    >>> is_fork(tree(1, [tree(2, [tree(3), tree(4), tree(5)]),
    tree(6)]))
    True
    >>> is_fork(tree(1, [tree(2, [tree(3)]), tree(4)]))
    True
    >>> is_fork(tree(1, [tree(2), tree(3), tree(4)]))
    True
    >>> is_fork(tree(1, [tree(2, [tree(3, [tree(5)])], tree(4), [tree(6)])])))
    True
    >>> is_fork(tree(1))
    False
    >>> is_fork(tree(1, [tree(2, [tree(3)])]))
    False
```
>>> is_fork(tree(1, [tree(2, [tree(3)]), tree(4, [tree(5), tree(6)])]))
False
>>> is_fork(tree(1, [tree(2, [tree(3, [tree(5, [tree(7), tree(8)])], tree(6)), tree(4)])]))
False

```python
neck = slide(t)
if is_leaf(neck):
    return _________
(a)

return _________([_________ for b in branches(_________)])
(b) (c) (d)

def slide(t):
    """Return the deepest node within tree t whose ancestors all have exactly one child."
    Definition: The ancestors of a node include its parent and the parents of all its ancestors.

>>> deepest = slide(tree(1, [tree(2, [tree(3)])]))
>>> label(deepest)
3
>>> label(slide(tree(1, [tree(2, [tree(3), tree(4)])])))
2

while _________:
    (e)

    t = _________
(f)

return t
```

i. (1.0 pt) Fill in blank (a).

return False

ii. (1.0 pt) Which of the following could fill in blank (b)?

- all
- any
- list
- tree
- branches
- label
iii. (2.0 pt) Fill in blank (c).

\[
is\_leaf(slide(b))
\]

iv. (1.0 pt) Which of the following could fill in blank (d)?

- t
- tree
- neck
- label(t)
- label(tree)
- label(neck)

v. (1.0 pt) Which of the following could fill in blank (e)?

- \( \text{len(branches(t))} == 1 \)
- \( \text{len(branches(t))} > 1 \)
- \( \text{len(branches(t))} != 1 \)
- \( \text{branches(t)} == 1 \)
- \( \text{branches(t)} > 1 \)
- \( \text{branches(t)} != 1 \)

vi. (2.0 pt) Fill in blank (f).

\[
\text{branches(t)[0]}
\]
(c) (8.0 points)

**Definition.** A tree $t$ contains a fork $u$ if $u$ is the result of pruning zero or more nodes from $t$.

Some forks contained in trees:
(The fork nodes are blue & black; pruned tree nodes are white)

Implement `max_fork`, which takes a tree $t$ whose labels are all positive integers. It returns the largest sum of the labels of a fork that is contained in $t$. If $t$ does not contain any forks, then `max_fork` returns 0.

You may call `tree`, `label`, `branches`, `is_leaf`, `is_tree`, `max_path`, `is_fork`, `slide`, and `max_fork`.

```python
def max_fork(t):
    """Return the largest sum of the labels in any fork contained in tree t, which has positive numbers as labels. If t contains no forks, return 0."

    n = len(branches(t))
    if n == 0:
        return 0
    elif n == 1:
        below = _________
        (a)
        if _________:
            (b)
```

```python
>>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
>>> max_fork(a) # 1 + 2 + 3 + 4 + 5
15
>>> b = tree(6, [a, a, a])
>>> max_fork(b) # 6 + (1 + 4 + 5) + (1 + 4 + 5) + (1 + 4 + 5)
36
>>> c = tree(7, [tree(8), b, tree(9)])
>>> max_fork(c) # 7 + (6 + (1 + 4 + 5) + (1 + 4 + 5) + (1 + 4 + 5))
43
>>> d = tree(9, [c])
>>> max_fork(d) # 9 + 7 + (6 + (1 + 4 + 5) + (1 + 4 + 5) + (1 + 4 + 5))
52
>>> max_fork(tree(1, [tree(2, [tree(3)])])) # No forks here!
0
"""```
return _________ + below
     (c)

else:

    return 0

else:

    here = sum([_________ for b in branches(t)])
     (d)

    there = max([_________ for b in branches(t)])
     (e)

    return label(t) + max(here, there)

i. (2.0 pt) Fill in blank (a).

max_fork(branches(t)[0])

ii. (1.0 pt) Which of the these could fill in blank (b)?

- below > 0
- label(t) > 0
- max_fork(t) > 0
- True
- max_fork(t) > max_path(t)
- is_fork(t)

iii. (1.0 pt) Which of these could fill in blank (c)?

- 1
- label(t)
- max_path(t)
- label(slide(t))
- max([label(b) for b in branches(t)])
- label(branches(t)[0])

iv. (2.0 pt) Fill in blank (d).

max_path(b)
v. (2.0 pt) Fill in blank (e).

max_fork(b)
No more questions.