CS 61A Structure and Interpretation of Computer Programs
Spring 2021

Final Solutions

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?

(c) By writing my name below, I pledge on my honor that I will abide by the rules of this exam and will neither give nor receive assistance. I understand that doing otherwise would be a disservice to my classmates, dishonor me, and could result in me failing the class.

1. (6.0 points) Link-Like Lists

Fill in each blank in the code example below so that executing it would generate the following environment diagram on tutor.cs61a.org.

RESTRICTIONS. You must use all of the blanks. Each blank can only include one statement or expression.

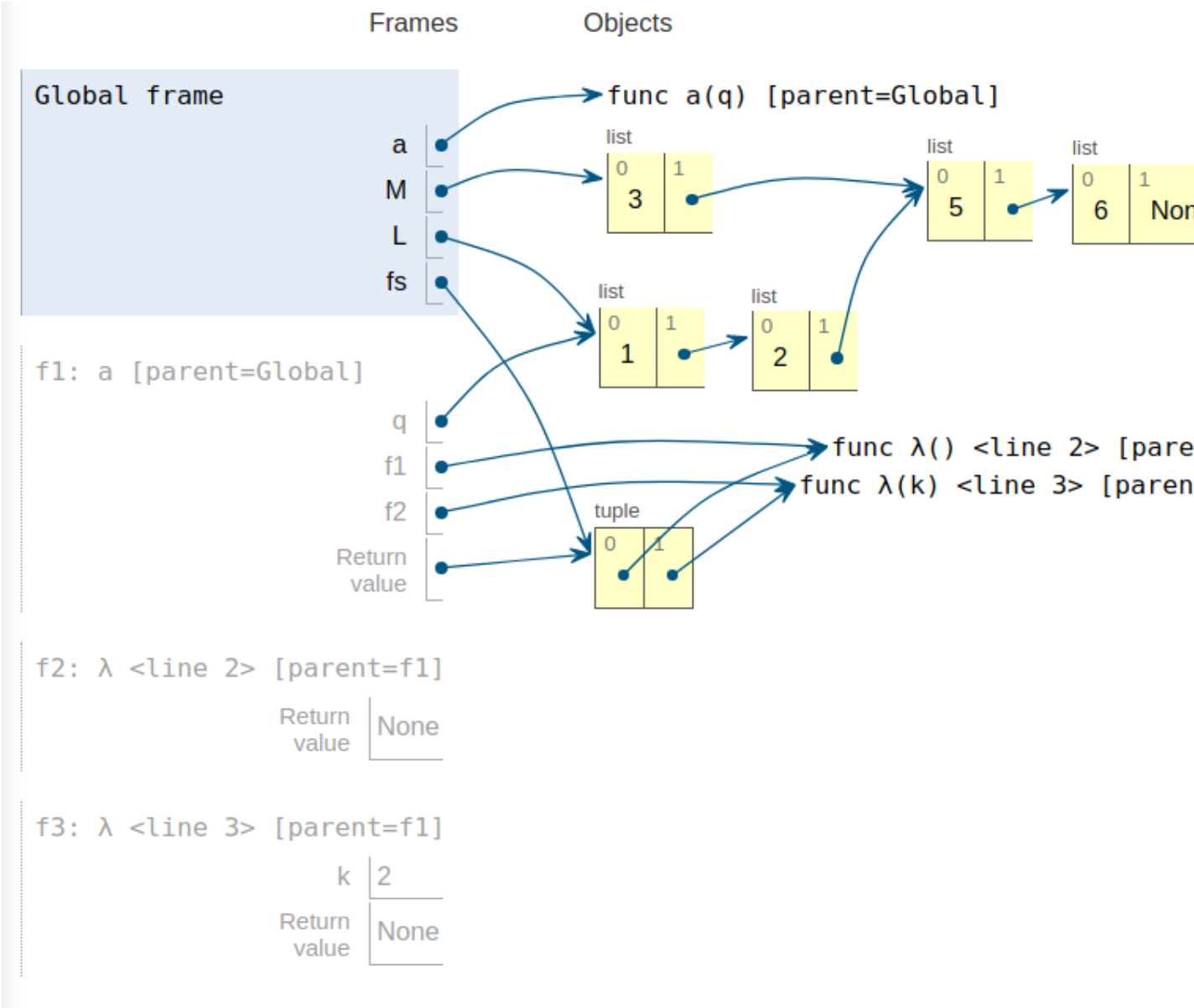
Useful Information: In Python, the function call

`X.__setitem__(k, Y)`

is the dunder-method equivalent of

`X[k] = Y`

Click here to open the diagram in a new window/tab




```
def a(q):

    f1 = lambda: _______.__setitem__(1, ___________________)
    #              (a)                          (b)
```

```
    f2 = lambda k: _______.__setitem__(1, ________________)
    #                (a)                        (c)

    return f1, f2

M = [3, [5]]
L = [1, M]

fs = a(L)

fs[0]()
fs[1](2)

--------------------------
         (d)

--------------------------
         (e)
```

(a) **(1.0 pt)** Which of these could fill in **both** of the blanks labeled (a)? **Select all that apply.**

- [x] `q`
- [ ] `q[1]`
- [ ] `q[0]`
- [ ] `M`
- [x] `L`

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

- ● `q[1][1]`
- ○ `q[1][0]`
- ○ `q[1][1][0]`
- ○ `q[0]`
- ○ `q[1]`

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

- ● `[k, q[1]]`
- ○ `[k]`
- ○ `[k, q[0]]`
- ○ `[k, q]`
- ○ `[k, q[1][0]]`

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

- [x] `M[1].append([6])`
- [ ] `M[1].extend([6])`
- [ ] `M[1].append(6)`
- [x] `L[1][1].append([6])`
- [ ] `L[1][1].append([2 * L[1][0]])`

(e) (2.0 pt) Fill in blank (e).

M[1][1].extend([None])

2. (11.0 points) What Can Separate Us?

The following problems involve our standard CS61A `Link` class and the `toLinked` function from Lecture 36:

```
def toLinked(L):
    """Returns a linked-list representation of the Python iterable L."""
    if len(L) == 0:
        return Link.empty
    result = last = Link(L[0], Link.empty)
    for item in L[1:]:
        last.rest = Link(item)
        last = last.rest
    return result
```

(a) (8.0 points)

Implement the function `gather`, which takes a linked list `L` and returns a linked list of two linked lists: first the list `L` after removing all items that are originally surrounded by two equal items; and second, the list of all the removed items. For example, if we mark items to be removed in the list

`<1 3 4 3 5 6 5 1 0>`

with asterisks, we get

`<1 3 *4* 3 5 *6* 5 1 0>,`

so that `gather` would return

`<<1 3 3 5 5 1 0> <4 6>>`

The computation proceeds left to right; when two equal items are found separated by a third, the third item is removed and not considered in further removals. Thus, in a sequence like `<1 2 1 2>`, only the first 2 is removed. That is, the list is treated as `<1 *2* 1 2>`, and the second `1` is not removed: `gather` returns

`<<1 1 2> <2>>`

Also, only items *originally* surrounded by equal values are considered, so that the list `<1 1 2 1>` is treated as `<1 1 *2* 1>` and `gather` returns

`<<1 1 1> <2>>`

Even though the second `1` is surrounded in the *resulting* list.

```
def gather(L):
    """Returns a linked list containing two lists: comp and sep.
    The list comp contains the values of L, but with all items that were
    initially surrounded by two equal values removed.  The list sep is the
    list of removed values, in order from left to right.
    The operation is non-destructive.
    >>> L = toLinked([0, 1, 3, 1, 3, 4, 5, 6, 5, 6, 7, 8, 7, 7, 9])
    >>> # L is <0 1 *3* 1 3 4 5 *6* 5 6 7 *8* 7 7 9>,
    >>> R = gather(L)
    >>> print(R)
    <<0 1 1 3 4 5 5 6 7 7 7 9> <3 6 8>>
    >>> print(L)  # Check that original list not mutated
    <0 1 3 1 3 4 5 6 5 6 7 8 7 7 9>
    >>> print(gather(toLinked([1] * 10))) # <1 *1* 1 *1* 1 *1* 1 *1* 1 1>
    <<1 1 1 1 1 1> <1 1 1 1>>
    >>> print(gather(toLinked([1, 2, 3, 1, 4])))
    <<1 2 3 1 4> ()>
    >>> print(gather(toLinked([1, 1])))
    <<1 1> ()>
    >>> print(gather(toLinked([1, 1, 2, 1])))  # <1 1 *2* 1>
```

```
    <<1 1 1> <2>>
    """
    def compressed(R):

        if ____________________________________________________________:
        #                                 (a)
            return R
        if ____________________________________________________________:
        #                                 (b)
            return ________________________________________________
            #                             (c)
        return ____________________________________________________________
        #                                 (d)

    def separators(R):
        if ____________________________________________________________:
        #                               (a)
            return ___________________________________
            #                     (e)
        if ____________________________________________________________:
        #                                 (b)
            return Link(_____________________________________________,
            #                           (f)
                        ________________________________________)
                       #              (g)
        return ____________________________________________________________
        #                                 (h)

    return toLinked([compressed(L), separators(L)])
```

i. **(1.0 pt)** Which of these could fill in the two blanks labeled (a)?

- ○ `R is Link.empty`
- ○ `R.rest is Link.empty`
- ○ `R.rest.rest is Link.empty`
- ○ `R.rest.rest.rest is Link.empty`
- ● `R is Link.empty or R.rest is Link.empty or R.rest.rest is Link.empty`

ii. **(1.0 pt)** Which of these could fill in the two blanks labeled (b)?

- ○ `R == R.rest.rest`
- ○ `R.first == R.rest.first`
- ○ `R.rest is not Link.empty and R.first == R.rest.first`
- ● `R.first == R.rest.rest.first`

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

- ◯ `compressed(R.rest)`
- ◯ `compressed(R.rest.rest)`
- ◯ `Link(R, R.rest.first)`
- ● `Link(R.first, compressed(R.rest.rest))`
- ◯ `Link(R.rest.first, compressed(R.rest.rest))`

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

- ◯ `compressed(R.rest)`
- ◯ `compressed(Link(R.first, R.rest))`
- ● `Link(R.first, compressed(R.rest))`
- ◯ `Link(R.first, compressed(R.rest.rest))`
- ◯ `Link(R.first, compressed(R.rest.first))`

v. **(1.0 pt)** What can fill in blank (e)?

Link.empty

vi. **(1.0 pt)** What can fill in blank (f)?

- ◯ `R`
- ◯ `R.first`
- ● `R.rest.first`
- ◯ `R.rest.rest.first`
- ◯ `R.rest`

vii. **(1.0 pt)** What can fill in blank (g)?

separators(R.rest.rest)

viii. **(1.0 pt)** What can fill in blank (h)?

separators(R.rest)

(b) (3.0 points)

The `dgather` function destructively computes the first item of the value returned by `gather`. That is, it takes a linked list `L` and returns `L` after removing all items that are originally surrounded by two equal items. The operation is destructive, so that the original data cannot be reconstructed in general. The function should not create any new Links.

```
def dgather(L):
    """Returns a linked list containing the values of L, but with all
    items that were initially surrounded by two equal values removed.
    The operation is destructive.
    >>> L = toLinked([0, 1, 3, 1, 3, 4, 5, 6, 5, 6, 7, 8, 7, 7, 9])
    >>> R = dgather(L)
    >>> print(R)
    <0 1 1 3 4 5 5 6 7 7 7 9>
    >>> print(dgather(toLinked([1] * 10)))
    <1 1 1 1 1 1>
    >>> print(dgather(toLinked([1, 2, 3, 1, 4])))
    <1 2 3 1 4>
    >>> print(dgather(toLinked([1, 1])))
    <1 1>
    >>> print(dgather(toLinked([1, 1, 2, 1])))
    <1 1 1>
    """
    compressed = L
    while ____________________________________________________________:
    #                           (a)
          if ____________________________________________________:
          #                     (b)
              ____________________________________________________
              #                 (c)
          L = L.rest

    return compressed
```

i. (1.0 pt) What can fill in blank (a)? **Select all that apply**.

- ☐ `L is not Link.empty`
- ☐ `L is not Link.empty and L.rest is not Link.empty`
- ☐ `L is not Link.empty and L.rest.rest is not Link.empty`
- ■ `L is not Link.empty and L.rest is not Link.empty and L.rest.rest is not Link.empty`

ii. (1.0 pt) What can fill in blank (b)?

- ○ `L == L.rest.rest`
- ○ `L.first == L.rest.first`
- ○ `L.first != L.rest.first`
- ○ `L.rest is not Link.empty and L.first == L.rest.first`
- ● `L.first == L.rest.rest.first`

iii. (1.0 pt) fill in blank (c) with one statement.

L.rest = L.rest.rest

3. (22.0 points) Tree Match-Maker

In this problem, we'll use our standard CS61A `Tree` class:

```
class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(repr(self.label), branch_str)
```

(a) (10.0 points)

A *tree pattern* is a tree that may contain *pattern variables*, which are leaves whose labels are strings that begin with `$`. In a tree pattern, pattern variables are also the *only* nodes whose labels start with `$`. A tree pattern P *matches* a tree `T` if either

- P is a pattern variable, or
- P's label equals T's label, P and T have the same number of children, and the children of P match the corresponding children of T.

In addition, if P contains multiple instances of a pattern variable labeled, say, `$X`, then each instance of a pattern variable labeled `$X` must match an identical subtree within `T`.

The function `tree_match` tests to see if a tree pattern matches a tree, returning either the value `False` if not, or a dictionary containing the matches for any pattern variables in the tree pattern if they do match. The dictionary returned can be empty (`{}`) if the tree pattern matches literally. For example,

```
>>> tree_match(Tree(1, [Tree(2)]), Tree(1, [Tree(2)]))
{}
>>> tree_match(Tree(1, [Tree(2)]), Tree(1, [Tree(3)]))
False
```

Careful: not all tree labels have to be strings!

```
def tree_match(pattern, tree):
    """If PATTERN matches all of TREE, returns a dictionary that maps
    the pattern-variable labels in PATTERN to subtrees of TREE that they
    match.  Otherwise, returns False.
    >>> T = Tree("+", [ Tree(0), Tree("*", [Tree(2), Tree(3)])])
    >>> P1 = Tree("+", [ Tree(0), Tree("$X") ])
```

```
>>> P2 = Tree("*", [ Tree(0), Tree("$X") ])
>>> P3 = Tree("+", [ Tree("$X"), Tree("$Y") ])
>>> P4 = Tree("+", [ Tree("$X"), Tree("$X") ])
>>> tree_match(P1, T)
{'$X': Tree('*', [Tree(2), Tree(3)])}
>>> tree_match(P2, T)
False
>>> tree_match(P3, T)
{'$X': Tree(0), '$Y': Tree('*', [Tree(2), Tree(3)])}
>>> tree_match(P4, T)
False
>>> tree_match(T, T)
{}
"""
result = { }

def matcher(p, t):
    if ___________________________________________:
    #                       (a)
        if ___________________________________________:
        #                       (b)
            return False
        ________________________________________________
        #                     (c)
    elif ________________________________________________:
    #                       (d)
        return False
    else:
        for k in range(len(p.branches)):
            if ____________________________________________:
            #                     (e)
                return False
    return True
return ______________________________
#                 (f)
```

i. (1.0 pt) Which of these could fill in blank (a)? **Select all that apply.**

- ☐ `p.is_leaf()`
- ☐ `p.label[0] == '$'`
- ☐ `p.is_leaf() and p.label[0] == '$'`
- ☐ `p.label != '' and p.label[0] == '$'`
- ■ `p.label != '' and str(p.label)[0] == '$'`
- ■ `(str(p.label) + " ")[0] == '$'`

ii. (1.0 pt) Which of these could fill in blank (b)? **Select all that apply.**

- ☐ `result[p.label] != t`
- ■ `p.label in result and result[p.label] != t`
- ■ `result.get(p.label, t) != t`
- ☐ `p.label in result and result[p.label] == t`

iii. **(3.0 pt)** Fill in blank (c).

result[p.label] = t

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

- ☐ `p.label != t.label and p.branches != t.branches`
- ☐ `p.label != t.label or p.branches != t.branches`
- ☐ `p.label != t.label and len(p.branches) != len(t.branches)`
- ■ `p.label != t.label or len(p.branches) != len(t.branches)`
- ■ `(p.label, len(p.branches)) != (t.label, len(t.branches))`

v. **(3.0 pt)** Fill in blank (e).

not matcher(p.branches[k], t.branches[k])

vi. **(1.0 pt)** Which of these could fill in blank (f)?

- ● `matcher(pattern, tree) and result`
- ○ `result`
- ○ `result and matcher(pattern, tree)`
- ○ `False if result is False else matcher(pattern, tree)`
- ○ `matcher(pattern, tree)`

(b) (5.0 points)

The `tree_subst` function non-destructively computes the result of replacing subtrees of a tree that match a given pattern. For each matching subtree, T, the replacement is computed by `update(d)`, where `d` is the dictionary returned by `tree_match` for T. Replacement happens from the bottom of the tree up; a subtree is matched only after all of its children have been substituted by `tree_subst`.

Assume that the function `tree_match` from the previous question works as specified.

```
def tree_subst(pattern, update, tree):
    """Return a tree that results from replacing each subtree of TREE that
    matches PATTERN (as defined by tree_match) with UPDATE(d), where d is
    the dictionary mapping pattern variable labels to trees that is returned by
    tree_match. Matching proceeds from the leaves up, with the pattern
    always being matched to the subtree resulting from
    performing substitution on its children.  The function is non-destructive.
    >>> T = Tree("*", [Tree("+", [Tree(10),
    ...                           Tree("+", [Tree(0),
    ...                                      Tree("*", [Tree(2), Tree(3)])])]),
    ...                 Tree("+", [Tree(0), Tree(0)])])
    >>> P1 = Tree("+", [ Tree(0), Tree("$X") ])
    >>> tree_subst(P1, lambda d: d['$X'], T)
    Tree('*', [Tree('+', [Tree(10), Tree('*', [Tree(2), Tree(3)])]), Tree(0)])
    >>> T
    Tree('*', [Tree('+', [Tree(10), Tree('+', [Tree(0), Tree('*', [Tree(2), Tree(3)])])]), Tree('+
    >>> tree_subst(P1, lambda d: d['$X'], tree_subst(P1, lambda d: d['$X'], T))
    Tree('*', [Tree('+', [Tree(10), Tree('*', [Tree(2), Tree(3)])]), Tree(0)])
    >>> tree_subst(Tree('+', [Tree(1), Tree(2)]),
    ...            lambda d: Tree(3),
    ...            Tree('*', [Tree('+', [Tree(1), Tree(2)]), Tree(7)]))
    Tree('*', [Tree(3), Tree(7)])
    """

    new_tree = Tree(tree.label, ____________________________________________)
    #                                          (a)
    d = tree_match(pattern, new_tree)

    if ____________________________________________________
    #                           (b)
        return ____________________________________________
        #                           (c)
    else:
        return new_tree
```

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

- (●) `[tree_subst(pattern, update, b) for b in tree.branches]`
- ( ) `map(tree_subst, tree.branches)`
- ( ) `[b for b in tree.branches]`
- ( ) `[update(b) for b in tree.branches]`
- ( ) `[update(tree_subst(pattern, update, b)) for b in tree.branches]`

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

d is not False

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

update(d)

(c) (7.0 points)

The `dtree_subst` function produces the same tree as `tree_subst`, but does so destructively. **Only the `update` function may create new trees.**

```
def dtree_subst(pattern, update, tree):
    """Return a tree that results from replacing each subtree of TREE that
    matches PATTERN (as defined by tree_match) with UPDATE(d), where d is
    the dictionary mapping pattern variable labels to trees that is returned by
    tree_match.  Matching proceeds from the leaves up, with the pattern
    always being matched to the subtree resulting from
    performing substitution on its children.  The function is destructive.
    >>> T = Tree("*", [Tree("+", [Tree(10),
    ...                           Tree("+", [Tree(0),
    ...                                      Tree("*", [Tree(2), Tree(3)])])]),
    ...                Tree("+", [Tree(0), Tree(0)])])
    >>> P1 = Tree("+", [ Tree(0), Tree("$X") ])
    >>> dtree_subst(P1, lambda d: d['$X'], T)
    Tree('*', [Tree('+', [Tree(10), Tree('*', [Tree(2), Tree(3)])]), Tree(0)])
    >>> T
    Tree('*', [Tree('+', [Tree(10), Tree('*', [Tree(2), Tree(3)])]), Tree(0)])
    >>> dtree_subst(P1, lambda d: d['$X'], T)
    Tree('*', [Tree('+', [Tree(10), Tree('*', [Tree(2), Tree(3)])]), Tree(0)])
    >>> dtree_subst(P1, lambda d: d['$X'],
    ...             Tree('+', [Tree('+', [Tree(0), Tree(0)]),
    ...                        Tree(5)]))
    Tree(5)
    >>> T = Tree('*', [Tree(5), Tree(10)])
    >>> T0 = T.branches[0]
    >>> dtree_subst(P1, lambda d: Tree(0), T) # Should not change tree.
    Tree('*', [Tree(5), Tree(10)])
    >>> T0 is T.branches[0]    # Check that new node not created.
    True
    """

    for k in ___________________________________________:
    #                            (a)

        ____________________________________________________________________
        #                       (b)
    d = tree_match(pattern, tree)
    if <same as blank (b) in the tree_subst problem>:
        return <same as blank (c) in tree_subst problem>
    else:
        return ____________________________________________________________
        #                           (c)
```

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

range(len(tree.branches))

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

> **tree.branches[k] = dtree_subst(pattern, update, tree.branches[k])**

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

> **tree**

iv. (2.0 pt) I'd like to use `dtree_subst` to replace all subtrees of `T` of the form `Tree('+', [Tree(A), Tree(B)])`, where `A` and `B` are integers, with `Tree(C)`, where `C` is the sum of `A` and `B`. We'll assume that Trees with integer labels are leaves. I use the call

```
dtree_subst(Tree('+', [Tree('$A'), Tree('$B')]), compute_subst, T)
```

and define `compute_subst` like this:

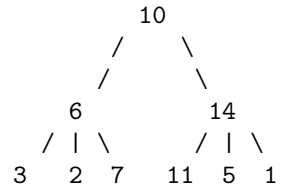
```
def compute_subst(d):
    A = d['$A'].label
    B = d['$B'].label

    if type(A) is int and type(B) is int:

       return Tree(A + B)
    else

       return ____________________________________________________
```

What should replace the blank?

- ◯ `T`
- ◯ `d['$A']`
- ◯ `d['$B']`
- ● `Tree('+', [d['$A'], d['$B']])`
- ◯ `Tree('+', [Tree(d['$A']), Tree(d['$B'])])`
- ◯ `Tree('+', [A, B])`

4. (8.0 points) Defer To Your Ancestors

The generator `humble` traverses a tree with numeric labels, yielding only labels of nodes that are strictly less than those of all their ancestors. So for a tree



`humble` would generate the labels `3 2 6 5 1 10` in that order.

```
def humble(tr, smallest_ancestor=float('inf')):
    """Yield the labels of the Tree TR that are strictly less than
    SMALLEST_ANCESTOR and the labels of all ancestors of TR
    in postorder (bottom to top, left to right).
    >>> T = Tree(10, [Tree(6, [Tree(3), Tree(2), Tree(7)]),
    ...               Tree(14, [Tree(11), Tree(5), Tree(1)])])
    >>> for p in humble(T):
    ...     print(p)
    3
    2
    6
    5
    1
    10
    """

    for _____________________________________________________:
    #                              (a)
        yield ____________________________________________________________
        #                                      (b)
    if ____________________________________________________:
    #                            (c)
        ____________________________________________________________
        #                         (d)
```

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

b in tr.branches:

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

from humble(b, min(tr.label, smallest_ancestor))

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

tr.label < smallest_ancestor

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

```
yield tr.label
```

5. (15.0 points) Sorting Out Scheme

There are *many* ways to sort a list of numbers (an entire volume of Knuth's famous *The Art of Computer Programming* series goes into many in exhaustively loving detail.) Let's consider a simple *insertion sort*. The algorithm here is to sort all but the first element of the input list and then insert the first element of the input into the sorted part just before the first value that is strictly larger (or at the end if there is no larger value). So, for example, to sort `(4 0 5 1)`:

- Sort (0 5 1):
  - Sort (5 1):
    - Sort (1) -> Returns (1)
    - Insert 5 into (1) -> Returns (1 5)
  - Insert 0 into (1 5) -> Returns (0 1 5)
- Insert 4 into (0 1 5) -> Returns (0 1 4 5)

(a) (4.0 points)

First, we'll implement `insert`, which inserts a number into an already sorted list of numbers. The `expect` tests at the end give examples.

```
(define (insert val L)
  ;;; Assuming that L is a list of numbers sorted in non-descending order,
  ;;; and VAL is a number, return the list resulting from inserting VAL into
  ;;; L so that the resulting list is also sorted.
  (cond ((null? L) _____________________________)
  ;                              (a)
        (_________________________ ______________________________)
        ;          (b)                        (c)
        (else ____________________________________________________)))
        ;                            (d)

; Some tests
(expect (insert 5 '(0 3 6 9 11)) (0 3 5 6 9 11))
(expect (insert 3 '()) (3))
(expect (insert 12 '(0 3 6 9 11)) (0 3 6 9 11 12))
(expect (insert -1 '(0 3 6 9 11)) (-1 0 3 6 9 11))
```

i. (1.0 pt) What can fill in blank (a)? **Select all that apply.**

- ☐ `'()`
- ☐ `L`
- ☐ `val`
- ☐ `(cons val)`
- ■ `(list val)`
- ■ `(cons val L)`

ii. (1.0 pt) What can fill in blank (b) in such a way that neither (b) nor (c) contains a recursive call? **Select all that apply.**

- ■ `(< val (car L))`
- ☐ `(< (car L) val)`
- ■ `(<= val (car L))`
- ☐ `(<= (car L) val)`

iii. (1.0 pt) What can fill in blank (c)?

- ○ `L`
- ● `(cons val L)`
- ○ `(cons L val)`
- ○ `(append val L)`
- ○ `(append L val)`

iv. (1.0 pt) What can fill in blank (d)?

- ○ `(cons val (insert (car L) (cdr L)))`
- ● `(cons (car L) (insert val (cdr L)))`
- ○ `(insert (car L) (cons val (cdr L)))`
- ○ `(insert val (cons (car L) (cdr L)))`
- ○ `(append (car L) (insert val (cdr L)))`

(b) (4.0 points)

Next, assuming that we have a working `insert` function, implement `insertion-sort`, which sorts a list of numbers into non-descending order. Again, the `expect` tests at the end give examples.

```
(define (insertion-sort L)
  (if _________________________  ________________________
  ;             (e)                     (f)
      (insert ___________________________________ ____________________________)))
      ;                   (g)                                 (h)

(expect (insertion-sort '(2 3 5 7 9)) (2 3 5 7 9))
(expect (insertion-sort '()) ())
(expect (insertion-sort '(42)) (42))
(expect (insertion-sort '(9 8 7 6 5 4 3 2 1)) (1 2 3 4 5 6 7 8 9))
```

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

(null? L)

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

'()

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

(car L)

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

(insertion-sort (cdr L))

(c) (2.0 points)

i. (1.0 pt) In the process of executing `insertion-sort` on a list of length N, how many `cons` operations are performed in the **best** case?

- ○ $\Theta(1)$
- ○ $\Theta(\lg N)$
- ● $\Theta(N)$
- ○ $\Theta(N^2)$
- ○ $\Theta(N^3)$
- ○ $\Theta(2^N)$

ii. (1.0 pt) In the process of executing `insertion-sort` on a list of length N, how many `cons` operations are performed in the **worst** case?

- ○ $\Theta(1)$
- ○ $\Theta(\lg N)$
- ○ $\Theta(N)$
- ● $\Theta(N^2)$
- ○ $\Theta(N^3)$
- ○ $\Theta(2^N)$

(d) (5.0 points)

In C, C++, and Java, one can write loops like this one:

```
v = 0;
for (k = 1; k < 10; k += 1) {
    v = v + k*k
}
```

to sum up the squares of numbers from 0 to 9. This is equivalent to the Python loop

```
v = 0
k = 1
while k < 10:
    v = v + k*k
    k += 1
```

Let's define a macro to do the same in Scheme. Here, I'd like to be able to write

```
(for k 1 (< k 10) (+ k 1) v 0 (+ v (* k k)))
```

and have it return the final value of `v`, as computed by the Scheme equivalent of the loops above. Fill in the following macro definition to do so:

```
(define-macro (for control-var control-init test incr result-var result-init body)
  `(begin
     (define ($loop$ ,control-var _____________________________)
     ;                                (a)
       (if ____________________________________________
                                (b)
           ($loop$ _____________________________________)
           #                    (c)
           ____________________________________))
           #                    (d)
     ($loop$ ________________________________________)))
     ;                                (e)

; Tests
(expect (for x 1 (< x 10) (+ x 1) v 0 (+ v x)) 45)
(expect (for L '(1 2 3 4 5 6 7 8 9) (not (null? L)) (cdr L) v 0 (+ v (car L)))
        45)
(expect (for x 0 (< x 5) (+ x 1) v '() (append v (list (* x x))))
        (0 1 4 9 16))
```

i. (1.0 pt) What can you fill in (a)?

- ◯ `,control-init`
- ◯ `control-init`
- ● `,result-var`
- ◯ `result-var`
- ◯ `,incr`
- ◯ `incr`

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

,test

iii. (1.0 pt) What can fill in blank (c)?

- (●) `,incr ,body`
- ( ) `,(cons incr body)`
- ( ) `,control-var ,incr`
- ( ) `,control-var ,body`
- ( ) `,(cons control-var body)`

iv. (1.0 pt) What can fill in blank (d)?

- (●) `,result-var`
- ( ) `,result-init`
- ( ) `,body`
- ( ) `,control-var`
- ( ) `,control-init`

v. (1.0 pt) What can fill in blank (e)?

- (●) `,control-init ,result-init`
- ( ) `control-init result-init`
- ( ) `,control-var ,control-init`
- ( ) `control-var control-init`
- ( ) `,result-init ,control-init`
- ( ) `result-init control-init`

6. (3.0 points) No Ifs, Ands, or Buts

Consider the following variation on our usual `Link` class:

```
class Link:
    """A linked-list node.

    >>> L = Link(1, Link(2, Link(3, Link(4))))
    >>> L.map(lambda x: x*x)
    Link(1, Link(4, Link(9, Link(16))))
    >>> L # Should be non-destructive
    Link(1, Link(2, Link(3, Link(4))))
    >>> Link.empty.map(lambda x: x*x)
    Link.empty
    """
    def __init__(self, first, rest=None):
        self.first = first
        self.rest = rest or Link.empty

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def map(self, func):

        return ____________________________________________________
        #                            (a)

class EmptyLink(Link):

    def __init__(self):
        pass

    def __repr__(self):
        return "Link.empty"

    def map(self, func):

        return ____________________________________________________
        #                           (b)


Link.empty = EmptyLink()
```

As you can see, we have replaced the usual value of `Link.empty` with an object whose type is a subtype of `Link`. The idea is to fill in the blanks so that it obeys the doctests of `Link`. **However, your code may not use `if`, `and`, or `or`.**

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

Link(func(self.first), self.rest.map(func))

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

Link.empty

7. (7.0 points) Regularize

Consider the following list of regular expressions.

(a) `(\d,\s*)+`
(b) `(\d,\s*)*`
(c) `(\d+,\s*)+`
(d) `(\d+,\s*)*`
(e) `\d+(,\s*\d+)*`
(f) `(\d+(,\s*\d+)*)?`
(g) `[e-z]*a[e-z]*b[e-z]*c[e-z]*d[e-z]*`
(h) `[e-z]+a[e-z]+b[e-z]+c[e-z]+d[e-z]+`
(i) `([e-z]*[a-d]){4}[e-z]*`
(j) `([a-f]*[g-z]*)*`
(k) `[^(]*([(].*[)])?[^(]*`
(l) `[^()]*([(][^()]*[)])*[^()]*`
(m) `.*([(][^()]*[)])*.*`

For each of the following descriptions, fill in the blank that follows with the number (1-13) of the regular expression that describes it. In each case, the regular expression must match all strings that fit the description in their entirety.

(a) (1.0 pt) "All lists of one or more decimal integer numerals, each one except the last followed by a comma and optional whitespace. For example, `3, 2, 5`."

5

(b) (1.0 pt) "All lists of zero or more decimal integer numerals, each one except the last followed by a comma and optional whitespace. (A zero-item list is simply the empty string.)"

6

(c) (1.0 pt) "All lists of zero or more decimal integer numerals, each (including the last) followed by a comma and optional whitespace."

4

(d) (1.0 pt) "All sequences of lower-case letters containing each of the letters `a`, `b`, `c`, and `d` exactly once in that order."

7

(e) (1.0 pt) "All sequences of 0 or more lower-case letters."

10

(f) (1.0 pt) "All sequences of characters other than newlines and carriage returns in which any opening parenthesis is eventually followed by at least one closing parenthesis (they need not balance, however; in fact, the same closing parenthesis may serve to follow any number of preceding opening parentheses, as in `x(b)c)d)`)."

11

(g) (1.0 pt) "All sequences of characters other than newlines and carriage returns in which all parentheses are balanced and no pair of parentheses is nested inside another. For example `ab`, `(c)d(fg)h`, or `(abc)`, but not `(ab))` or `)a(`."

12

8. (3.0 points) Backus Up

(a) (3.0 pt) The syntax trees produced by the following Lark grammar for a calculator will be processed to produce the appropriate calculated value. That is, we process from the leaves up, evaluating each subtree as we go up to yield an appropriate integer number.

```
start: expr1

expr1: expr2 | expr1 "//" expr2

expr2: NUMBER | NUMBER "-" expr2

NUMBER: /\d+/

%ignore /\s+/
```

Assuming that the operators have the same meanings they do in Python, what would be the value computed for

```
96 // 8 - 2 - 2 // 3
```

?

- ◯ 2
- ● 4
- ◯ 8
- ◯ 10
- ◯ 12
- ◯ 48
- ◯ 96

9. (1.0 points) Extra!

(a) (1.0 pt) An eccentric billionaire has bought a group of islands somewhere in the Pacific, and is using them to raise (and thus preserve) some colonies of rare animal species. He has fenced each of these islands, dividing them into separate regions surrounded by fences, each of which houses exactly one of the species (all different). Each fence consists of sections that run between two fenceposts; any fencepost can be a terminus of any number of sections. The fences on each individual island are connected together. The total number of fenceposts on all the islands is 1000, with 1500 sections, enclosing 510 species. How many islands are there?

10

No more questions.