CS 61A Structure and Interpretation of Computer Programs

Fall 2011

Final Exam Solutions

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except a one-page crib sheet of your own creation and the three official 61A exam study guides, which are 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.

Last name	
First name	
SID	
Login	
TA & section time	
Name of the person to your left	
Name of the person to your right	
For which assignments do you have unresolved regrade requests?	
All the work on this exam is my own. **(please sign)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6	Total
/12	/16	/12	/18	/10	/12	/80

1. (12 points) What Would Python Print?

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

```
def oracle(a, b, c):
    if a == 42:
        return b(a)
    return c

class Big(object):
    def medium(self, d):
        def small(e):
            nonlocal d
            if e > 0:
                d = d + self.medium(e)(-e)
            return d
        return small

class Huge(Big):
    def medium(self, d):
        return Big.medium(self, d+1)
```

For each of the following expressions, write the `repr` string (i.e., the string printed by Python interactively) of the value *to which it evaluates* in the current environment. If evaluating the expression causes an error, write "Error." Any changes made to the environment by the expressions below *will affect* the subsequent expressions.

(a) (2 pt) `oracle(41, lambda x: 1/0, 'blue pill')`

`'blue pill'`

(b) (2 pt) `oracle(42, lambda x: 'red pill', 1/0)`

`Error`

(c) (2 pt) `Big.medium(self, -2)(-1)`

`Error`

(d) (2 pt) `Big().medium(1)(Big().medium(2)(3))`

`6`

(e) (2 pt) `Huge().medium(4)(5)`

`11`

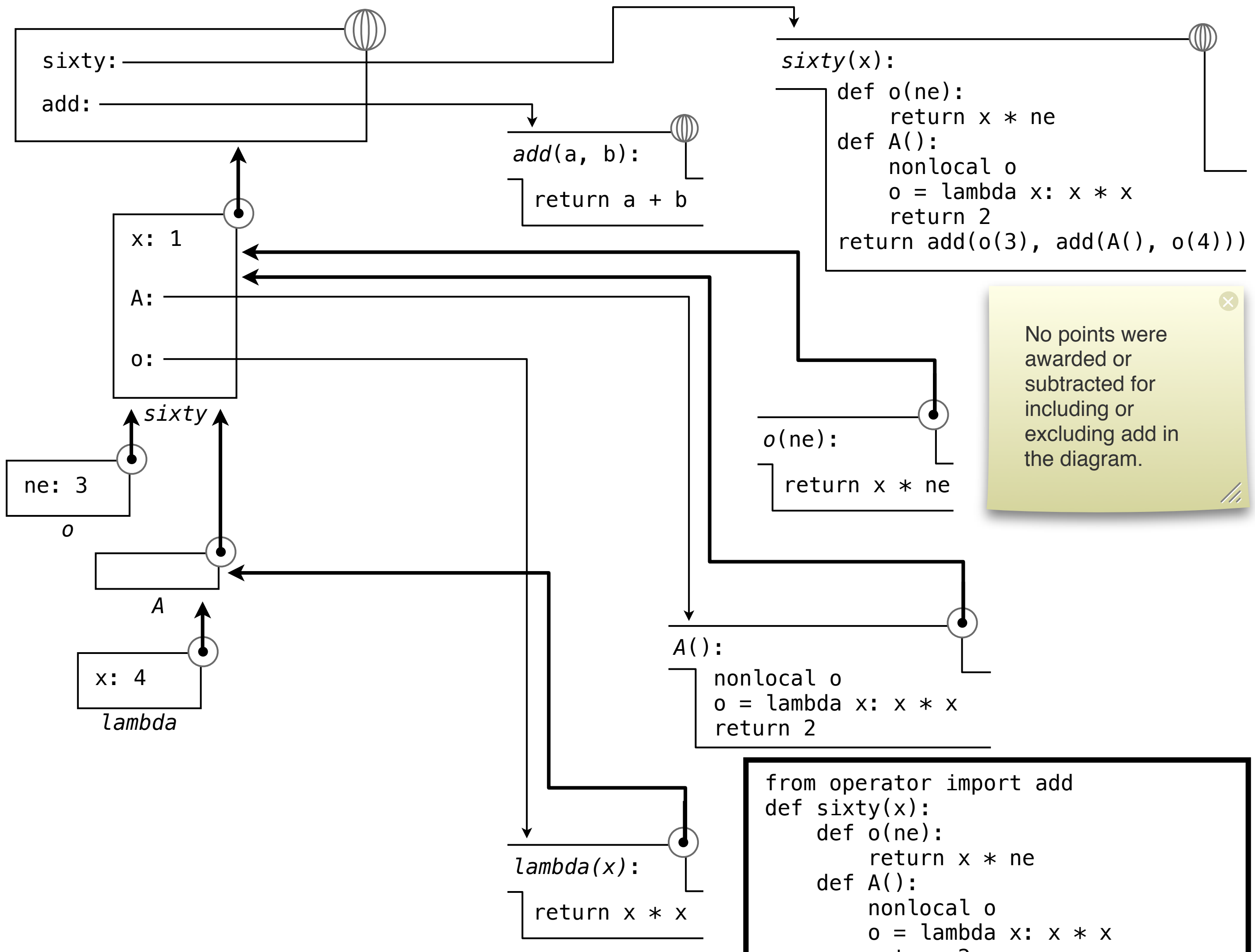
(f) (2 pt) `Big() == Huge()`

`False`

2. (16 points) Environment Diagrams.

(a) (6 pt) Complete the environment diagram for the program in the box below. You **do not** need to draw an expression tree. A complete answer will:

- Complete all missing arrows. Arrows to the global frame can be abbreviated by small globes.
- Add all local frames created by applying user-defined functions.
- Add all missing names in frames.
- Add all **final** values referenced by frames.



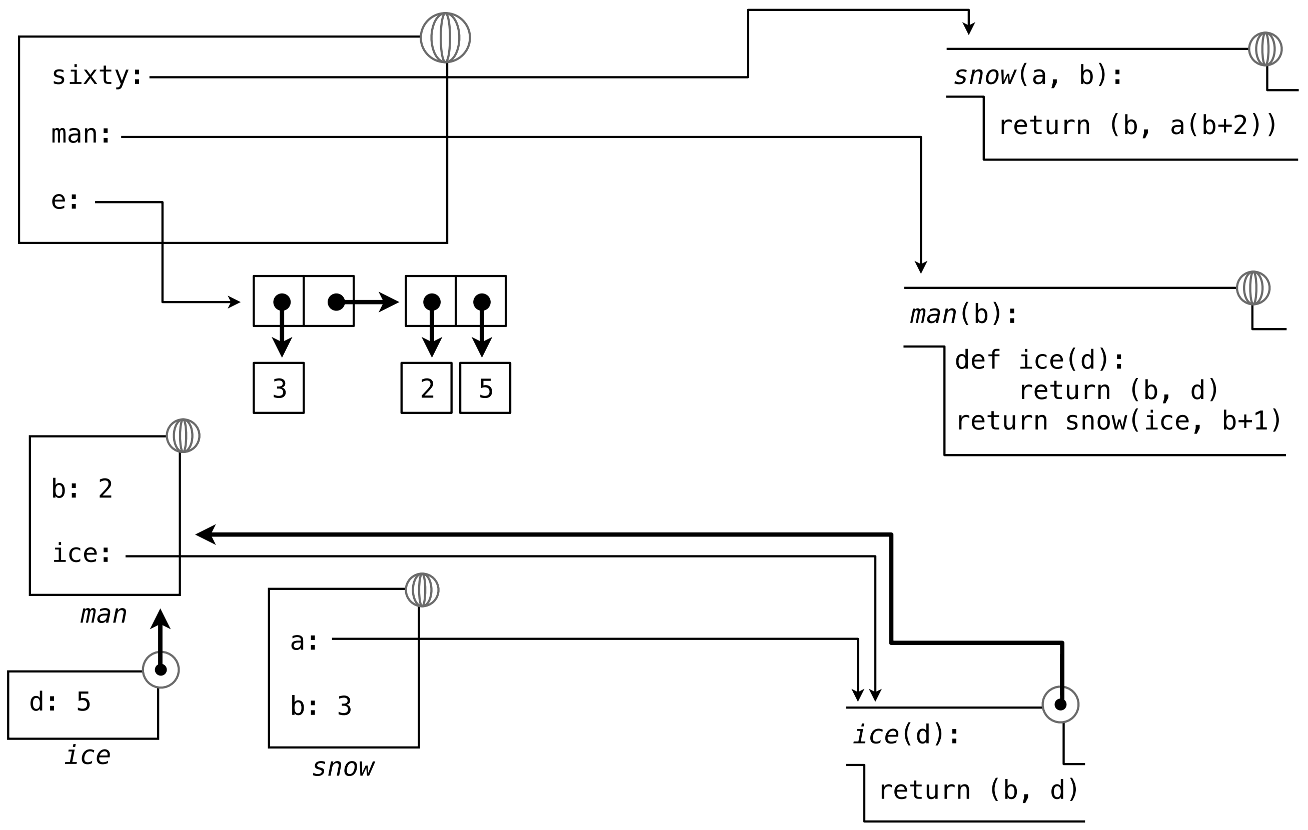
```
from operator import add
def sixty(x):
    def o(ne):
        return x * ne
    def A():
        nonlocal o
        o = lambda x: x * x
        return 2
    return add(o(3), add(A(), o(4)))
sixty(1)
```

(b) (2 pt) What value is returned by evaluating `sixty(1)`? If evaluation causes an error, write "Error."

21

(c) **(6 pt)** Complete the environment diagram for the program in the box below. Assume Python's normal **lexical scoping** rules. You **do not** need to draw an expression tree. A complete answer will:

- Complete all missing arrows. Arrows to the global frame can be abbreviated by small globes.
- Add all local frames created by applying user-defined functions.
- Add all missing names in frames.
- Add all **final** values referenced by frames. Represent tuple values using box-and-pointer notation.



```
def snow(a, b):
    return (b, a(b+2))

def man(b):
    def ice(d):
        return (b, d)
    return snow(ice, b+1)

e = man(2)
```

(d) **(2 pt)** If Python were instead a **dynamically scoped** language, what would be the value bound to `e` in the global frame after this program was executed? Write the repr string of this value.

`(3, (3, 5))`

3. (12 points) Pyth-On Vacation.

Finish implementing this account object in Logo. Unlike a Python `Account` from lecture, the balance of this Logo account is stored as a global variable called `bal`.

The desired behavior of the account is to accept `deposit` and `withdraw` messages. `Deposit` increases the balance by an amount, while `withdraw` reduces the amount if funds are available.

```
? make "john_account make_account 100
? print invoke "withdraw :john_account 40
60
? print invoke "withdraw :john_account 70
insufficient_funds
? print invoke "deposit :john_account 20
80
? print invoke "withdraw :john_account 70
10
```

Invoking a method calls the corresponding named procedure. The `deposit` and `withdraw` procedures are implemented below.

```
to deposit :amt
  make "bal :bal + :amt output :bal
end

to withdraw :amt
  ifelse :amt > :bal [output "insufficient_funds] [make "bal :bal - :amt output :bal]
end
```

(a) **(6 pt)** Add punctuation and the word `run` to `make_account` and `invoke` so that the account object behaves as specified. You may add quotation marks, square brackets, colons, parentheses, and the word `run`, but **no other words**. *Do not remove any words. Multiple calls to* `run` *may be needed.*

```
to make_account :balance
  make  "bal   :balance
  output  [run   sentence   :message   :amount]
end

to invoke :message :account :amount
  output   run   :account
end


OR


to make_account :balance
  make  "bal   :balance
  output  [sentence  :message  :amount]
end

to invoke :message :account :amount
  output  run  run  :account
end
```

```
OR


to make_account :balance
  make  "bal   :balance
  output  sentence  ":message  ":amount
end

to invoke :message :account :amount
  output  run  :account
end
```

(b) **(6 pt)** The D33P language includes three types of tokens: open parentheses, close parentheses, and integers. An expression is well-formed if it contains balanced parentheses, and each integer correctly indicates its depth: the number of nested sets of parentheses that surround that integer.

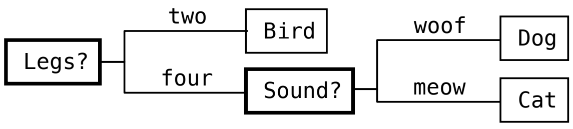
Implement `correct_depth`, which takes a list of tokens as input and returns True if and only if a prefix of the input is a well-formed D33P expression. Assume that the input contains a balanced set of nested parentheses with single-digit positive integers surrounded by parentheses. You only need to check that the integers indicate the correct depths.

Do not change any of the code that is provided. You may not use any `def` *statements or* `lambda` *expressions.*

```
def correct_depth(s, depth=0):
    """Return whether a prefix of list s is a well-formed D33P expression.

    >>> list('(1)')
    ['(', '1', ')']
    >>> correct_depth(list('(1)'))
    True
    >>> correct_depth(list('(2)'))
    False
    >>> correct_depth(list('((2)((3)))'))
    True
    >>> correct_depth(list('((2)(3))'))
    False
    >>> correct_depth(list('((3)(2))'))
    False
    >>> correct_depth(list('(((3)((4))(3))(2)((3)))'))
    True
    """
    first = s.pop(0)
    if first != '(':
        return depth==int(first)

    while s[0] != ')':
        if not correct_depth(s, depth+1):
            return False
    s.pop(0)
    return True

OR

    depth = 1
    ok = True
    for c in s:
        if c == '(':
            depth += 1
        elif c == ')':
            depth -= 1
            if depth == 0:
                break
        else:
            if int(c) != depth:
                ok = False
                break
    return ok
```

4. (18 points) Interpreters. *It is possible to complete each part of this question without completing the others.*

The *Decider* language applies *decisions*, which are tree-structured rules that allow complex classification schemes to be decomposed into hierarchies of lookups. For instance, the problem of classifying household pets might use the following "decision" to distinguish among birds, cats, and dogs.



To decide what sort of animal something is, we look up its *features*. We first check whether it has two legs or four. If it has four legs, we check what sound it makes. Applying a decision always begins at the root (left).

All values in *Decider* are dictionaries with string-valued keys, such as the dog `fido` and the bird `tweety`.

```
fido =   {'name': 'fido',   'legs': 'four', 'sound': 'woof'}
tweety = {'name': 'tweety', 'legs': 'two'}
```

A *decision* is a dictionary that has `'?'` as a key bound to its decision feature. A decision also contains options. The values of options can be decisions, forming a tree. The `animals` decision below matches the diagram above.

```
four_legged = {'?': 'sound',   'woof': {'kind': 'dog'},   'meow': {'kind': 'cat'}}
animals =     {'?': 'legs',    'two': {'kind': 'bird'},   'four': four_legged}
```

(a) **(4 pt)** The `decide` function takes a `decision` and some `features` (both dictionaries). First, it finds the decision feature `f`. Next, it looks up the option for that feature stored in `features`. Finally, it returns the value for that option. Fill in the *four* missing names in the blanks in `decide`, which raises a `DecisionError` whenever the `decision` cannot be applied to the `features`.

```
def decide(decision, features):
    """Apply a decision to some features, both of which are dictionaries.

    >>> decide(four_legged, fido)
    {'kind': 'dog'}
    >>> decide(animals, fido)  # Returns the four_legged decision
    {'meow': {'kind': 'cat'}, 'woof': {'kind': 'dog'}, '?': 'sound'}
    """
    if '?' not in decision:
        raise DecisionError('Decision has no ? -- not a decision.')

                  decision
    f = _____________________________['?']
    if f not in features:
        raise DecisionError('Features does not contain the decision feature.')

                         features
    option = _____________________________[f]
    if option not in decision:
        raise DecisionError('Decision does not contain selected option.')

                      decision                           option
    return _____________________________ [ _____________________________ ]

class DecisionError(Exception):
    """An error raised while applying a decision."""
```

(b) (6 pt) The `animals` decision is a depth-two tree that contains `four_legged` as the value for its `'four'` option. The `decider_apply` function repeatedly applies the result of a decision to some features until it finds a result that cannot be applied. In this way, it traverses a tree-structured decision.

Implement `decider_apply`, which must call `decide` within a `try` statement. This function repeatedly applies the result of `decide` to the `features`. It returns the result of the last successful call to `decide`. If `decision` cannot successfully be applied to `features` at all, return `decision`.

You may not use any `def` *statements or* `lambda` *expressions in your implementation.*

```
def decider_apply(decision, features):
    """Traverse the decision by applying sub-decisions to features.

    >>> decider_apply(four_legged, fido)
    {'kind': 'dog'}
    >>> decider_apply(animals, fido)
    {'kind': 'dog'}
    >>> decider_apply(animals, tweety)
    {'kind': 'bird'}
    >>> decider_apply(fido, tweety)  # fido is not a decision, so return it
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    """

    error = False
    while not error:
        try:
            decision = decide(decision, features)
        except DecisionError:
            error = True
    return decision
```

(c) **(6 pt)** All expressions in *Decider* are dictionaries. *Compound expressions* contain the keys ‘`operator`‘ and ‘`operand`‘. Two compound expressions, `big_pet_exp` and `kind_of_pet_exp`, appear below.

```
pets = {'?': 'size', 'small': tweety, 'large': fido}
big_pet_exp = {'operator': pets, 'operand': {'size': 'large'}}
kind_of_pet_exp = {'operator': animals, 'operand': big_pet_exp}
```

To evaluate a compound expression, evaluate its operator, evaluate its operand, then apply the `decision` dictionary that is the value of the operator to the `features` dictionary that is the value of the operand. All expressions that are not compound expressions are self-evaluating.

Complete the implementation of `decider_eval` by filling in the missing expressions.

```
def decider_eval(exp):
    """Evaluate a Decider expression.

    >>> decider_eval(fido)
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    >>> decider_eval(big_pet_exp)
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    >>> decider_eval(kind_of_pet_exp)
    {'kind': 'dog'}

    >>> decisions = {'?': '?', 'legs': animals, 'size': pets}
    >>> sub_exp = {'operator': decisions, 'operand': {'?': 'legs'}}
    >>> nested_exp = {'operator': sub_exp, 'operand': fido}
    >>> decider_eval(nested_exp)
    {'kind': 'dog'}
    """

              'operator' in exp and 'operand' in exp
    if ________________________________________________________________________:

                              decider_eval(exp['operator'])
        decision = ______________________________________________

                              decider_eval(exp['operand'])
        features = ______________________________________________

        return decider_apply(decision, features)

    else:

        return exp
```

(d) **(2 pt)** To what value does the following Python expression evaluate? Write its repr string.

```
decider_eval({'operator': pets, 'operand': {'size': 'small'}})
```

```
{'legs':  'two', 'name':  'tweety'}
```

5. (10 points) Concurrency.

The following two statements are executed in parallel in a shared environment in which `x` is bound to `3`.

```
>>> x = x + 1
>>> x = x + 2 * x
```

(a) (2 pt) List all possible values of `x` at the end of execution.

4, 9, 10, 11, 12 (answers omitting 11 were marked as correct)

(b) (2 pt) From the values you listed in (a), list all possible *correct* values of `x` at the end of execution.

10, 12

The following program manages concurrent access to a shared dictionary called `grades` that maps student names to their 61A grades. The list `roster` contains the names of all students who have assigned grades.

```
grades = {}
roster = []
grades_lock, roster_lock = Lock(), Lock()
```

```
def add_grade(name, grade):
    grades_lock.acquire()
    grades[name] = grade
    roster_lock.acquire()
    if name not in roster:
        roster.append(name)
    roster_lock.release()
    grades_lock.release()
```

```
def remove_grade(name):
    roster_lock.acquire()
    grades_lock.acquire()
    if name in grades:
        grades.pop(name)
    if name in roster:
        roster.remove(name)
    grades_lock.release()
    roster_lock.release()
```

For each set of statements (left and right) executed concurrently below, circle all of the problems that *may* occur, or circle *None of these are possible* if none of the listed problems may possibly occur.

(c) (2 pt)

`>>> add_grade('eric_k', 1)`	`>>> add_grade('steven', 2)`

(A) Deadlock

(B) Both processes simultaneously write to `grades` or `roster`

(C) [✓ None of these are possible]

(d) (2 pt)

`>>> add_grade('stephanie', 3)`	`>>> roster.append('eric_t')`
	`>>> grades['eric_t'] = 4`

(A) Deadlock

(B) [✓ Both processes simultaneously write to `grades` or `roster`]

(C) None of these are possible

(e) (2 pt)

`>>> add_grade('phill', 5)`	`>>> add_grade('aki', 6)`
	`>>> remove_grade('aki')`

(A) [✓ Deadlock]

(B) Both processes simultaneously write to `grades` or `roster`

(C) None of these are possible

6. (12 points) Iterators and Streams.

(a) (4 pt) The generator function `unique` takes an iterable argument and returns an iterator over all the unique elements of its input in the order that they first appear. Implement `unique` **without** using a `for` statement. *You may not use any* `def`*,* `for`*, or* `class` *statements or* `lambda` *expressions.*

```
def unique(iterable):
    """Return an iterator over the unique elements of an iterable input.

    >>> list(unique([1, 3, 2, 2, 5, 3, 4, 1]))
    [1, 3, 2, 5, 4]
    """

    observed = set()
    i = iter(iterable)
    while True:
        el = next(i)
        if el not in observed:
            observed.add(el)
            yield el
```

The function `sum_grid` takes a stream of streams `s` and a length `n` and returns the sum of the first `n` elements of the first `n` streams in `s`. `Stream` *and* `make_integer_stream` *are defined in your study guide.*

```
def sum_grid(s, n):
    total = 0
    for _ in range(n):
        t = s.first
        for _ in range(n):
            total = total + t.first
            t = t.rest
        s = s.rest
    return total

def make_integer_grid(first=1):
    def compute_rest():
        return make_integer_grid(first+1)
    return Stream(make_integer_stream(first), compute_rest)
```

(b) (2 pt) What is the value of `sum_grid(make_integer_grid(1), 4)`?

64

(c) (2 pt) Define a mathematical function $f(n)$ such that evaluating `sum_grid(make_integer_grid(1), n)` performs $\Theta(f(n))$ addition operations.

$f(n) = n^2$

(d) **(4 pt)** The function `repeating` returns a `Stream` of integers that begins with `start` and increments each successive value until `stop` would be reached, at which point it returns to `start` and repeats.

Cross out lines from the body of `repeating` so that it correctly implements this behavior.

```
def repeating(start, stop):
        """Return a stream of integers that repeats the range(start, stop).

        >>> s = repeating(3, 6)
        >>> s.first, s.rest.first, s.rest.rest.first, s.rest.rest.rest.first,
        (3, 4, 5, 3)
        >>> s.rest.rest.rest.rest.first
        4
        """
        def make_stream(current):

                def compute_rest(next):

                def compute_rest():

                        nonlocal start

                        next = current+1

                        next = current % (stop-start)

                        if next > stop:

                        if next == stop:

                                next = start

                                start = next

                        return make_stream(start)

                        return make_stream(next)

                        return Stream(current, make_stream(start))

                        return Stream(current, make_stream(next))

                return Stream(current, compute_rest)

                return Stream(current, compute_rest())

                return compute_rest

                return compute_rest()

        return make_stream(start)
```

Solution:

```
def make_stream(current):
    ~~def compute_rest(next):~~
    def compute_rest():
        ~~nonlocal start~~                                                                    (optional)
        next = current+1
        ~~next = current % (stop-start)~~
        ~~if next > stop:~~
        if next == stop:
            next = start
            ~~start = next~~                                             (ok if "nonlocal start" crossed out)
        ~~return make_stream(start)~~
        return make_stream(next)
        ~~return Stream(current, make_stream(start))~~  (ok if "return make_stream(next)" not crossed out)
        ~~return Stream(current, make_stream(next))~~   (ok if "return make_stream(next)" not crossed out)
    return Stream(current, compute_rest)
    ~~return Stream(current, compute_rest())~~            (ok if "return Stream(...)" not crossed out)
    ~~return compute_rest~~                               (ok if "return Stream(...)" not crossed out)
    ~~return compute_rest()~~                             (ok if "return Stream(...)" not crossed out)
return make_stream(start)
```