CS164 First Midterm Exam
Fall 2014

October 28th, 2014

- Please read all instructions (including these) carefully.
- **This is a <u>closed-book exam</u>. You are allowed a one-page, one-sided handwritten cheat sheet.**
- Write your name and login on this first sheet, and your login at the top of each sheet.
- No electronic devices are allowed, including **cell phones** used merely as watches.
- Silence your cell phones and place them in your bag.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are **14** pages in this exam and **3** questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. **Do not** use any additional scratch paper.

LOGIN: _________________________

NAME: _________________________

Problem	**Max points**	**Points**
1	**40**	
2	**30**	
3	**30**	
Sub Total	100	

Question 1: Prototypes and Friends [40 points]

This question will test your understanding of inheritance in Lua. A tip on error messages: Whenever we ask you to write the output of a program, use the string "Error" to represent any error that your Lua program might produce.

Part A [8 points] What is the output of the following Lua program, which uses standard Lua inheritance?

```
Plant = {yearsRemaining = 1}
function Plant:new (o)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    return o
end
function Plant:bloom()
    self.yearsRemaining = 0
end

Perennial = Plant:new({yearsRemaining = 5})

basil = Plant:new()
coneflower = Perennial:new()
lavender = Perennial:new()

print("basil: "..tostring(basil.yearsRemaining)) -- '..' is string concat in Lua
basil:bloom()
print("basil: "..tostring(basil.yearsRemaining))

print("coneflower: "..tostring(coneflower.yearsRemaining))
coneflower:bloom()
print("coneflower: "..tostring(coneflower.yearsRemaining))

function Perennial:bloom()
    self.yearsRemaining = self.yearsRemaining - 1
end

print("lavender: "..tostring(lavender.yearsRemaining))
lavender:bloom()
print("lavender: "..tostring(lavender.yearsRemaining))
```

```
basil: 1
basil: 0
coneflower: 5
coneflower: 0
lavender: 5
lavender: 4
```

Part B [4 points] Put a check next to all the tables that are accessed to execute the field lookup in line 22.

☐Plant table [✓ Perennial table] ☐basil table [✓ coneflower table] ☐lavender table

Part C [4 points] Put a check next to all the tables that are accessed to execute the field lookup in line 32.

☐Plant table ☐Perennial table ☐basil table ☐coneflower table [✓ lavender table]

Part D [9 points] If we append the following code to the end of the **Part A** code, what additional output will the program produce?

```
33  Annual = Plant:new({yearsRemaining = 1})
34  function Annual:new (o)
35      o = o or {}
36      setmetatable(o, o) -- changed
37      self.__index = self
38      return o
39  end
40
41  petunia = Annual:new({yearsRemaining = .5})
42  print("petunia: "..tostring(petunia.yearsRemaining))
43  petunia:bloom()
44  print("petunia: "..tostring(petunia.yearsRemaining))
```

```
petunia: .5
Error
```

Part E [6 points] Below, please write a version of line 12 that does not use any colons. Replacing the old line 12 with your modified line 12 must produce a program that exhibits exactly the same behavior.

```
12  Perennial = Plant:new({yearsRemaining = 5})
```

```
Perennial = Plant.new(Plant,{yearsRemaining = 5})
```

Part F [9 points] Is it possible to write the program from **Part A** in a language that uses traditional class-based inheritance, rather than prototype-based inheritance? Argue why or why not. If not, point to a specific range of lines that prevents us from writing the Part A program using classes.

No. A method in a prototype object (Perennial) changes during execution. A class-based language uses stable method definitions throughout execution. Lines 26-28 and their effects would not be reproducible with class-based inheritance.
(If you talk about reflection here, you can give a different answer.)

Question 2: Tail Calls and Desugaring [30 points]

This question deals with the interplay between desugaring and optimizations.

Part A [5 points] What is tail call optimization? In what situations would a developer notice if an interpreter implemented tail call optimization? Do not write outside the box. Shorter is better.

Tail call optimization prevents a new stack frame being created, whenever a function call is in the return position. One situation in which a developer would notice tail call optimization is during recursive functions. Without tail call optimization, the stack may grow large enough to consume all memory. But with tail call optimization, the stack remains at a bounded size. Another situation is code which relies primarily on callbacks.

Part B [5 points] Given the following cs164 code, list out all lines which contain calls that can be safely transformed to tail calls before execution.

```
1   function foo(arg) {
2     def x = arg + 1
3     bar(x)
4   }
5
6   function bar(arg) {
7     baz(arg) + 1
8   }
9
10  function baz(arg) {
11    arg
12  }
```

Lines: 3

Part C [5 points] Tail call optimization can get tricky with control flow, such as **if**, **for** and **while**. Given the following code which contains such control flow, which of the underlined calls can be converted into a tail call? Explain why the other cannot be converted.

Hint:

- The return value of a cs164 **if**, **for** or **while** is the last statement executed inside the body.

```
1   function qux(arg) {
2     if (arg == 0) {
3       0
4     } else {
5       bar(arg)
6     }
7   }
8
9   function corge(arg) {
10    while (arg > 0) {
11      arg = arg - 1
12      qux(arg)
13    }
14  }
```

Line 5, Line 12 cannot since the while loop may repeat again.

Part D [6 points] During the desugaring phase of your interpreter, we reduce all control flow (such as for, while, and if) to lambdas. Write the equivalent cs164 program that is produced when the **qux** function from the code above is desugared. (The body of **qux** should not contain **if** after desugaring.)

```
function qux(arg) {
  (ite(arg == 0, lambda() { 0 }, lambda() { bar(arg) }))()



}
```

Part E [9 points] Once desugaring is complete, tail call optimization is simple since there are no control flow constructs to handle except for lambdas and calls. Fill in the algorithm below which takes in bytecode, and identifies which calls should be converted to tail calls.

Tips:

- To identify a call to a tail call, simply set the “tailcall” field of the call’s bytecode instruction to true
    - `inst.tailcall = true`
- You can assume that the last instruction in any block of bytecode is a return instruction
- You can also assume that each instruction has a unique target register

To familiarize you with the bytecode format, we provide an example of CS164 converted to bytecode instructions.

```
function foo(x)
{
  x + 1
}

foo()
```

```
[
  // define function foo
  {type: ‘lambda’, args: [‘x’]
    body: [
      {type: ‘id’, name: ‘x’, target: ‘r1’},
      {type: ‘int-lit’, value: ‘1’, target: ‘r2’},
      {type: ‘add’, op1: ‘r1’, op2: ‘r2’, target: ‘r3’},
      {type: ‘return’, value: ‘r3’}
    ],
    target: ‘r4’
  },
  {type: ‘def’, name: ‘foo’, value: ‘r4’, target: ‘r5’},

  // call function foo
  {type: ‘id’, name: ‘foo’, target: ‘r6’},
  {type: ‘call’, function: ‘r6’, args: [], target: ‘r7’},
  {type: ‘return’, value: ‘r7’}
]
```

```
function tailcall(insts) {
  var returnReg = insts[insts.length - 1].value

  for (var i = 0; i < insts.length; ++i) {
    var inst = insts[i];
    var type = inst.type;

    if (type == ‘call’) {
      if (inst.target == returnReg)
        inst.tailcall = true;



    } else if (type == ‘lambda’) {
      tailcall(inst.body);




    }
  }
}
```

Question 3: Coroutines [30 points]

In this problem, we are interested in solving the *same fringe* problem. Two binary trees have the same fringe if they have exactly the same leaves reading from left to right.

For instance, consider the trees **t1** and **t2** in the following 164 code:

```
# leaves
def leafA = { leaf = "a" }
def leafB = { leaf = "b" }
def leafC = { leaf = "c" }

# subtrees
def tAB = { leaf = null, left = leafA, right = leafB }
def tBC = { leaf = null, left = leafB, right = leafC }

# tree 1:
#      *
#     / \
#    *   c
#   / \
#  a   b
def t1 = { leaf = null, left = tAB, right = leafC }

# tree 2:
#    *
#   / \
#  a   *
#     / \
#    b   c
def t2 = { leaf = null, left = leafA, right = tBC}
```

Both **t1** and **t2** have the same fringe, namely, the sequence **[a, b, c]**.

You will write methods that can be used to solve the same fringe problem using coroutines.

Part A [8 points] As a starting point, we give you below a function to print the fringe of a tree:

```
def printFringe(tree) {
  if (tree.leaf != null) {
    print(tree.leaf)
  } else {
    printFringe(tree.left)
    printFringe(tree.right)
  }
}
```

Complete the following 164 code which uses coroutines to return an iterator of fringe elements for one tree.

Your implementation must:

- use coroutines: Fringe elements are yielded one by one to the resumer.
- return **null** to indicate the end of the stream.
- be lazy: Don't precompute all elements at the first call. Compute one element in each call to the iterator.

```
def fringeCo(tree) {
  # TODO: Complete this
  if (tree.leaf != null) {
    yield(tree.leaf)
  } else {
    fringeCo(tree.left)
    fringeCo(tree.right)
  }
  null









}

def fringeIter(tree) {
  def co = coroutine(fringeCo)
  lambda() { resume(co, tree) }
}
```

Part B [14 points] Now, you will write a function that traverses two trees concurrently, and returns an iterator over pairs of elements, one element from each fringe in order.

Your implementation must:

- take a list of (two) trees as argument.
- enumerate pairs of elements up to (and including) the first pair that includes a **null** element.
- use **fringeIter** from Part A.
- return **null** to indicate the end of the stream.
- be lazy, as in Part A.

You may use the syntax **{x, y}** to construct a list with the two elements **x** and **y**.

Example:

input	**output**

`def trees = {t1, t2}`
`for (e in fringePairsIter(trees)) {`
`  print e`
`}`
	`{0: a, 1: a}`
	`{0: b, 1: b}`
	`{0: c, 1: c}`
	`{0: null, 1: null}`

`def trees = {t1, leafC}`
`for (e in fringePairsIter(trees)) {`
`  print e`
`}`
	`{0: a, 1: c}`
	`{0: b, 1: null}`

```
def fringePairsCo(trees) {
  # TODO: Complete this
  def iter1 = fringeIter(trees[0])
  def iter2 = fringeIter(trees[1])
  while (True) {
    def e1 = iter1()
    def e2 = iter2()
    # yield {e1, e2}, or in plain 164:
    def res = { }
    res[0] = e1
    res[1] = e2
    yield(res)
    if (e1 == null || e2 == null) {
      yield(null)
    }
  }



}

def fringePairsIter(trees) {
  def co = coroutine(fringePairsCo)
  lambda() { resume(co, trees)}
}
```

Part C [8 points] Generalize your code from the previous part to compare fringes for any number of trees.

Given a list of N trees as argument, your implementation must:

- return lists (of length N) that contain the next element from each fringe.
- enumerate all lists up to (and including) the first one that includes a **null** element.
- use **fringeIter** from Part 1.
- return **null** to indicate the end of the stream.
- be lazy, as in Part 1.

Assume you have access to the following functions: **map**, **exists**, **append**. These functions do **not** modify their arguments. Here are examples of their usage:

input	**output**
`print map({1, 2, 3}, lambda(x) {x + 1})`	`{2, 3, 4}`
`print exists({1, 2, 3}, lambda(x) {x == 2})`	`true`
`print exists({1, 2, 3}, lambda(x) {x == 4})`	`false`
`print append({1, 2, 3}, 4)`	`{1, 2, 3, 4}`

Example:

input	**output**

`def trees = {t1, t2, leafC}`
`for (e in fringeTuplesIter(trees)) {`
`  print e`
`}`
	`{0: a, 1: a, 2: c}`
	`{0: b, 1: b, 2: null}`

```
def fringeTuplesCo(trees) {
  # TODO: Complete this
  def iterators = map(trees, fringeIter)
  def isNull(x) { x == null }

  while (True) {
    def tuple = {}
    for (iter in iterators) {
      tuple = append(tuple, iter())
    }
    yield(tuple)
    if (exists(tuple, isNull)) {
      yield(null)
    }
  }




}

def fringeTuplesIter(trees) {
  def co = coroutine(fringeTuplesCo)
  lambda() { resume(co, trees)}
}
```